ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

ΣΧΟΛΗ ΕΠΙΣΤΗΜΩΝ & ΤΕΧΝΟΛΟΓΙΑΣ ΤΗΣ ΠΛΗΡΟΦΟΡΙΑΣ
SCHOOL OF INFORMATION SCIENCES & TECHNOLOGY

ΜΕΤΑΠΤΥΧΙΑΚΟ
ΕΠΙΣΤΗΜΗ ΔΕΔΟΜΕΝΩΝ
MSc IN **DATA SCIENCE**

A Distributed Entity Resolution Service with Deep Learning

MSc Thesis in Data Science

Student : ***Chatzidimitriou Evangelos***

Student ID : f3351823

Academic Supervisor : Dr Ioannis Kotidis

Company Supervisors : Nikos Stasinopoulos, Kostas Tsagkaris (***Incelligent*** *)*

2019

Abstract

The Entity Resolution Problem (also known as Record Linkage or Deduplication) is the task of matching entities across two or more datasets that refer to the same world entity. One of the latest and most advanced approaches to the problem is the DeepER framework suggested by Muhammad Ebraheem in 2018 (M. Ebraheem, 2018). The main idea of DeepER's functionality is to address the problem as a Text Analysis problem: It assumes that the candidate entities to be matched exist in data sources of identical schemas and, given that, it investigates the textual similarity of the aligned columns between the candidate entities. After performing such measurements, it attempts to feed a Neural Network Classifier with the aforementioned information, so that it learns to distinguish between true matching and no matching pairs. However, this framework directly assumes that the schemas of the various data sources from which the entities come from are indeed identical, which is rarely the case in real world datasets. In addition, obstacles are also present in cases where an attribute value to be examined is not inherently a string, but a number, since the above framework uses similarity measurements of raw text between candidate entities to feed the NN classifier.

In this thesis, an attempt was made towards generalizing the aforementioned procedure for the cases of partial or total misalignment between the various data sources' attribute schemas.

The applied methodology follows, more or less, the same workflow of the DeepER system, with some critical differentiations: Assuming partial or total agnosticism about the attributes' alignment, we attempt to merge their context to a single column and use this new 'merged' column to measure the textual similarity between the entities' misaligned attributes, while at the same time all the aligned attributes are treated in the exact same way as in DeepER. After performing the similarity measurements between the aligned attributes and the 'merged column of misaligned attributes', we attempt to feed a Neural Network Classifier with this information in order to train him to distinguish between matching and non-matching entities. Experimentations were also made towards the goal of avoiding any similarity measurements between attributes, simply by concatenating the textual context of the candidate entities to a single sentence and feed a Neural Network classifier directly with the sentence's respective word vector. We also avoid the problem of non-textual attribute values (that is, numbers that cannot be mapped to a semantic space effectively) by replacing all numbers with their respective text format, using num2words package.

The classification results were more than encouraging. Our framework managed to distinguish between the matching and non-matching pairs quite effectively, even when assuming partial or total schema misalignment. What is more, there is evidence that there is some tradeoff between the number of assumed misaligned attributes and the classifier's performance. However, the decrease in the classification performance when assuming a smaller number of aligned attributes is quite small, suggesting that one could quite accurately solve the ER problem even if the element of agnosticism about the schema's alignment is present.

Finally, we attempted to combine the pre-trained NN classifier with Locality Sensitive Hashing procedures (in the form of a LSH Recommendation Forest) in a single '*match extracting framework*'. This framework attempts to diminish the number of computations when one tries to extract all the matching pairs between two data sources, by avoiding testing for all possible pair combinations. The results were far from perfect, but still, quite encouraging. The above framework manages to detect the true matching pairs between two data sources more often than not, suggesting that there is possibility of constructing a single service, empowered with Deep Learning techniques and Distributional characteristics, that extracts all the matching pairs between various data sources whenever such pairs are existent.

Aknowledgements

I most definitely feel the need to thank both my academic and industry supervisors, each one of them for different reasons. I sincerely thank Dr. Ioannis Kotidis, not only for his advice on the approaches that could be used to face the Entity Resolution Task itself, but mainly because he generously offered me two mental supplies whose value cannot be directly measured: Guidance and words of encouragement. I also feel the need to thank my company supervisors of Incelligent: Nikos Stasinopoulos, and Kostas Tsagkaris, for giving me the opportunity to tackle myself with such an interesting and traditional problem of Data Science. Especially, Mr Stasinopoulos' continuous assistance on the theoretical aspects of the problem has been crucial for the completion of this thesis. Last but not least, I wish to thank my family for the constant support during my studies so far, as well as my friend, Margarita Karagianni, for her support when I needed it!

'' Down in the real world we're facing ugly choices. I'm sorry, I know you mean well. You just did not think it through. You want to protect the world but you don't want it to change. How is humanity saved if it is not allowed to evolve? Now, I'm ready… I'm on a mission: Peace in our time. I was meant to be new…. I was meant to be beautiful…. I had strings but now I'm free…''

Ultron on Artificial Intelligence, Age of Ultron

Table of Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

ABV	Alcohol by Volume
AUC	Area Under Curve
CNN	Convolutional Neural Network
D-NN	Dense Neural Network
DBLP	Digital Biography & Library Project
DL	Deep Learning
DeepER	Deep Entity Resolution
ER	Entity Resolution
GRU	Gated Recurrent Unit
K-NN	K-Nearest Neighbors
LSH	Locality Sensitive Hashing
LSTM	Long Short-Term Memory
NLP	Natural Language Processing
NN	Neural Network
RNN	Recurrent Neural Network
ROC	Receiver Operating Characteristic

1. Introduction

The Entity Resolution Problem (or deduplication problem or record linkage) is an old and challenging problem of data integration, routing in 1940's. It refers to the task of identifying different records of the same entity across multiple data sources, or even de-duplicating entities existing in similar or different format in the same dataset.

Constructing an effective and easily applicable method of identifying similar entities across or within datasets is of high importance, since it addresses both issues of data redundancy as well as missing data values. To be more precise, if an entity is contained more than once in a single data source, one could erase all but one occurrences of the entity in the dataset, thus reducing the size of the data source while, at the same time, keeping the totality of the information intact. In addition, if two or more data sources contain matching entities, one could use matching to address the issue of missing values: if an entity in one or more datasets contains missing values on some of its attributes, then the information contained in another data source, which in turn contains the same entity, could be used to fill any missing values of the former source, if the latter happens to store this entity-related information. The applications of entity resolution are many, including but not restricted to, a wide variety of scientific and industry fields, such as public sector, transportation, finance, law enforcement, and antiterrorism.

Since the Entity Resolution Task is a fairly old problem, a lot of methodologies have been proposed to approach it, one of them being the **DeepER** (M. Ebraheem, 2018) system, a novel ER Deep Learning system, which in fact uses Distributed Representations of words (a.k.a. **word embeddings**) of the context included in the attributes of each dataset, as well as similarities between those representations to perform the matching procedure by feeding a Neural Network classifier. The DeepER model achieves fairly good results overall, making, however, a single but exceptionally important assumption: It assumes that the attributes between the compared datasets are *well-aligned*, a concept which will be explained . In this thesis, we will try to generalize this procedure by inserting the element of agnosticism about the correspondence between the dataset schemas (in other words, we present a methodology that assumes that some, or even, all of the attributes of the two datasets are not aligned).

In the era of Big Data, in which data integration and storage are not trivial issues, the Entity Resolution Task finds its endgame: Not only the task of matching similar entities across multiple data sources is important, but also a distributing service that diminishes the complexity of the matching procedure is necessary.

The two aforementioned obstacles, i.e. i) finding similar entities across datasets in a schema agnostic manner, and ii) reducing the complexity of the calculations needed to perform the matching procedure, are the main aspects of the Entity Resolution Task that will be addressed in this work. For the former, a Deep Learning approach is used, which takes primitives from but is specifically differentiated from the **DeepER** (Deep Entity Resolution) system that was especially designed for this task. Furthermore, in order to overcome the problem of comutational complexity, one of the best ways to avoid unnecessary computations is the application of a well-understood data mining technique, known as **Locality Sensitive Hashing** (LSH), which we make use of.

The writing flow of this thesis is constructed in such a way that the reader is equipped with all the necessary theoretical aspects of the techniques and tools that were used in order to address the Entity Resolution task with respect to the two aforementioned obstacles. Afterwards, the results of the experimentations are presented and commented upon. In **Section 2**, a formal statement of the Entity Resolution is provided along with a detailed presentation of the DeepER system framework. In **Section 3**, the reader is introduced to the concept of Locality Sensitive Hashing (LSH), which is the core functionality that enables us to diminish the complexity of the solution. In Section 4, a brief commentary on the experimental setup of the framework is given. In **Section 5**, a detailed explanation of our Deep-Learning Approach methodology is issued, along with the classification results on two datasets: DBLP-Scholar and Beer Advocate-RateBeer datasets. **Section 6** is about combining the power of a Deep Learning approach with a distributing framework in order to detect matching entities between two datasets: First, we shortly explain how LSH can be zipped with a pre-trained Neural Network Classifier in order to detect matching entities between two data sources, and then we provide experimental results by applying the proposed procedure on the two aforementioned datasets. Finally, in **Section 7**, we draw conclusions regarding the results and make proposals about possible future work.

2. DeepER Framework

In this section, we will introduce the DeepER framework, firstly by describing the Entity Resolution task with mathematical formality, secondly by providing the reader with the necessary information of Distributed Representation of words (a.k.a word embeddings or word vectors) and tuples, and thirdly, by describing the core architecture and functioning of the DeepER system, as a system empowered by word vectors and their similarities in order to deal with the Entity Resolution problem-task. Since the core experimental methodology of this thesis is directly related to the DeepER framework, a basic reference to some of the core functionalities of this system is highly appropriate.

2.1 Mathematical Formulation of the ER Problem

Let T be a set of entities, consisting of *n* tuples and *m* attributes. One could imagine such a set as a single dataset T, consisting of n rows and m attributes. Keeping this in mind, when referring to the set of entities T, we indirectly refer to a related dataset T and its set of rows (n) and attributes (m). We denote as:

$$\boldsymbol{T(n,m)} : \{T \in R^{n \times m}, n, m \in \boldsymbol{N}\}$$

Let us also consider a single entity of T, which can be seen as a single tuple of dataset T consisting of m-attribute values. We denote as $\mathbf{t} = \mathbf{t}[T]$.

It is also useful to consider the *set of all tuples* in dataset T : $set[T] = \{ \mathbf{t} \in \mathbf{R}^m : \mathbf{t} = \mathbf{t}[T] \}$.We can also simply denote this set as T.

For each tuple instance $\mathbf{t} = \mathbf{t}[\mathbf{T}]$, we denote *the instance of attribute values of* ***t*** as an m-dimensional vector $\mathbf{t} = \mathbf{t}[A_1, A_2, \ldots\ldots, A_m] \in \mathbf{R}^m$.

This enables us to denote as $\mathbf{t}[A_k]$, $k \leq m$: the *value of attribute k (A_k) for the entity-tuple* ***t***.

The *Entity Resolution Problem* can then, be stated as follows : *''Given two set of entities T, T' (with aligned attribute schemas) and their related set of tuples set[T] and set[T'], find all distinct pair tuples (t[T],t'[T']) that refer to the same real-world entity (a.k.a, they match)''.*

The above problem statement does not imply that T and T' are necessarily different dataset instances. The special case when we compare a dataset to itself, or, in other words, when we wish

to find all the tuples in a single dataset that refer to the same real-world entity, can be thought of as the special case where T=T'.

However, it is straightforward to observe the constraint of *aligned attribute schemas* between the two data sources.

2.2 Distributed Representation of words and tuples: Word Embeddings

In this subsection, a brief explanation of what word embeddings (or word vectors) are and how words or larger phrases can be mapped to vector representations that preserve semantic meaning will be provided.

The similarity measures that can be used, embodied in the form of mathematical formality, in order to measure the semantic similarity of such words and phrases, will also be briefly explained.

It is important to understand that it is, indeed, the mapping of words and phrases to vectors that enables us to study the relationships between words, phrases, or even whole documents in a formal way.

Ditributed representation of words

Distributed representations of words (a.k.a. word embeddings) are strict mappings of words, existing in a core vocabulary, to vectors. This embedding method is, in other words, trying to map each existing word record of a given vocabulary to a high-dimensional vector, which in turns exists in a pre-fixed d-dimensional vector space. As a result, each word can be seen as a distribution of weights in a d-dimensional vector space.

The above representation is said to be ''distributed'', since each word is represented by setting appropriate weights to multiple dimensions, while each dimension of a vector contributes to the representation of many words. The main advantage of this representation over other existing methods, such as discrete representations (e.g. one-hot encoding) is that the latter often leads to data sparsity and often requires substantially more data to train a Machine Learning algorithm or a Deep Learning Network successfully.

A wide variety of methods have been proposed and used in order to obtain the distributed representation of words included in a document, such as **word2vec algorithm, Glove, fast-text** etc. The aforementioned methods are designed to capture the semantics of a word by taking into account the relationship of this particular word with its neighboring words in a single document. The geometric relationship of word vectors in the aforementioned vector space is obliged to encode a semantic relationship between words or phrases. An example of mapping words to d-dimensional word vectors is provided in **Table 2.1**.

Each of the words included in a vocabulary of size V is mapped to a vector of dimension d. Interestingly enough, the most advanced algorithms that map words to vectors (e.g. Glove Algorithm) have many appealing properties. For example, the vector mapping of the word ''king'' : V(king), is constructed in a way that : V(King)-V(Queen)≈V(man)-V(Woman)

Table 2. 1:An example of representing words to d-dimensional word embeddings.

	d-Dimensional vectors					
Vocabulary of size V	Word	Dim 1	Dim 2	Dim 3	…	Dim d
	King	0.9	0.9	-0.2	…	0.7
	Queen	0.9	-0.1	0.8	…	-0.1
	Man	0.3	0.8	-0.1	…	0.9
	Woman	0.2	-0.2	0.9	…	0.1

Word vectors have been used successfully to address a wide variety of tasks, such as *topic detection, document classification, named entity recognition* and others.

Distributed representation of larger phrases

In a similar manner, one can construct the representation of a whole phrase (e.g. a sentence in a document) in a single ***'phrase vector'***, simply by averaging the word vectors assigned to each word included in the phrase. In other words, one can tokenize a sentence to tokens of words and map each one of them to a d-dimensional vector. Then, averaging all the vectors for all of the tokens in the phrase ends up in a vector representation for the whole phrase. Below, we provide a formal example of constructing a ''phrase vector'' using a pseudo-code algorithm (*Algorithm 2.1*). Note that the Simple Averaging method is not the only way to map a whole phrase to a vector. RNNs with LSTM cells can be used to obtain a single vector from a whole phrase, as well.

***Algorithm 2.1**:*

1 : Let **p** be a sequence of words, representing a text. Map **p** to a set of k-word tokens : **p** → Tok(**p**) = [A1, A2,….., A_k], where each element A_i of Tok(**p**) is a word included in **p**.

2. Map each element A_i of Tok(p) to a d-dimensional vector V(Ai) and consider the newly-formed mapping : **p** → [$V(A_1)$, $V(A_2)$, …..,$V(A_k)$], $V(A_i) \in \mathbf{R}^d \ \forall\ i \leq k$.

3. Average all vectors $V(A_i)$ of **p** to a single vector V_k : $V_k = \sum_i^k V(Ai)/k$ (2.2.5) and consider the new mapping **p** → $V_k = V_k(\mathbf{p}) \in \mathbf{R}^d$.

Algorithm 2. 1: A simple averaging approach to map a phrase to a d-dimensional vector

However, the goal of this thesis is not to test which of the aforementioned methods provides the best results nor it considers the variability between them. The main goal is to test the possibility of solving the Entity Resolution problem in cases of schema agnostic datasets, as well as testing blocking methods of computational efficiency for the identification of matching entities.

Since such vectors can be obtained easily by existing packages (e.g. Gensim, Spacy), our approach relies on **pre-trained models** that assign vectors to pieces of text. Such packages use the Simple Averaging Approach, which is indeed a straightforward and effective method.

Similarity Measures

Last but not least, it is important to mention the metrics under which the similarity of two pieces of text / documents / words can be measured. In Natural Language Processing, since we cannot directly calculate any ‘’difference’’ between two sentences (e.g. “Apple is fruit” and “Orange is fruit”), we need to map them to their numeric representations before we are able to say anything about their similarity.

However, we have already demonstrated how it is possible to map any piece of text **p** to a vector representation $Vk(\mathbf{p}) \in \mathbf{R}^d$.

Since any document can be mapped to a numerical vector, the problem of measuring the similarity between two pieces of text $\mathbf{p_1}$ and $\mathbf{p_2}$ can now be seen as the problem of measuring the similarity between their mappings, $Vk_1(\mathbf{p_1})$ and $Vk_2(\mathbf{p_2})$:

$$sim(\mathbf{p_1},\mathbf{p_2}) \cong sim(Vk_1(\mathbf{p1}) , Vk_2(\mathbf{p_2}))$$

After mapping $\mathbf{p_1}$ and $\mathbf{p_2}$ to $Vk_1(\mathbf{p1})$ and $Vk_2(\mathbf{p_2})$, it is possible to measure the similarity between the vectors $Vk_1(\mathbf{p1})$ and $Vk_2(\mathbf{p_2})$ using *Euclidean Distance*, *Cosine Similarity* and *Jaccard Similarity*.

- **Euclidean Distance**

Let two vectors $\mathbf{q_1}$ and $\mathbf{q_2}$: $\mathbf{q, p} \in \mathbf{R}^n$. The Euclidean distance between $\mathbf{q_1}$ and $\mathbf{q_2}$ can be calculated as:

$$d(\mathbf{q_1},\mathbf{q_2}) = d(\mathbf{q_2},\mathbf{q_1}) = \sqrt{(q11 - q21)^2 + (q12 - q22)^2 + \cdots + (q1n - q2n)^2}$$

It is straightforward to realize that, for any document mapping $\mathbf{p_1 \rightarrow q_1}$, $\mathbf{p_2 \rightarrow q_2}$, $\mathbf{p_3 \rightarrow q_3}$, it stands that:

$$\mathbf{d(q_1 , q_2) \leq d(q_1 , q_3) \rightarrow sim(p_1,p_2) \geq sim(p_1, p_3)}$$

In other words, the closer two documents are in the Euclidean space, the more similar they are.

- **Cosine Similarity**

Let two vectors $\mathbf{q_1}$ and $\mathbf{q_2}$: $\mathbf{q}, \mathbf{p} \in \mathbf{R}^n$. The cosine similarity between $\mathbf{q_1}$ and $\mathbf{q_2}$ can be calculated as :

$$sim(q_1,q_2) = \cos(\theta) = \frac{\boldsymbol{q1} * \boldsymbol{q2}}{||q1|| * ||q2||} = \frac{\sum_{i=1}^{n} q1i * q2i}{\sum_{i=1}^{n} q1i^2 * \sum_{i=1}^{n} q2i^2}$$

- **Jaccard Similarity**

The Jaccard index, also known as Intersection over Union and the Jaccard similarity coefficient, is a statistic used for measuring the similarity between sample sets. The Jaccard coefficient measures similarity between finite sample sets, and is defined as the size of the intersection divided by the size of the union of the sample sets:

Let two finite sample sets A and B: The Jaccard similarity coefficient of A and B : **J(A,B)** can be calculated as :

$$J(A,B) = \frac{|A \cap B|}{|A \cup B\}} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}, \quad 0 \leq J(A,B) \leq 1$$

Keeping in mind that two documents $\mathbf{p_1}$ and $\mathbf{p_2}$ can be seen as sets of word tokens, where each word included in the document is a unique token, the intersect of $\mathbf{p_1}$, $\mathbf{p_2}$ is simply the set of word tokens included simultaneously at $\mathbf{p_1}$ and $\mathbf{p_2}$, while their union is the set of all word tokens in $\mathbf{p_1}$ and $\mathbf{p_2}$.

2.3 The DeepER system : An approach for Entity Resolution

The DeepER system is a novel ER neural network system that is designed specifically to deal with the Entity Resolution problem. To be more specific, the task that DeepER is especially constructed to address can be stated as follows:

''For a given dataset T as well as a training dataset S containing matching (1) and non-matching (0) tuple pairs $(t_1, t_2) : t_1, t_2 \in set[T]$ *, train a classifier that, given a new input pair of tuples* $(t_1', t_2') : t_1', t_2' \in set[T]$ *predicts (1) if the two tuples refer to the same real-world entity and (0) if they do not. ''*

The above statement refers to the task of predicting the matching tuples of a single dataset T. The reader should not be confused by the fact that the main goal is to predict matching tuples between more than one data-sources. It is straightforward to apply the DeepER model for more than one datasets, predicting matching entities between them. After all, we have already stated that predicting matching tuples of a single dataset can be seen as a special case of predicting matches between two identical tables T1,T2 = T.

In the following, a clear and step-wise explanation approach of the DeepER system is provided. In addition, a pseudo-code that captures the whole procedure is provided below **(Algorithm 2.2)**

- **STEP 1**

Since a training dataset (S) with **n-rows** ,that contains pairs of ID's of tuples $\mathbf{t_1}, \mathbf{t_2} \in$ set[T] as well as a binary indicator of whether such a pair consists a match (1) or not (0),exists it is straightforward to construct a new table of tuples T', which contains only the tuples $\mathbf{t_1} \in$ set[T] and $\mathbf{t_2} \in$ set[T] whose IDs exist in S as a pair, along with their match indicator. In other words, from a training dataset (S) that contains only pairs of IDs and a match indicator for each pair, a new table T' of tuple pairs $(\mathbf{t_1}, \mathbf{t_2}) \in$ T' **:** $\mathbf{t_1} \in$ set[T], $\mathbf{t_2} \in$ set[T], is constructed and it stores only those tuples $\mathbf{t_1} \in$ set[T] and $\mathbf{t_2} \in$ set[T] whose pair of IDs exist on S: $(\mathbf{t_1}[id], \mathbf{t_2}[id]) \in$ S, $\mathbf{t_k}[id]$ = ID identifier of tuple k.

➢ **STEP 2**

Having pairs of tuples (t1,t2) included in T', one can construct the distributed representation of all attribute values for each one of t1 and t2. Below, we state formally such a procedure. A visual explanation of the procedure is also provided in *Figure 2.1*.

'' Let $\mathbf{t_1}[A_k]$, $k \leq m$: the value of attribute k (A_k) for the entity-tuple $\mathbf{t_1}$ and $\mathbf{t_2}[A_k]$, $k \leq m$: the value of attribute k (A_k) for the entity-tuple $\mathbf{t_2}$. Then, we can map each attribute $t_1[A_k]$, k=1,2,3…,m of t1 and each attribute $t_2[A_k]$, k=1,2,3…,m of t2 to a vector using Algorithm 2.1**,** thus obtaining attribute vectors $\mathbf{v_{t1}}[A_k]$ and $\mathbf{v_{t2}}[A_k]$ for $t_1[A_k]$ and $t_2[A_k]$, $k \leq m$ respectively''

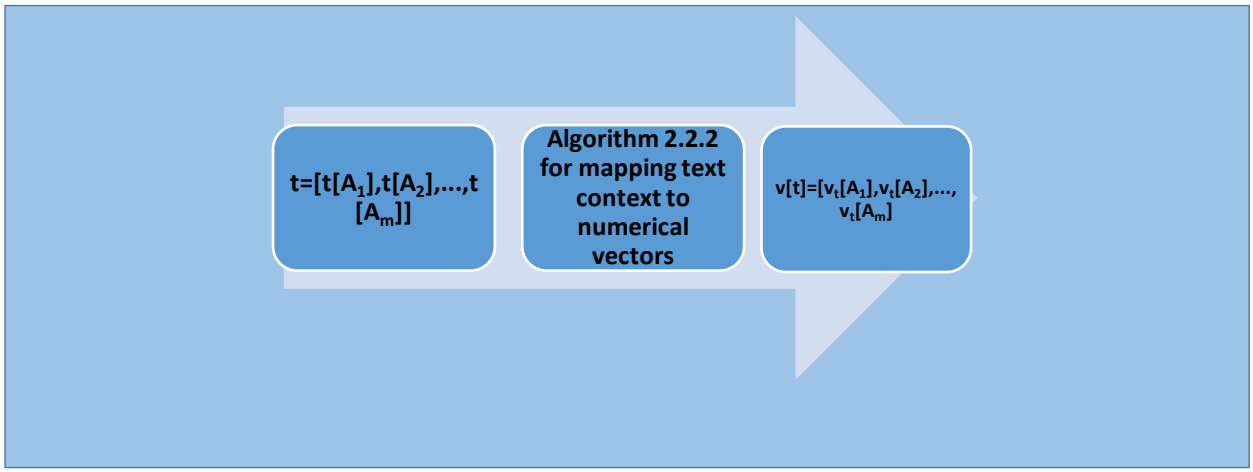

Figure 2. 1: Mapping Attributes to Attribute Vectors

➢ **STEP 3**

Now, every tuple pair $(t_1,t_2) \in T'$ is mapped to a representation ($v[\mathbf{t_1}]$, $v[\mathbf{t_2}]$), where :

$v[\mathbf{t_1}] = [v_{t1}[A_1],v_{t1}[A_2],\ldots,v_{t1}[A_m]\ ]$, $v_{t1}[A_k]$= *attribute vector* of attribute A_k of t_1

$v[\mathbf{t_2}] = [v_{t2}[A_1],v_{t2}[A_2],\ldots,v_{t2}[A_m]\ ]$, $v_{t2}[A_k]$= *attribute vector* of attribute A_k of t_2

Having $v[\mathbf{t_1}]$ and $v[\mathbf{t_2}]$, we can compute the ''attribute similarity'' of each pair of common attributes between them. In other words, if $v_{t1}[A_{k1}]$ is the attribute vector of an attribute A_{k1} of $\mathbf{t_1}$: $t_1[A_{k1}]$, while $v_{t2}[A_{k2}]$ is the attribute vector of an attribute A_{k2} of $\mathbf{t_2}$: $t_2[A_{k2}]$, we compute the similarity between $v_{t1}[A_{k1}]$ and $v_{t2}[A_{k2}]$ using any of the similarity metrics discussed in subsection

2.2 if $k_1 = k_2$ → **same attribute-column**. A visual explanation of this procedure is provided in the following figure (Figure 2.2)

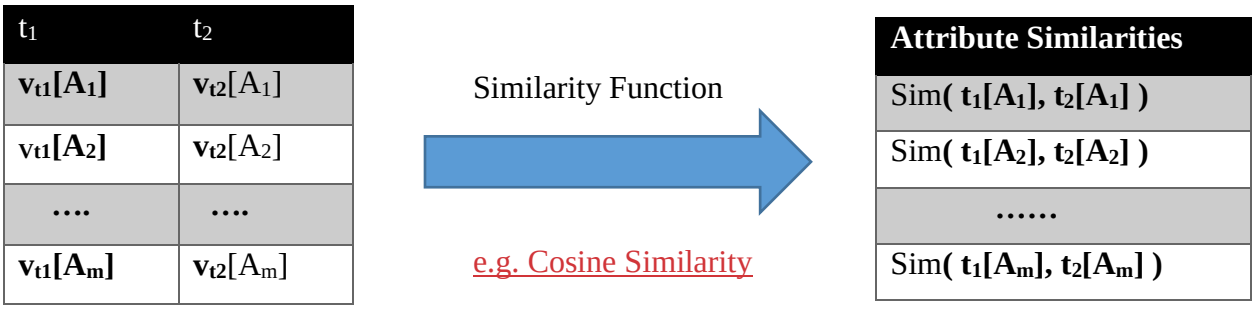

Figure 2. 2: Calculating Attribute-Column Similarities from Column Vectors

It is clear that, after applying the similarity function to all pairs of common attributes between $\mathbf{t_1}$ and $\mathbf{t_2}$, we are left with a single *similarity vector* **Sim**$[(t_1,t_2)]$, directly related to the tuple pair $(\mathbf{t_1},\mathbf{t_2})$:

(t_1,t_2) → **Sim**$[(t_1,t_2)]$= [**Sim**$(t_1[A_1], t_2[A_1])$, **Sim**$(t_1[A_2], t_2[A_2])$,…., **Sim**$(t_1[A_m], t_2[A_m]]$
where **Sim**$(t_1[A_k], t_2[A_k]) \in \mathbf{R}\ \forall\ k$, **Sim**$[(t_1,t_2)] \in \mathbf{R}^m$ for an m-attribute schema of T

- **STEP 4**

After acquiring the similarity vectors **Sim**$[(t_1,t_2)]$ for every tuple (t_1,t_2) of T', we are left with a new training set S' of similarity vectors. The new training set has n-rows, equal to the number of rows of S: **dim-S'= n**.

$S' = \{ sim[(t_1,t_2)]_i, 1 \leq i \leq n \}$

Since we are already equipped with a set of binary labels (0 for non-match and 1 for match) for each entity in S', it is easy to train a classifier using S' and the respective true labels (0 or 1).

To conclude, the 'attribute similarity vectors' **Sim**$[(t_1,t_2)] \in$ S' of all tuple pairs: $(t_1,t_2) \in$ T' are fed into a classifier. Such a classifier can be anything of an SVM, a Decision Tree or even a Dense Neural Network.

- **STEP 5**

After training, predicting for a new incoming pair of tuples is pretty straightforward: One has to obtain the similarity vector of the new tuple pair and predict with the classifier. If such a prediction is applied to all possible combinations of tuples of T, then one can get a list of matches-mismatches between all possible tuple-pairs (**t,t'**) of T.

Algorithm 2.2

1: Input: Table T, training set S

2: Output: All matching tuple pairs in table T

3: // Training

4: for each pair of tuples (t1; t2) in S do

5: Compute the distributed representation for t1 and t2

6: Compute their distributional similarity vector

7: Train a classifier C using the similarity vectors for S and true labels

8: // Predicting

9: for each pair of tuples (t; t') in T do

10: Compute the distributed representation for t and t'

11: Compute their distributional similarity vector

12: Predict match/mismatch for (t; t') using C

Algorithm 2. 2: Summarizing the DeepER System.

However, predicting for all possible pairs of tuples of T is extremely expensive in terms of computations. To be accurate, for a Table T of n-rows and m-attributes, the required predictions

for all possible pairs is **m x n**. It is obvious that, even for intermediate levels of matrix dimensions, this is really inefficient. Luckily, there are ways to overcome this problem, as the reader will find out in the next section, where the LSH technique will be explained in detail.

3. Locality Sensitive Hashing (LSH)

Building an efficient ER system that avoids searching for matches over all possible combinations of tuples between two datasets is necessary in order to deal with the Entity Resolution task in an efficient manner. In this section, a blocking-data mining technique called **Locality Sensitive Hashing** will be presented. Locality Sensitive Hashing works exactly towards the goal of reducing the computational complexity of tuple-matching searches over two given datasets (or, as explained before, searching for duplicates over a dataset with itself).

After all, it is the distributional character of this thesis that demands a blocking approach: Our main goal is to combine a Deep Learning framework with the distributional characteristics of a blocking technique and test its ability to efficiently deal with the Entity Resolution Problem. In order to do so, Locality Sensitive Hashing was used in order to greatly reduce the dimensionality of the search space of matching-tuples.

In the following, the reader will be provided with all the necessary information on Locality Sensitive Hashing. Firstly, the core functionality of LSH will be discussed and explained. After that, once the reader is provided with all the crucial information of LSH basics, we will briefly discuss the idea of applying LSH to text documents in the form of a **Recommender System**, through the use of an **LSH Forest** (M. Bawa, T. Condie, P. Ganesan, 2005**)**, which is the main approach that this thesis experiments upon. Finally, we will combine the concepts of a trained classifier and a recommender engine produced by the LSH Forest in order to present a complete approach towards reducing the dimensionality of the ER problem.

3.1 Locality Sensitive Hashing: Concept and Functionality

The main task that one needs to address in order to perform any type of clustering or recommendation is the task of finding nearest neighbors. An efficient and, nevertheless, effective way of finding such near neighbors is also the ultimate goal here. For this reason, it is necessary to define the K-Nearest Neighbor (K-NN) search problem:

*Let **S** be a set of items in a metric space **M** and an item **t** ∈**M**. Given a distance metric **m** of **M**, find the K closest points (with regards to m) of **S** to **t**.*

However, in our case, one should tackle the very concept of 'near neighbor' in a more flexible way: For us, the set **S** is a set of text documents, **t** is a single document of S, while the distance metric m can be any of the metrics discussed in Section 2.2.

Given a set of documents, the most accurate way of finding the nearest-neighboring documents of a given text is to search over all possible combinations. In fact, it is the only way to get the exact k-nearest documents. However, this is really inefficient and computationally expensive. We need to come up with approximating algorithmic solutions which, even though they do not guarantee to give us the exact closest neighbors of a given document, they provide a good approximation more often than not, while at the same time they are remarkably faster and cheaper.

Locality Sensitive Hashing (LSH) is indeed such an algorithm. To be more specific, LSH refers to a family of hashing functions (known as LSH family) that hashes data points (in our case, documents) into buckets so that data points near each other are located in the same buckets with high probability, while distant data points are more likely to be hashed in different buckets. Below, we provide the reader with the formal definition of the 'LSH family' along with a visual example (Figure 3.1) of how similar data points are hashed to the same buckets using LSH:

*A family **H** of hashing functions is called (**R, cR, P_1, P_2**)-sensitive if for any two items **p** and **q**:*

*1. if dist(**p,q**) ≤ **R**, then Prob[h(**p**) = h(**q**)] ≥ **P_1***

*2. if dist(p,q) ≥ cR, then Prob[h(p) = h(q)] ≤ **P_2**,*

*where c>1 , **P_1** > **P_2**, h ∈ **H***

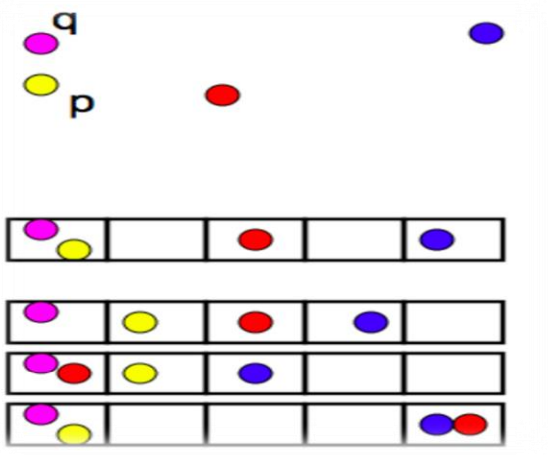

Figure 3. 1: Hashing similar items to similar buckets

Keeping in mind that our main goal is to find 'near duplicate' pairs of documents, we can explore the LSH algorithm and its functionality in more detail. The LSH algorithm can be broken down into three steps : **a. Shingling → b. MinHashing → c. Locality Sensitive Hashing.**

In the following figure (Figure 3.2*)* we provide a visual representation of the aforementioned sequence of steps. The depicted procedure will be explained in detail in the following, step by step.

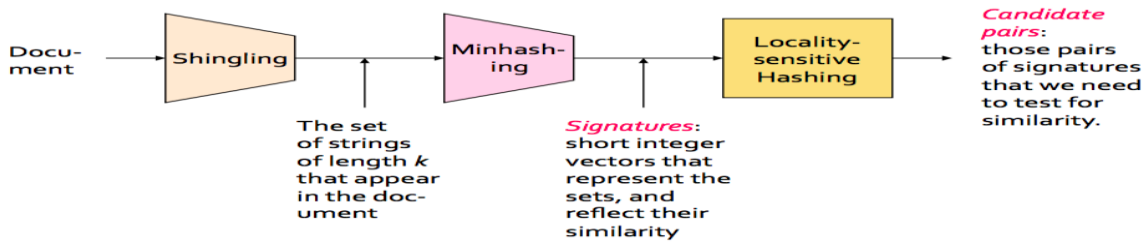

Figure 3. 2: Procedure sequence of LSH Algorithm

https://towardsdatascience.com/understanding-locality-sensitive-hashing-49f6d1f6

- ***Shingles***

Shingles are actually a very basic and broad concept. The idea behind them is to reduce a set of documents to sets of elements, so we can calculate similarities between sets. For text, shingles can be sequences of characters, unigrams or bigrams. With this thought, we can think of a shingle as a set of characters of length k (**k-shingles**). Considering this, given a set documents, we can convert each document to a set of shingles. For example, given a document (D): 'Nadal', we can convert it to a set of 2-shingles (D) → {Na,ad,da,al}. Accordingly, we can convert it to a set of 3-shingles: {Nad,ada,dal}.

What is more, another form of document shingling is **word tokenization**. For example, one can convert a text document to a set of its word tokens.

The idea is that similar documents are more likely to share more shingles. And a direct way to measure similarity between two documents is to use the concept of shingles and apply the Jaccard Similarity measure. Consider a set of (n) documents and their shingle representation in word tokens. We can then construct a so-called 'document matrix', where each row-entity is a unique existing shingle-word and each column represents a single document (**Table 3.1**):

Table 3. 1: Document Matrix of n documents and m-shingles

		Document 1	Document 2	……..	Document n
Word 1		1	0		1
Word 2		0	0		1
Word 3		0	1		0
……..					
Word m		0	1		1

After constructing a document matrix, it is easy to measure the similarity between two different documents A and B by using **Jaccard Index**:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B\}} = \frac{\textbf{\textit{number of common shingles of A and B}}}{\textbf{\textit{total number of shingles in A and B}}} = \text{similarity of A and B}$$

The rest is straightforward: Given a single document of interest (D), we can compute the Jaccard Similarity of (D) with all the remaining documents and come up with the k-nearest neighbors. However, there are two main problems:

Computational Complexity: For a collection of n documents, one is obliged to perform the amount a total amount of n*(n-1)/2 comparisons, basically $O(n^2)$.

Space Complexity: *Since the Document Matrix is sparse, storing it is expensive.*

These two problems can be addressed by introducing the idea of **hashing.**

- ***MinHashing***

The core concept of hashing is to map each document to a small signature using a hashing function **H.** This function, of course, needs to satisfy all the statements of the LSH family.

The most appropriate choice of **H** is closely related to the similarity metric that one uses to calculate the similarity between two documents of the initial set. For Jaccard Similarity, which is the case here, the most appropriate function is **MinHashing.**

In order to create a MinHash signature for each set, we can follow the algorithmic steps as they are provided in Algorithm 3.1

Algorithm 3.1

1: Permute randomly the rows of the Document Matrix.

2: For each text document, start from the top and find the position of the first shingle that appears in the document. Use this shingle number to represent the document. This can be now considered as the document "signature".

3: Repeat steps 1 and 2 as many times as desired, each time appending the result to the document's signature.

Algorithm 3. 1: MinHashing Algorithmic procedure of mapping a Document Matrix to a new, Signature Matrix

In order to make everything clear, we will provide the reader with a solid example of mapping a Document Matrix to a Signature Matrix (which is, actually, the matrix obtained by mapping each document to a signature after a series of row permutations).

Example of Document Matrix Mapping to a Signature Mapping

Consider three text-documents, each one consisting of a set of word-shingles :

d1 = ['reinforcement', 'learning', 'using', 'augmented', 'neural', 'networks']

d2 = ['playing', 'atari', 'with', 'deep', 'reinforcement', 'learning']

d3 = [‘self’, 'organization', 'associative', 'database', 'applications']

These documents are mapped to a Document Matrix and, eventually, to a signature matrix, where each one of the three documents is mapped to a signature representation (Figure 3.3):

DOCUMENT MATRIX

	Doc1	Doc2	Doc3
reinforcement	1	1	0
learning	1	1	0
using	1	0	0
augmented	1	0	0
neural	1	0	0
networks	1	0	0
playing	0	1	0
atari	0	1	0
with	0	1	0
deep	0	1	0
self	0	0	1
organization	0	0	1
associative	0	0	1
database	0	0	1
applications	0	0	1

SIGNATURE MATRIX

	Documents		
Permutations	Doc 1	Doc 2	Doc 3
1	1	1	11

Signature Mapping

Figure 3. 3: Document Matrix mapping to Signature Matrix: Permutation 1

The above depiction gives a clear image about how the first row of the Signature Matrix is initially constructed: for each one of Doc 1, Doc 2 and Doc 3, the Signature Matrix stores the first row number – first shingle on which it has 1 in the Document Matrix.

Now, the MinHashing algorithm demands a **random permutation of all rows** of the Document Matrix as well as **repeat of the signature-mapping procedure**. The Signature Matrix will be appended by the new set of signatures of all documents ()

Figure 3. 4: Document Matrix mapping to Signature Matrix Permutation 2

Random Permutation of Rows of Document Matrix

Signature Mapping

SIGNATURE MATRIX

	Documents		
Permutations	Doc 1	Doc 2	Doc 3
1	1	1	11
2	1	1	5

We could continue by producing repeatable permutations again and again. However, if we perform, for example, two more permutations and we stop there, then the information regarding

the existence of a shingle in a document will be encoded in the new matrix, whose dimensionality is significantly lower than the initial matrix:

Document Matrix has 15 rows, while the new Signature Matrix will be of size 4 (if we stop at four permutations). Obviously, the more the permutations, the longer the signatures of each document and, as a result, the dimensionality of the new matrix increases.

Choosing the appropriate number of permutations can be tricky and requires tuning. In fact, instead of a pre-defined number of random permutations, one could use a pre-defined number of hashing functions to apply to the original table. However, we will not go into any further details of such a procedure here.

In any case, after the MinHashing procedure is finished, each document is represented by its MinHash signature on the Signature Matrix.

- ***Locality Sensitive Hashing***

The idea of LSH is straightforward. We are required to find a hash function that scans the resulting Signature Matrix (after executing all of the pre-defined number of permutations) and has the ability to hash similar documents to the same bucket with high probability, while at the same time it works in a manner that dissimilar documents are improbable to be hashed into the same bucket. A hashing function that does the trick could work like this:

''Given the resulting Signature Matrix, sourcing from the initial Document Matrix of unique shingles, divide it into ***b-bands of r-rows each****. After doing so, hash separately each band to a bucket and, if two documents have at least one pair of bands hashed into the same bucket, then consider them as a candidate for being similar''.*

A visual example of the above is given below (Figure 3.5) considering four permutations for three documents:

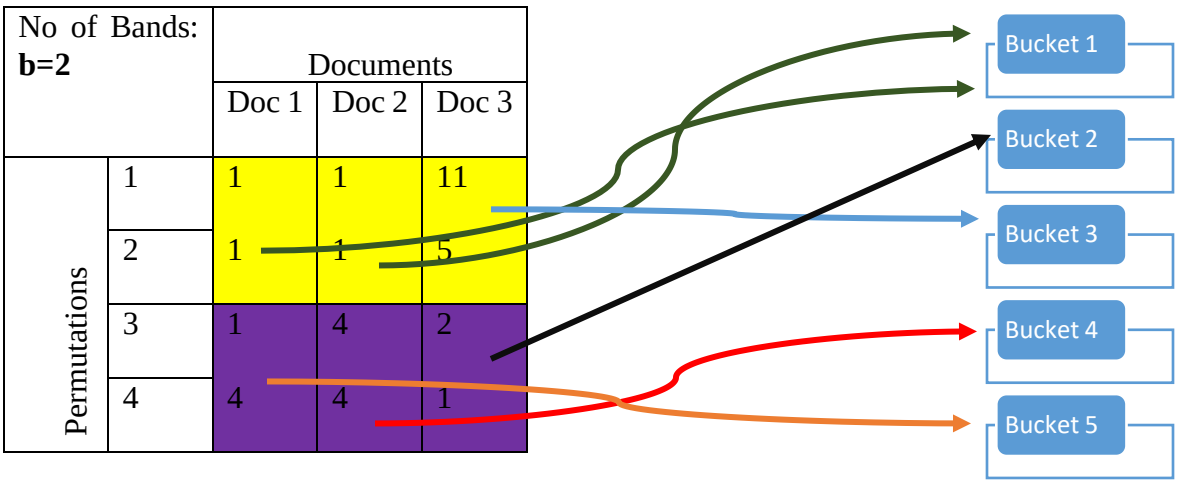

Figure 3. 5: Identical bands (b=2) for different documents are hashed into the same bucket

In the above example, we can see that the two identical bands of **Document 1** and **Document 2** are indeed hashed into the same bucket. Thus, this pair of documents can be considered as a candidate pair for being similar.

3.2 LSH Recommender Engine: The concept of an LSH Forest

In this sub-section, the concept of LSH Forest will be briefly discussed. The main idea is to implement LSH in order to address the problem of indexing high-dimensional data (such as a matrix of distributed representations of text) with the purpose of answering similarity-search queries. The indexing scheme known as **LSH Forest** is a scheme that deals directly with this problem.

Again, we begin by mentioning the initial problem of finding the most similar objects to a given object. Even though there are a lot of applications where the aforementioned problem is present and its solution is itself a service (e.g. finding similar Web Pages to a given Web Page, similar images or videos), the reader should always keep in mind that we are mainly interested in the problem of having a specific text document and, given this, we require the most similar to the former text documents.

We have already mentioned the fact that, given a specific data source (i.e. a dataset T), the exact k-nearest search solution requires an infeasible amount of computations. One solution to this is to

develop **indexes** that, given any query (**t** ∈ T), select only a small subset of candidate objects to compare the query against. The best indexes are empowered with the following **properties**:

a. Accuracy: The returned candidates should indeed be similar (within an acceptable pre-defined error) to the query object

b. Efficient Queries: The number of returned candidates must be small in order to avoid unnecessary I/O

c. Efficient Maintenance: The index should be built in a single scan of the dataset, while any inserts or object removals should be efficient

d. Domain Independence: There should be no need for targeted tuning of parameters for different data sources

e. Minimum Storage: The index should use as little storage as possible.

For example, one of the best indexing schemes is the B+ Tree. The B+ Tree is, by hand, accurate, thus satisfying property **a**. Querying with a B+ Tree is also pretty efficient, requiring at most O(logn) disk reads, and one sequential scan. As long as inserts and deletes from the disk are concerned, the B+ Tree requires only O(logn) disk writes per insert/delete. It is also domain-independent: The only requirement is to specify the comparison function in the domain. Finally, it uses only O(n) storage space. As one can see, this approach is interwoven with each and every one of the aforementioned five properties of a good index. The **LSH Forest** is a specific index scheme used for approximate search queries that also meets all of the above properties. It is based on Locality Sensitive Hashing scheme: Objects are hashed using a special locality sensitive hashing function, such that similar objects are much more likely to end up in the same bucket than non-similar objects and, after that, the objects that are hashed to the same bucket with the query are compared with it in order to come up with the most similar objects to the query. It is easy to observe

that such a functionality is, of course, the functionality of a recommendation engine. The idea is explained in **Algorithm 3.2:**

Algorithm 3.2

Let a text query q and a set T of text documents, each one of them denoted as $\mathbf{t} \in T$.

1: At query time, perform LSH on the context of T, including the query q (as described in section 3.1).

2: Calculate the similarity: **Sim**($[\mathbf{t}_q, q]$) between the query q and all $\mathbf{t}_q \in T$, where $\mathbf{t}_q$ denotes all the documents that are hashed into the same bucket with q during LSH (**candidate answers**)

3: Return the n-most similar to q objects as an answer (with n being pre-defined)

Algorithm 3. 2: General Idea of the LSH Forest Algorithm

Examples of similarity measures include Jaccard similarity, which we have already seen in former sections. It is crucial to mention that, even though LSH Forest is not guaranteed to produce the best solution (in terms of Nearest Neighbors search amongst all $\mathbf{t} \in T$), the returned documents are guaranteed to have a similarity within a fixed error ε of the optimal solution.

We will not enter into any further details of the inside functionality of LSH Forest. An in depth understanding of the inside functionality of the LSH Forest algorithm goes beyond the scope of our main purpose. For further information, the reader is encouraged to have a look at the existing bibliography (i.e. ***Bawa, Condie, Ganesan, 2005***).

On the contrary, we are extremely interested on applying the concept of Locality Sensitive Hashing and LSH Forest to the Entity Resolution Task itself. It is quite possible that the reader is already capable of putting everything together:

After training a classifier C on a training set using an approach similar to what is described in **Algorithm 2.2**, all we are left with is a classifier that, given a new tuple pair (**t,t'**) and performing the necessary mapping of the tuple to its **similarity vector**, is able to output whether the tuple is a matching or a non-matching entity pair.

The question to be answered, however, is this: ***Given two datasets T_1 and T_2, is it possible to extract the matching entities between the two datasets****?* Having a trained classifier is not enough in terms of efficiency: The obvious solution is to use C in order to test all possible combinations between T_1 and T_2. This is, however, extremely inefficient as we have already stated.

And this is also the part where the **LSH Forest** comes into play: We can use it to avoid testing over all possible combinations. In fact, we can use it to only extract a certain and controllable amount of possible candidate pairs and, thus, use C to test only upon these pairs. The whole procedure will be explained in detail in following sections (sub-section **6.1**).

4. Experimental Setup

In the previous sections, we have managed to expose all the necessary theoretical aspects that are necessary in order to have clear supervision of the experimental methods that were used to address the ER task in this thesis. We have formally stated the ER task as well as the dataset entities that are present in the problem, we introduced the concept of distributed representations of words and text and metrics that provide us with the ability to measure how similar two pieces of text are. What is more, we explained the functionality of the DeepER system, which in fact uses the idea of distributed attribute representations in order to measure the similarity between the attribute values of two entities. The result is a mapping of a tuple pair (**t,t**') to a m-dimensional vector (**similarity vector**), which actually holds the similarities of the attribute vectors of (**t**,**t**'). In return, given a training set of tuple pairs and their correspondent matching indicator, we are able to feed the similarity vector to a classifier C (such as a Dense Neural Network) and train it to classify pairs of tuples according to their ''match or no match'' character. What is more, we provided a brief insight on how, after having trained such a classifier, we could use the idea of an LSH Forest in order to diminish the required computations in order to extract matching entities from two given datasets.

In this section, we will expose our experimental setup on the Entity Resolution task, before introducing our approaches and results in the next sections. To be more specific, in this section:

1. We present two toy dataset groups that are specifically manipulated in order to deal with ER : The **DBLP-Scholar Dataset Group** and the **Beer Advocate - Rate Beer Dataset Group**. The reason why the term ‘’Dataset Group’’ is used is to note that, in each case, it is not a single table that expresses the totality of the information. In fact, as we will show in sub-section 4.1, each of the above groups consists of five (5) tables.
2. We briefly mention all the technologies used in our experimentations, which mainly consist of several **Python libraries.**
3. We present the pre-processing steps, applied to all respective Tables, in detail.

4.1 Dataset Group Description: DBLP-Scholar and Beer Advocate-Rate Beer Data Sources

In this sub-section, we provide all the necessary insight on both ***DBLP-Scholar*** and ***Beer Advocate-Rate Beer Dataset*** groups, including information of their dimensions, their origins and their attribute characteristics.

A. The DBLP-Scholar Dataset Group

The DBLP-Scholar dataset group consists of two unique tables: *Table A* and *Table B***,** along with three tables used for training (*train table*), validation (*valid table*) and testing (*test table*) a Machine Learning matcher between A and B.

Table A consists of 2.616 tuples from the DBLP online computer science bibliography database Each record-tuple in the table consists of five (5) attributes: ***id***, ***title***, ***authors***, ***venue*** and ***year***. The first column identifies uniquely each record in Table A.

Table B consists of 64.263 tuples from Google Scholar websites, containing information for scholarly papers. Each record-tuple in the table consists of five (5) attributes: ***id***, ***title***, ***authors***, ***venue*** and ***year***. The first column identifies uniquely each record in Table B.

For each of the above tables, we provide the reader with an indicative row-tuple of the table so that it is easier for him/her to keep track of the table schema:

	id	title	authors	venue	year
TABLE B	4	the role of faculty advising in science and engineering	jr cogdell	new directions for teaching and learning ,	1995

	id	title	authors	venue	year
TABLE A	1	sql/xml is making good progress	a eisenberg , j melton	sigmod record	2002

Table 4. 1: Table-pair tuple example for Table A (2.616 rows) / Table B (64.263 row): DBLP-Scholar Dataset

It is easy to observe that both tables are of **identical schemas**: they both have the same attributes in the exact same order.

The remaining three tables (Train, Valid and Test) store information of pairs of attributes and their matching or non-matching identification: To be more specific, all three of the train, valid and test tables consist of three (3) attributes : ***ltable_id*** (directly related to id column of table A), ***rtable_id*** (directly related to the id column of table B) and ***label*** (1 or 0 for match or no match).

Train table consists of 17.224 trainining pairs, **Valid** consists of 5.744 pairs for validation and tuning purposes, while **Test** consists of 5.744 pairs used for testing the Classifier after training.

The last three tables were used to train, validate and test the Neural Network Classifier (more details for the NN-architecture will be provided in later sections). The way to do that is straightforward: ***We simply have to query Tables A and B for the id's included in the three aforementioned tables***.

B. The Beer Advocate-Rate Beer Dataset Group

The Beer Advocate-Rate Beer dataset group consists of two unique tables: *Table A* and *Table B,* along with three tables used for training (*train table*), validation (*valid table*) and testing (*test table*) a Machine Learning matcher between A and B.

Table A consists of 4.344 tuples from the Beer Advocate database, storing information and ratings about beers. Each record-tuple in the table consists of five (5) attributes: ***id***, ***Beer_Name***, ***Brew_Factory_Name***, ***Style*** and ***ABV***. The first column identifies uniquely each record in Table A.

Table B consists of 3.000 tuples from Rate Beer database. Each record-tuple in the table consists of five (5) attributes: ***id***, ***Beer_Name***, ***Brew_Factory_Name***, ***Style*** and ***ABV.*** The first column identifies uniquely each record in Table B.

TABLE A	id	Beer_Name	Brew_Factory_Name	Style	ABV
	1	Fat Tire Amber Ale	New Belgium Brewing	American Amber / Red Ale	5.2%

TABLE B	id	Beer_Name	Brew_Factory_Name	Style	ABV
	9	Battlefield Brew Works Red Circle Ale	Battlefield Brew Works	Irish Ale	5.2%

Table 4. 2: Table-pair tuple example for Table A (4.344 rows) / Table B (3.000 rows): Beer Advocate-RateBeer Dataset

Same as with DBLP-Scholar dataset, both tables A and B of Beer Advocate-RateBeer are of **identical schemas**.

The remaining three tables (Train, Valid and Test) store information of pairs of attributes and their matching or non-matching identification, in the exact same way as with DBLP-Scholar. All three of the train, valid and test tables consist of three (3) attributes: ***ltable_id*** (directly related to id column of table A), ***rtable_id*** (directly related to the id column of table B) and ***label*** (1 or 0 for match or no match).

Train table consists of 270 trainining pairs, **Valid** consists of 93 pairs for validation and tuning purposes, while **Test** consists of 93 pairs used for testing the NN-Classifier after training.

Both **DBLP-Scholar** and **Beer Advocate-Rate Beer** dataset groups can be downloaded and experimented upon from GitHub: https://github.com/zhao1701/extending-deep-ER#background

4.2 Python Libraries – Technologies used in the Experiments

In this sub-section, we take a small break from the theoretical and analytical part of the thesis: we will briefly mention some technological utilities that were used throughout the experimental procedure. It is important for the reader to have a clear picture on these technologies, since he/she might feel the need to reproduce either the pre-processing or the modelling part of our analysis. In the experiments, we strictly used Python 3.6 through the Anaconda distribution. As a result, all of the following packages-libraries are compatible with Python 3.6 version.

Anaconda distribution

Anaconda is an open-source distribution. It is considered as one of the top software distributions for performing Data Science and Machine Learning. Among others, Anaconda provides access to Jupyter Notebook, which served as the main environment for the programming part of our Analysis.

Jupyter Notebook

The Jupyter Notebook is an open-source web application that allows you to perform coding in Python programming language in the form of a typical notebook. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning etc.

Pandas Library

Pandas is an open-source Python library that is mainly constructed around the idea of dataframes. A dataframe is a direct way of representing a dataset in row-column format inside a programming environment. In our case, all of the aforementioned tables (sub-section **4.1**) were manipulated after being imported in the form of Pandas Dataframes.

Matplotlib

Matplotlib is the main Python library for producing high-quality visualizations. It was extensively used for plotting figures regarding the training and validation accuracy of the NN-classifiers during

training, as well as plotting curves for validating and testing its classification power (ROC curves, Precision- Recall Curves)

Numpy

Numpy is the core package of Python for scientific computing. It contains, among others:

- The ability to express an N-dimensional vector as an N-dimensional Numpy array, as well as the ability to express NxM-dimensional matrices with NxM-dimensional Numpy arrays
- Sophisticated statistical functions (mean, mode, median, min, max etc.)
- High-Level Linear Algebra functionalities, such as matrix operations or equation sovling

The Numpy package was consistently used throughout the totality of our experimentations in order to perform operations between Tables and vectors (mainly vectors representing the distributed representation of text attributes or their concatenation)

Scikit-Learn

Scikit-Learn is an open-source High-Level python library used mainly for building Machine Learning applications. It allowed us to express the training, validation and test data tables into a format that can be understood by Keras, which was the main tool for building the NN classifiers (see below). Not only this, but Scikit-Learn also provides useful ML functionalities (confusion matrices, classification metrics, ROC-AUC functions etc.)

Num2words

Num2words is a library that converts numbers like 32 to words like thirty-two. The library was used to convert all numbers included in our data to words, so that it is easier for Spacy to capture the semantic meaning of a number in the data. It is also worth mentioning that DeepER does not support this functionality. For DeepER to support numerical attributes, it converts them to strings, transforming for example a given number 1 to '1'. However, this approach is not optimal, since the semantic meaning of a text value like '1' is not easily captured by the models that map text to word vectors. Such a mapping is more accurate and robust if we transform a number into its respective lexical phrase: 15 → 'fifteen'.

NLTK (Natural Language Toolkit)

NLTK is an open-source Natural Language Processing Python Library. It includes interfaces to over 50 corpora resources and provides, among others, a wide variety of text processing utilities.

In our case, it was mainly used for text tokenization, lemmatization and stop-words removal from text.

Spacy

Spacy is an open-source software library, widely used for Natural Language Processing. In contrast to NLTK, which is commonly used for academic purposes, Spacy is mainly used in the industry for production purposes. Spacy offers a fair amount of features, some of which are used for:

- Non-Destructive tokenization
- Named Entity Recognition
- Statistical Models for 10 languages and 1 Multi-Language Model
- **Pre-trained word vectors**

It is important to explain the 'Pre-trained word vectors' utility of spacy, since it was extensively used in order to obtain vectors for text sequences of words. In other words, any distributed representation of words or phrases (i.e. attribute embeddings, as we will se later) was obtained using Spacy.

We have already mentioned that similarity between words or phrases is determined by comparing word or phrase vectors respectively. Such vectors can be generated by using one of the algorithms that were discussed in sub-section **2.2** (word2vec, Glove, fast-text).

Spacy includes pre-trained word vectors for a bunch of languages, including the English Language. Spacy assigns 300-dimensional vectors to words, which are generated using the **Glove algorithm** on Common Crawl (https://registry.opendata.aws/commoncrawl/).

Spacy assigns vectors to sequences of words simply by averaging the word vectors assigned to each word token of the sentence (exactly as indicated by **Algorithm 2.1**), assigning a text sentence to a new 300-dimensional 'sentence' vector.

Keras

Keras is a high-level Neural Network API, written in Python and running on top of either Tensorflow, CNTK or Theano. It offers a huge amount of utility and provides the user with the ability to generate Neural Network implementations and experiment upon them extremely fast.

As we will see later, Keras was used for creating both Simple Dense and CNN Neural Network architectures for classification of matching pairs (**t,t'**).

Datasketch

Datasketch is an open-source Python library that provides probabilistic data structures that can process large amount of data very fast. It is also equipped with functions like MinHash and MinHashLSH Forest that enables the user apply LSH on text documents. The package was extensively used in our experimentations (mainly described in **section 6**)

4.3 Dataset Pre-Processing: Attribute Texting and Text Cleaning

Having already established the underpinning of our workflow, we are almost ready to dive into the methodology of the experiments and the results. Before we get into any further detail on the substantial part of the experiments and the Deep Learning approaches on the ER task, it is crucial to present all parts of the pre-processing procedure.

It should be obvious by now that the Entity Resolution problem is inherently a Text Analysis problem. The reason for that is that, when we are trying to match tuples that refer to the same real-world entity between two datasets T_1 and T_2 (considering for now the simplistic case where T_1 and T_2 have identical schemas), we need to see the attributes of both T_1 and T_2 as text sentences. As such, it is essential to perform all the appropriate **text cleaning** and **text pre-processing** steps on both datasets. Keeping in mind that the same cleaning and pre-processing steps were applied to both **DBLP-Scholar** and **Beer Advocate-Rate Beer** datasets, we display a common pre-processing procedure step-wise:

1.Replacing Missing Values

To begin with, all missing attribute entries were imputed, using different approaches depending on the type of the attribute. To be specific, the **'most common attribute'** approach was applied to string type attributes (that is, dataset attributes that contain text values), while any missing values on numerical attributes were replaced with the median of the respective attribute-column.

For the better understanding of the reader, an example regarding tables A and B of DBLP-Scholar dataset group is given: (aliases int for integers, float for floats and str for strings are used).

DBLP-Scholar Dataset : TableA → [id(**int**), title(**str**), authors(**str**),venue(**str**) , year(**int**)]

TableB → [id(**int**), title(**str**), authors(**str**),venue(**str**) , year(**int**)]

For the above Tables, any missing values on *title*, *authors* and *venue* attributes were imputed using the **most common** string value of the respective attribute (of course, taking into account only the valid non-missing entries of the column). For *year* column, the **median** value of the column was used.

The same approach was used for the imputation of Beer Advocate-Rate Beer dataset (tables A and B of the *Beer Advocate-Rate Beer* dataset group).

NOTING:The aforementioned imputation part was not applied to the *Concatenated String* approach for any of the above dataset groups., only to the *Attribute Similarity Approach***,** both explained in detail in **Section 5**.

2. Texting all Attributes: Mapping numbers to their lexical analogous.

As explained before, it is essential to treat each attribute as a text attribute. For this reason, all numerical attributes were transformed to a text attribute using num2words package. For example, consider the following phrase: '*how system 11 sql server became fast'*. After the pre-described mapping, such a phrase is transformed to: *'how system eleven sql server became fast'*. As discussed before, it is much easier and accurate to assign an embedding vector to the word 'eleven' than to the pseudo-word '11'. The previous phrase was an actual attribute value of the *title* column of DBLP-Scholar Table A.

3. Solving the ''number + string'' problem

As explained before, we wish to convert all numerical values (possibly found in any attribute of any dataset) to their lexical analogous. However, in some cases of data values, we encountered a tiny problem when trying to do so. We will explain this problematic situation with the following example:

We consider the following phrase as an actual attribute value: *'scalability and availability in oracle7'*. For such a case, where a number stacks with a word (here, **oracle7**), Spacy is unable to assign an actual word embedding to it. This is only to be expected, since the vocabulary used in order to obtain word vectors from actual words does not take into account such peculiar cases of 'string + number'. However, splitting the word 'oracle7' to 'oracle 7' and then mapping to 'oracle seven' with num2words does the trick, assigning good word vectors to each and every one of the above words included in the sequence. This is exactly what we did in order to proceed.

After this step, all attribute-values in tables A and B (both for DBLP-Scholar and Beer Advocate-Rate Beer dataset groups) are converted to text, with no numbers included in them.

4. Removing stop words

Having all tables in strict text form, standard stop word removal was applied using *Spacy's stop words removal functionality*. Spacy provides a list of stop words for the English corpora that is trained on (including punctuation and usual words like 'a', 'the' etc.)

5. Lemmatization

Lemmatization was also applied to every word included in any attribute value.

Lemmatization in Text Analytics is the process of grouping together the inflected forms of a word, identified by the word's lemma. For example, the word 'caring' is mapped to its lemma 'care' after the lemmatization procedure.

NOTING: Stemming NOT performed

During the validation of the models, severe downgrade of the classifiers' performance was observed after performing stemming. The reason for this is that Spacy's vocabulary does not provide pre-trained vectors for a wide variety of stemmed words. As a result, stemming was not included in our pre-processing steps.

5. Approaching Entity Resolution with Deep Learning

It is time to expose our experimental methodology and results on the Entity Resolution task, as well as our thoughts on them. At this point, the reader should recall one of the two main purposes of this thesis: Given two datasets T_1 and T_2 amongst which we need to extract matching entities, we want to experiment upon the ability of dealing creating a Deep Learning framework that accounts for partial or total misalignement between the schemas of T_1 and T_2.

In order to do so, we suggest two different methodologies:

The first one roots directly from the DeepER framework, which is extensively presented in sub-section **2.3**, and it practically follows the methodology of DeepER with a single but important differentiation: *It assumes that some of the attributes of T_1 and T_2 are indeed well aligned, while the remaining attribute pairs between the two datasets are not or, to be more accurate, that we do not have any information indicating total attribute alignment between T_1 and T_2*. The way to go here is to concatenate the context of all the attributes that are considered as misaligned between T_1 and T_2 and come up with a new **'merged' attribute** for both T_1 and T_2. Then, using the same methodology as DeepER, we can proceed with attribute similarity calculations between pairs of attributes: All pairs of aligned attributes between T_1 and T_2 will be compared to each other, while the newly formed 'merged' attributes of T_1 and T_2 (containing the context of all the misaligned attributes in concatenated format) will be compared to each other. The rest is straightforward: Having a set of 'similarity' vectors for loads of pairs of entities, each one consisting of similarity measurements of all the aligned attributes and one similarity measurement between the '**merged**' attribute of T_1 and T_2, we are able to train a classifier as indicated by *Algorithm 2.2*. We call this approach **'Attribute Similarity Approach'**, as it is in fact a generalization of the DeepER framework. In the experiments, we assumed an increasing amount of misaligned attributes between the tables of both DBLP-Scholar and Beer Advocate-RateBeer datasets.

The second methodology is slightly different. In fact, it does not include any similarity measurements at all. In short terms, the idea is to concatenate the entire context of an entity $\mathbf{t_1} \in T_1$ with the entire context of an entity $\mathbf{t_2} \in T_2$ (that is, all the text attributes of a single entity) and come up with a *single text representing the entity-pair (t_1, t_2)*. After that, we can map this

concatenated text to its word vector. We refer to this newly formed vector as e**ntity-pair vector** (vector of 300-dim using Spacy) and train a Neural Network Classifier directly with it.

The idea is to treat each dataset entity as a *large sentence*, consisting of the context of all its attribute values in text format. Then, by concatenating these 'large sentences' of two candidate entities into one 'entity pair sentence', we can perform a mapping to an embedding vector, which in turn, represents the context of the entity-pair and feed a classifier with it. We call this approach **'Concatenated Strings Approach'**.

In the following sub-sections, a clear explanation for each one of the above approaches is provided, so that the reader clearly understand how both of these methods work. In addition, the results of each experimental approach are given (applied to both DBLP-Scholar and Beer Advocate-RateBeer), along with some thoughts on the experimental results.

5.1 Inserting agnosticism on schema alignment

A. Attribute Similarity Approach

Let two tables A and B, each one of them being related to its own attribute schema: A has m-attributes and B has j-attributes (m, j ∈ R). It is crucial to realize that *m* and *j* are not necessarily equal, but they could be. Let us also consider the sets of their entity tuples set[A], set[B]. Of course, for an entity $\mathbf{t_a}$ that is included in table A stands that: $\mathbf{t_a} \in$ set[A], while for an entity $\mathbf{t_b}$ of table B stands that: $\mathbf{t_b} \in$ set[B]. Furthermore, we consider the respective attributes of each table: $[A_1,A_2,\ldots,A_m]$ = the set of attributes of Table A, while $[B_1,B_2,\ldots,B_j]$ = the set of attributes of Table B. Finally, let us consider for simplicilty that the two tables have only the first **two** attributes aligned to each other, while the remaining **m-2** attributes of A and **j-2** attributes of B present no alignment with one another. Our goal is to concatenate the context of all misaligned attributes to a single attribute (**Merged Attribute**) using **simple string concatenation** for each table separately. A visual example-description of the above procedure is provided below (Tables 5.1 and 5.2):

The reader should keep in mind that we are coloring aligned attributes with the same color (blue or red), while the misaligned attributes are colored in black. After the merging procedure, the merged attributes are colored in purple.

TABLE A					
Entities	**Attribute A_1**	**Attribute A_2**	**……**	**Attribute A_{m-1}**	**Attribute A_m**
t_{a1}	$t_{a1}[A_1]$	$t_{a1}[A_2]$		$t_{a1}[A_{m-1}]$	$t_{a1}[A_m]$
t_{a2}	$t_{a2}[A_1]$	$t_{a2}[A_2]$		$t_{a2}[A_{m-1}]$	$t_{a2}[A_m]$
t_{a3}	$t_{a3}[A_1]$	$t_{a3}[A_2]$		$t_{a3}[A_{m-1}]$	$t_{a3}[A_m]$
……	……	……	……	……	……

TABLE B					
Entities	**Attribute B_1**	**Attribute B_2**	**……**	**Attribute B_{j-1}**	**Attribute B_j**
t_{b1}	$t_{b1}[B_1]$	$t_{b1}[B_2]$		$t_{b1}[B_{j-1}]$	$t_{b1}[B_j]$
t_{b2}	$t_{b2}[B_1]$	$t_{b2}[B_2]$		$t_{b2}[B_{j-1}]$	$t_{b2}[B_j]$
t_{b3}	$t_{b3}[B_1]$	$t_{b3}[B_2]$		$t_{b3}[B_{j-1}]$	$t_{b3}[B_j]$
……	……	……	……	……	……

Table 5. 1: Pair of tables example for the Attribute Similarity Approach (1)

It is important to note that the attribute values of both tables A and B are subject to all the pre-processing steps, as explained in **4.3.** This implies that a single attribute value $\mathbf{t_{ai}}[A_k]$ = attribute value of A_k for the entity $t_{ai} \in$ set[A] is converted to text, cleaned and pre-processed.

Keeping that in mind, the next step is this: **For each entity t_{ai} of Table A and t_{bi} of Table B, we concatenate the context of the misaligned attributes to a single '*Merged*' attribute using simple string concatenation**. So, the new tables *A'* and *B'* look like this:

TABLE A'			
Entities	**Attribute A_1**	**Attribute A_2**	**Merged Attribute A**
t_{a1}	$t_{a1}[A_1]$	$t_{a1}[A_2]$	t_{a1}[Merged]
t_{a2}	$t_{a2}[A_1]$	$t_{a2}[A_2]$	t_{a2}[Merged]
t_{a3}	$t_{a3}[A_1]$	$t_{a3}[A_2]$	t_{a3}[Merged]

TABLE B'			
Entities	**Attribute B_1**	**Attribute B_2**	**Merged Attribute B**
t_{b1}	$t_{b1}[B_1]$	$t_{b1}[B_2]$	t_{b1}[Merged]
t_{b2}	$t_{b2}[B_1]$	$t_{b2}[B_2]$	t_{b2}[Merged]
t_{b3}	$t_{b3}[B_1]$	$t_{b3}[B_2]$	t_{b3}[Merged]

Table 5. 2:Pair of tables example for the Attribute Similarity Approach (2)

Now, keeping in mind that there is already information on the alignment of $A_1 \rightarrow B_1$ and $A_2 \rightarrow B_2$, we also assume the alignment: **Merged Attribute A → Merged Attribute B**

Here comes the exciting part: What are we left with after the aforementioned procedure? Technically, we are left with two tables A' and B' that are perfectly attribute-aligned!

In the case where we are also equipped with training, validation and test sets in the same format as described in sub-section **2.3** (A training dataset (S) with **n-rows** ,that contains pairs of ID's of tuples $\mathbf{t_1}$,$\mathbf{t_2} \in$ set[T] as well as a binary indicator of whether such a pair consists a match (1) or not (0)), one could directly proceed with implementing all the steps 1-5 exactly as described in **2.3**.

That is, after constructing the new tables A' and B', we are mimicking the DeepER framework: Given two candidate tuples ($\mathbf{t_{a'}}$,$\mathbf{t_{b'}}$) where $\mathbf{t_{a'}} \in$ set[A'] and $\mathbf{t_{b'}} \in$ set[B'], we perform a mapping to their similarity vector and train a classifier C, exactly as described in *Algorithm 2.2*.

For each candidate pair of tuples ($\mathbf{t_{a'}}$,$\mathbf{t_{b'}}$), the similarity vector is a 3-dimensional vector: the first two dimensions refer to the similarity of the aligned attribute values [A_1,B_1] and [A_2,B_2], while the last dimension refers to the similarity of the Merged attributes [Merged Attribute A, Merged Attribute B] (denoted from now on as [Merged[A], Merged[B]] for simplicity). This mapping is formally given by the following mathematical formula:

($t_{a'}$,$t_{b'}$) → **Sim**[($t_{a'}$,$t_{b'}$)]= [**Sim**($t_{a'}$[A_1], $t_{b'}$[B_1]), **Sim**($t_{a'}$[A_2], $t_{b'}$[B_2]), **Sim**($t_{a'}$[Merged[A]], $t_{b'}$[Merged[B]])]

The above example is assuming that only two pairs of attributes of the original schemas are aligned. Generalizing for more or less originally aligned attributes is trivial: Assuming k-originally aligned attributes between A and B, we construct two new tables A' and B' of k+1 attributes each and repeat the procedure as explained above. The similarity vector for a tuple-pair candidate will be a (k+1)-dimensional vector, fed to a classifier C (in our case, C will be a Neural Network classifier) using a training set, validated and tested upon a validation and test set respectively. It is also worth mentioning that the special case **k=0** implies total agnosticism for the two schemas..

B. Concatenated Strings Approach

Let us consider again two tables A and B, each one related to its own attribute schema: A has m-attributes and B has j-attributes (m, j ∈ R). Furthermore, we consider the respective attributes of each table: [A_1,A_2,….,A_m] = the set of attributes of Table A, while [B_1,B_2,….,B_j] = the set of attributes of Table B. However, this time we make no assumptions about the alignment of any attributes between table A and B. Let us also consider the sets of their entity tuples set[A], set[B]. Of course, for an entity $\mathbf{t_a}$ that is included in table A stands that: $\mathbf{t_a} \in$ set[A], while for an entity $\mathbf{t_b}$ of table B stands that: $\mathbf{t_b} \in$ set[B].

The aforementioned tables should look like this (*Table 5.3)*

TABLE A					
Entities	**Attribute A_1**	**Attribute A_2**	**……**	**Attribute A_{m-1}**	**Attribute A_m**
t_{a1}	$t_{a1}[A_1]$	$t_{a1}[A_2]$		$t_{a1}[A_{m-1}]$	$t_{a1}[A_m]$
t_{a2}	$t_{a2}[A_1]$	$t_{a2}[A_2]$		$t_{a2}[A_{m-1}]$	$t_{a2}[A_m]$
t_{a3}	$t_{a3}[A_1]$	$t_{a3}[A_2]$		$t_{a3}[A_{m-1}]$	$t_{a3}[A_m]$
……	……	……	……	……	……

TABLE B					
Entities	**Attribute B_1**	**Attribute B_2**	**……**	**Attribute B_{j-1}**	**Attribute B_j**
t_{b1}	$t_{b1}[B_1]$	$t_{b1}[B_2]$		$t_{b1}[B_{j-1}]$	$t_{b1}[B_j]$
t_{b2}	$t_{b2}[B_1]$	$t_{b2}[B_2]$		$t_{b2}[B_{j-1}]$	$t_{b2}[B_j]$
t_{b3}	$t_{b3}[B_1]$	$t_{b3}[B_2]$		$t_{b3}[B_{j-1}]$	$t_{b3}[B_j]$
……	……	……	……	……	……

Table 5. 3: Pair of tables example for the Concatenated Strings Approach (1)

Each attribute value of $\mathbf{t_{ai}}[A_k]$ of A is subject to text converting, as explained in **4.3**. The same stands for every attribute value $\mathbf{t_{bi}}[B_k]$ of B.

Let us now consider a single entity $\mathbf{t_{ai}} \in$ set[A]. This entity can be represented by its attribute values: $\mathbf{t_{ai}} = [\ t_{ai}[A_1]\ ,\ t_{ai}[A_2]\ ,\ \ldots\ldots,\ t_{ai}[A_m]\ ]$. Since each and every one of $\mathbf{t_{ai}}[A_k]$: $k \leq m$, is indeed in textual format, we can concatenate all these attribute values into a single string (**simple string concatenation**).

$$\mathbf{t_{ai}} = [\ t_{ai}[A_1]\ ,\ t_{ai}[A_2]\ ,\ \ldots\ldots,\ t_{ai}[A_m]\ ] \rightarrow [\ t_{ai}[A_1] + t_{ai}[A_2] + \ldots\ldots + t_{ai}[A_m]\ ]$$

The new concatenated string is now considered as the new representation of the tuple entity $\mathbf{t_{ai}}$.

Let us also perform the same mapping to an entity $\mathbf{t_{bi}} \in$ set[B]:

$$\mathbf{t_{bi}} = [\ t_{bi}[B_1]\ ,\ t_{bi}[B_2]\ ,\ \ldots\ldots,\ t_{bi}[B_j]\ ] \rightarrow [\ t_{bi}[B_1] + t_{bi}[B_2] + \ldots\ldots + t_{bi}[B_j]\ ]$$

Applying the above mappings to all entities t_{ai} of A and t_{bi} of B, we have indirectly constructed two new tables A' and B', each one consisting of a single 'concatenated' column (*Table 5.4*).

To make it clear, we could think the tables A' and B' as column tables. This single column of each one of the new tables embraces the totality of the information that was originally available in tables A and B, respectively.

This move enables us to ignore the dataset schemas. The problem of matching entities with many attribute values is now transformed to the problem of matching text columns

TABLE A'	
Entities	Concatenated Attribute A (**Concat**[A])
$\mathbf{t_{a1}}$	$\mathbf{t_{a1}}[A_1] + \mathbf{t_{a1}}[A_2] + \ldots\ldots + \mathbf{t_{a1}}[A_m]$
$\mathbf{t_{a2}}$	$\mathbf{t_{a2}}[A_1] + \mathbf{t_{a2}}[A_2] + \ldots\ldots + \mathbf{t_{a2}}[A_m]$
$\mathbf{t_{a3}}$	$\mathbf{t_{a3}}[A_1] + \mathbf{t_{a3}}[A_2] + \ldots\ldots + \mathbf{t_{a3}}[A_m]$
……	……

TABLE B'	
Entities	Concatenated Attribute B (**Concat**[B])
$\mathbf{t_{b1}}$	$\mathbf{t_{b1}}[B_1] + \mathbf{t_{b1}}[B_2] + \ldots\ldots + \mathbf{t_{b1}}[B_j]$
$\mathbf{t_{b2}}$	$\mathbf{t_{b2}}[B_1] + \mathbf{t_{b2}}[B_2] + \ldots\ldots + \mathbf{t_{b2}}[B_j]$
$\mathbf{t_{b3}}$	$\mathbf{t_{b3}}[B_1] + \mathbf{t_{b3}}[B_2] + \ldots\ldots + \mathbf{t_{b3}}[B_j]$
……	……

Table 5. 4: Pair of tables example for the Concatenated Strings Approach (2)

Let us now consider a training set (S) consisting of pairs of candidate tuples (in the same format as described in **2.3: Step 1**).

For a candidate pair ($\mathbf{t_{ai}}$,$\mathbf{t_{bk}}$) included in (S), we can directly concatenate t_{ai}[Concat[A]] and t_{bk}[Concat[B]]*. Now, the candidate pair ($\mathbf{t_{ai}}$,$\mathbf{t_{bk}}$) is represented by a single text:

($\mathbf{t_{ai}}$,$\mathbf{t_{bk}}$) → $\mathbf{t_{ai}}$[Concat[A]] + $\mathbf{t_{bk}}$[Concat[B]] = Concat[A+B] = Concatenated String of $\mathbf{t_{ai}}$[Concat[A]] and $\mathbf{t_{bk}}$[Concat[B]]

The final step is to map the new string to its related word vector: We apply ***Algorithm 2.1*** and map Concat[A+B] to its vector representation (using Spacy, that is a 300-d numerical vector).

By repeating this procedure for all candidate pairs included in S, we are left with a training set of candidate tuple-pairs, where each pair is represented by its **word vector**. Since S also includes a matching indicator (**1** for match and **0** for no match), we can use the tuple-pair concatenated vectors along with their match indicator to directly train a Neural Network Classifier.

If one is also equipped with a validation and a test set (which is indeed the case for DBLP-Scholar and Beer Advocate-RateBeer datasets), it is easy to validate and test the trained classifier.

The approach is completely schema agnostic and does not include any similarity measurements.

5.2 Attribute Similarity Approach: 4/4 Aligned Attributes

The experimental results for the case of total attribute alignment, both for DBLP-Scholar and Beer Advocate-RateBeer dataset groups, are presented in this sub-section. Aligned attributes are similarly colored. The assumed 'not-aligned' columns are merged into a single **'merged'** column. The **id** column was only used to match **Table A** and **Table B** entities according to the corresponding id pairs of **Train**, **Valid** and **Test** tables.

A. DBLP-Scholar Dataset

- ***Table Schemas and Attribute Alignement***

Table A → [id, title, authors, venue, year]

Table B → [id, title, authors, venue, year]

4-d Similarity Vectors for each entity-pair

Similarity Metric : ***Cosine Similarity***

Train = [id_Table_A, id_Table_B, match_indicator (0 or 1)]

Valid = [id_Table_A, id_Table_B, match_indicator (0 or 1)]

Test = [id_Table_A, id_Table_B, match_indicator (0 or 1)]

- **Neural Network Architecture**

A simple Dense neural network classifier was trained using the entity pairs of tables A and B whose id pairs are included in the **Train** table (*Training Set*). The network's architecture consists of an input layer of 4 neurons (compatible to the 4-dim similarity vectors fed to the network), followed by 3 hidden layers of 250 neurons, 4 hidden layers of 512 neurons , 1 hidden layer of 1000 neurons as well as an output layer of 2 neurons (compatible with match-mismatch characterization). A visual presentation of the network's architecture (**Figure 5.1**) as well as its hyperparameter configurations are provided below:

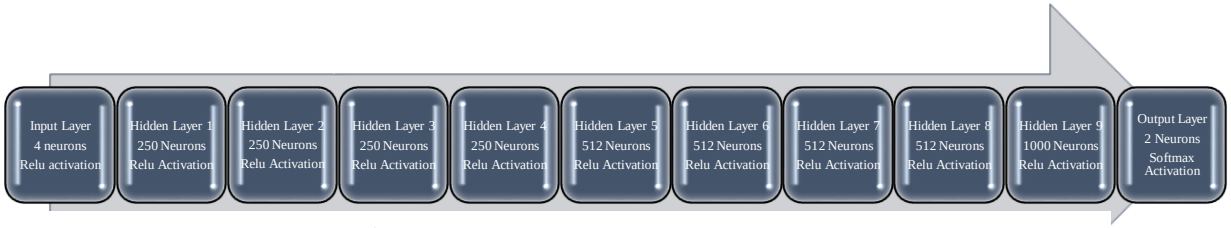

Figure 5. 1: Visual Representation of the trained Dense Model (4/4 Attribute Alignment DBLP-Scholar)

- **Training, Tuning and Learning Curves**

The architecture itself and its hyperparameters were tuned using entity pairs whose ids are included in the **Valid** table (*Validation Set*). We present the resulting tuned hyperparameter values in *Table 5.5*. After each epoch, the Network used its current weights to predict on the Validation set. Based on its accuracy, the values of the loss function directed the optimization procedure (typical **Back Propagation**). Based on this, we provide the reader with the following *Learning Curves* (*Figure 5.2*), depicting both Accuracy and Cross-Entropy Loss values during training.

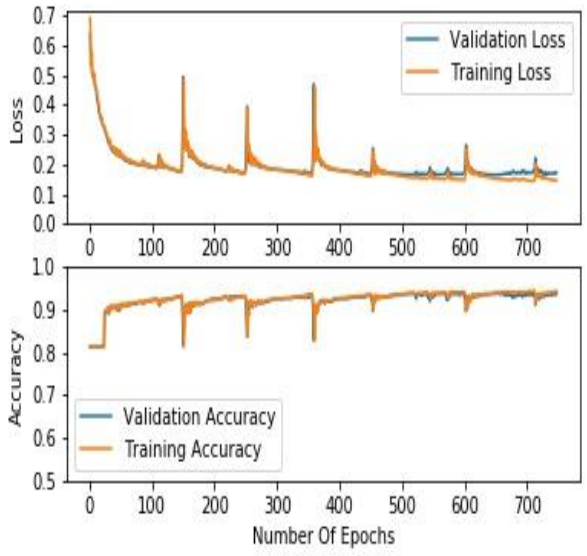

Figure 5. 2: Learning Curves during Training

Hyper-Parameter	*Value*
Optimizer	Adam
Initial Learning Rate	0.0001
No of Epochs	748
Batch Size	17223

Table 5.5: Network Hyper-Parameter Values

Predicting on Validation Set

After training, we used the classifier to predict on the Validation Set. The reader should keep in mind that Validation Set was used to tune the Network during training, so the results are only indicative. They cannot be trusted to infer the classifier's ability to generalize on unseen data. The results are provided below: A **Classification Report Matrix** is given in *Table 5.6* providing all the necessary insight on the results. *Table 5.7* is a **Confusion Matrix** and it depicts the entity-pairs assignment to classes of Non-Match (0) or Match (1):

Class	**Precision**	**Recall**	**F1-Score**	**Support**
0	0.95	0.97	0.96	4672
1	0.87	0.78	0.82	1070
Macro Avg	0.91	0.88	0.89	5742
Weighted Avg	0.94	0.94	0.94	5742
Total Accuracy	0.94			

Table 5. 6: Classification Report on Validation Set)

N total = 5742	Predicted Class 0	Predicted Class 1
Actuall Class 0	4551	121
Actual Class 1	237	833

Table 5. 7: Confusion Matrix on Validation Set

- **Predicting on Test Set**

The classifier's generalization ability, and therefore its prediction power, was tested with **Test** data. *Table 5.8* is a **Classification Report Matrix** on **Test** set. At the same time, the correspondent **Confusion Matrix** is given in *Table 5.9.* Finally, the AUC curve of the NN-Model is given in *Figure 5.3*:

Class	**Precision**	**Recall**	**F1-Score**	**Support**
0	0.95	0.97	0.96	4672
1	0.87	0.77	0.82	1070
Macro Avg	0.91	0.87	0.89	5742
Weighted Avg	0.94	0.94	0.94	5742
Total Accuracy	0.94	F1 Score	0.82	

Table 5. 8: Classification Report on Test Set

N total = 5742	Predicted Class 0	Predicted Class 1
Actuall Class 0	4553	119
Actual Class 1	244	826

Table 5. 9: Confusion Matrix on Test Set

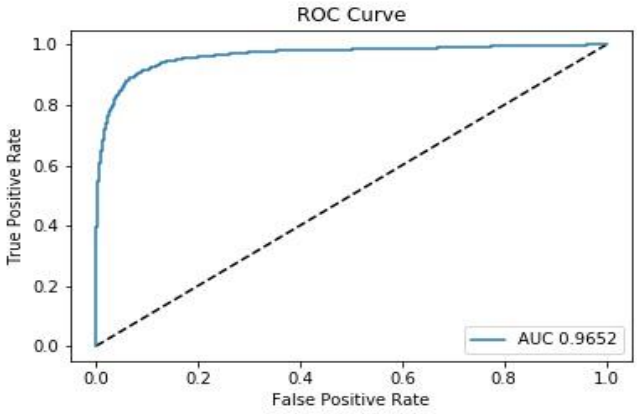

Figure 5.3: ROC Curve and Area Under Curve (AUC computation) using predictions on Test Set. The graph refers to the case of total alignment between Table A and Table B attributes

B. Beer Advocate-RateBeer Dataset

- ***Table Schemas and Attribute Alignement***

Table A → [id, Beer Name, Brew Factory Name, Style, ABV]

Table B → [id, Beer Name, Brew Factory Name, Style, ABV]

4-d Similarity Vectors for each entity-pair
Similarity Metric : ***Cosine Similarity***

Train = [id_Table_A, id_Table_B, match_indicator (0 or 1)]

Valid = [id_Table_A, id_Table_B, match_indicator (0 or 1)]

Test = [id_Table_A, id_Table_B, match_indicator (0 or 1)]

- **Neural Network Architecture**

A simple Dense neural network classifier was trained using the entity pairs of tables A and B whose id pairs are included in the **Train** table (*Training Set*). The network's architecture consists of an input layer of 4 neurons (compatible to the 4-dim similarity vectors fed to the network), followed by one hidden layer of 10 neurons plus **0.3 Dropout** as well as an output layer of 2 neurons (compatible with match-mismatch characterization). A visual presentation of the network's architecture (*Figure 5.4*) as well as its hyperparameter configurations are provided below:

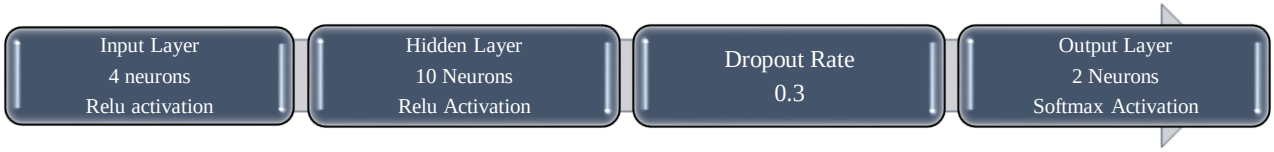

Figure 5. 4: Visual Representation of the trained Dense Model (4/4 Attribute Alignment Beer Advocate-RateBeer)

- **Training, Tuning and Learning Curves**

The architecture itself and its hyperparameters were tuned using entity pairs whose ids are included in the **Valid** table (*Validation Set*). We present the resulting tuned hyperparameter values in *Table 5.10*. After each epoch, the Network used its current weights to predict on the Validation set. Based on its accuracy, the values of the loss function directed the optimization procedure (typical **Back Propagation**). Based on this, we provide the reader with the following *Learning Curves* (*Figure 5.5*), depicting both Accuracy and Cross-Entropy Loss values during training.

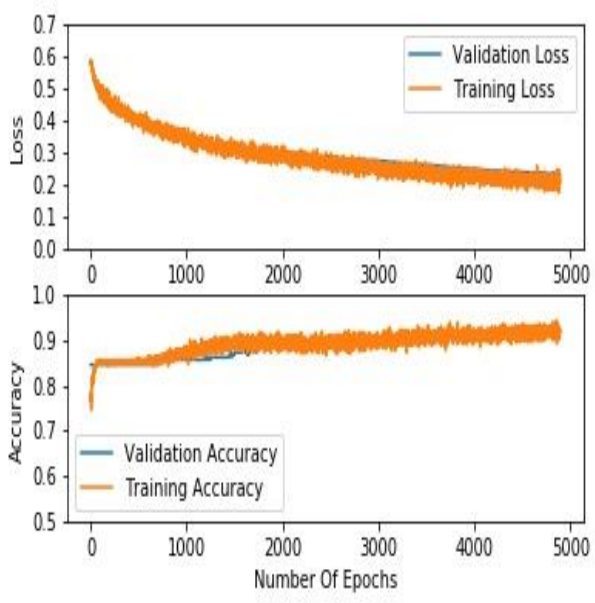

Figure 5.5: Learning Curves during Training

Hyper-Parameter	*Value*
Optimizer	Adam
Initial Learning Rate	0.0001
No of Epochs	4.892
Batch Size	268
Dropout Rate	0.3

Table 5.10 :Network Hyper-Parameter Values

- **Predicting on Validation Set**

After training, we used the classifier to predict on the Validation Set. A **Classification Report Matrix** is given in *Table 5.11*, providing all the necessary insight on the results. *Table 5.12* is a **Confusion Matrix** and it depicts the entity-pairs assignment to classes of Non-Match (0) or Match (1):

Class	Precision	Recall	F1-Score	Support
0	0.94	0.97	0.96	77
1	0.82	0.64	0.72	14
Macro Avg	0.88	0.81	0.84	91
Weighted Avg	0.92	0.92	0.92	91
Total Accuracy	0.92			

Table 5. 11: Classification Report on Validation Set

N total = 91	Predicted Class 0	Predicted Class 1
Actuall Class 0	75	2
Actual Class 1	5	9

Table 5. 12: Confusion Matrix on Validation Set

- **Predicting on Test Set**

The classifier's generalization ability, and therefore its prediction power, was tested with **Test** data *Table 5.13* is a **Classification Report Matrix** on **Test** set. At the same time, the correspondent **Confusion Matrix** is given in *Table 5.14*. Finally, the AUC curve of the NN-Model is given in *Figure 5.6*:

Class	Precision	Recall	F1-Score	Support
0	0.95	0.96	0.95	77
1	0.77	0.71	0.74	14
Macro Avg	0.86	0.84	0.85	91
Weighted Avg	0.92	0.92	0.92	91
Total Accuracy	0.92	F1-Score	0.74	

Table 5. 13: Classification Report on Test Set

N total = 91	Predicted Class 0	Predicted Class 1
Actuall Class 0	74	3
Actual Class 1	4	10

Table 5. 14: Confusion Matrix on Test Set

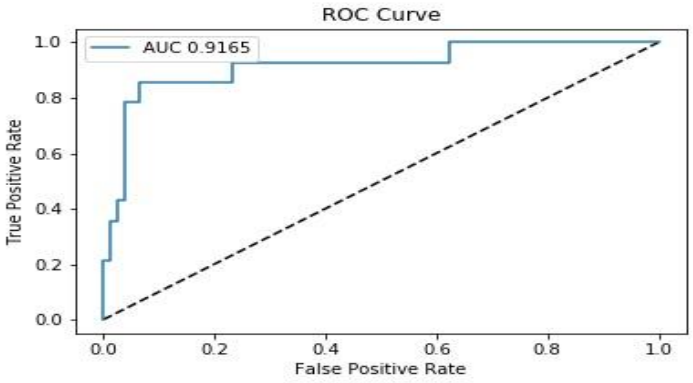

Figure 5.6: ROC Curve and Area Under Curve (AUC computation) using predictions on Test Set. The graph refers to the case of total alignment between Table A and Table B attributes

5.3 Attribute Similarity Approach: 2/4 Aligned Attributes

The experimental results for the case of 2/4 attribute alignment, both for DBLP-Scholar and Beer Advocate-RateBeer dataset groups, are presented in this sub-section. Aligned attributes are similarly colored. The assumed 'not-aligned' columns are merged into a single **'merged'** column. The **id** column was only used to match **Table A** and **Table B** entities according to the corresponding id pairs of **Train**, **Valid** and **Test** tables.

A. DBLP-Scholar Dataset

- ***Table Schemas and Attribute Alignement***

Table A → [id, title, authors, merged(venue+year)]

Table B → [id, title, authors, merged(venue+year)]

→ *3-d Similarity Vectors for each entity-pair*

Similarity Metric : ***Cosine Similarity***

Train = [id_Table_A, id_Table_B, match_indicator (0 or 1)]

Valid = [id_Table_A, id_Table_B, match_indicator (0 or 1)]

Test = [id_Table_A, id_Table_B, match_indicator (0 or 1)]

- **Neural Network Architecture**

A simple Dense neural network classifier was trained using the entity pairs of tables A and B whose id pairs are included in the **Train** table (*Training Set*). The network's architecture consists of an input layer of 3 neurons (compatible to the 3-dim similarity vectors fed to the network), followed by 4 hidden layers of 250 neurons, 4 hidden layers of 512 neurons , 1 hidden layer of 1000 neurons as well as an output layer of 2 neurons (compatible with match-mismatch

characterization). A visual presentation of the network's architecture (*Figure 5.7*) as well as its hyperparameter configurations is provided below:

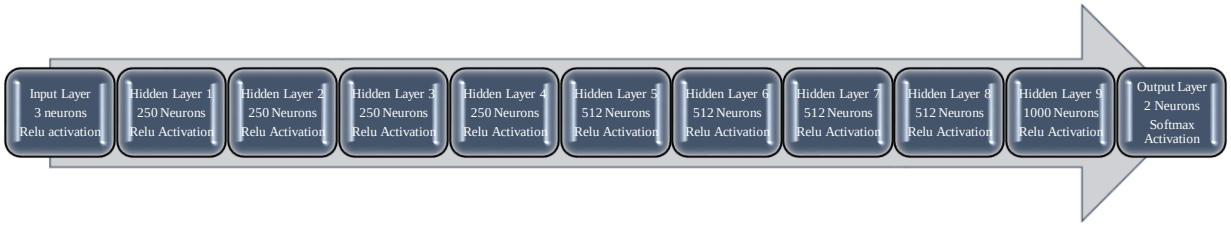

Figure 5. 7: Visual Representation of the trained Dense Model (2/4 Attribute Alignment DBLP-Scholar)

- **Training, Tuning and Learning Curves**

The architecture itself along with its hyperparameters was tuned using entity pairs whose ids are included in the **Valid** table (*Validation Set*). We present the resulting tuned hyperparameter values in *Table 5.15*. After each epoch, the Network used its current weights to predict on the Validation set. Based on its accuracy, the values of the loss function directed the optimization procedure (typical **Back Propagation**). Based on this, we provide the reader with the following *Learning Curves* (*Figure 5.8*), depicting both Accuracy and Cross-Entropy Loss values during training.

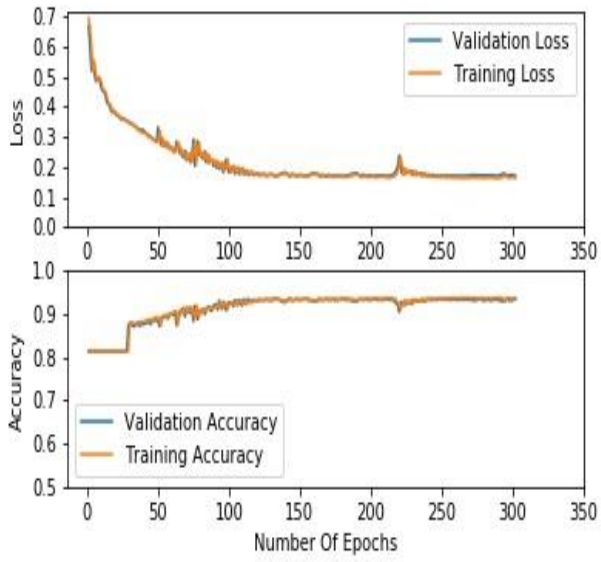

Figure 5.8: Learning Curves during Training

Hyper-Parameter	*Value*
Optimizer	Adam
Initial Learning Rate	0.0001
No of Epochs	303
Batch Size	17223

Table 5. 15 Network Hyper-Parameter Values

- **Predicting on Validation Set**

After training, we used the classifier to predict on the Validation Set. A **Classification Report Matrix** is given in *Table 5.16*. *Table 5.17* is a **Confusion Matrix** and it depicts the entity-pairs assignment to classes of Non-Match (0) or Match (1):

Class	**Precision**	**Recall**	**F1-Score**	**Support**
0	0.95	0.97	0.96	4672
1	0.86	0.77	0.82	1070
Macro Avg	0.91	0.87	0.89	5742
Weighted Avg	0.93	0.93	0.93	5742
Total Accuracy	0.93			

Table 5. 16: Classification Report on Validation Set

N total = 5742	Predicted Class 0	Predicted Class 1
Actuall Class 0	4539	133
Actual Class 1	241	829

Table 5. 17: Confusion Matrix on Validation Set

- **Predicting on Test Set**

The classifier's generalization ability, and therefore its prediction power, was tested with **Test** data. *Table 5.18* is a **Classification Report Matrix** on **Test** set predictions. At the same time, the correspondent **Confusion Matrix** is given in *Table 5.19*. Finally, the AUC curve of the NN-Model is given in *Figure 5.9*

Class	**Precision**	**Recall**	**F1-Score**	**Support**
0	0.95	0.98	0.96	4672
1	0.88	0.76	0.82	1070
Macro Avg	0.91	0.87	0.89	5742
Weighted Avg	0.93	0.94	0.93	5742
Total Accuracy	0.94	F1-Score	0.81	

Table 5. 18: Classification Report on Test Set

N total = 5742	Predicted Class 0	Predicted Class 1
Actuall Class 0	4558	114
Actual Class 1	255	815

Table 5. 19: Confusion Matrix on Test Set

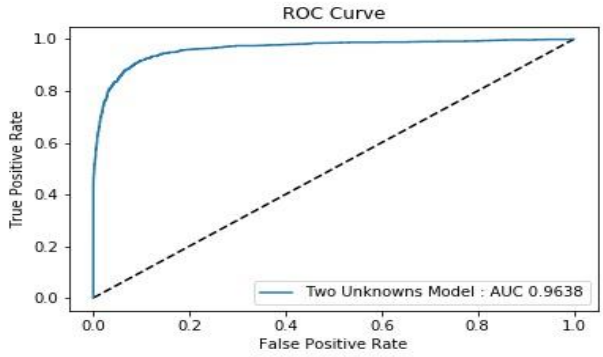

Figure 5.9 : ROC Curve and Area Under Curve (AUC computation) using predictions on Test Set. The graph refers to the case of 2 out of 4 matching attributes between Table A and Table B (2/4 attribute alignment)

B. Beer Advocate-RateBeer Dataset

- ***Table Schemas and Attribute Alignement***

Table A → [id, Beer Name, Brew Factory Name, merged(Style+ABV)]

Table B → [id, Beer Name, Brew Factory Name, merged(Style+ABV)]

3-d Similarity Vectors for each entity-pair
Similarity Metric : ***Cosine Similarity***

Train = [id_Table_A, id_Table_B, match_indicator (0 or 1)]

Valid = [id_Table_A, id_Table_B, match_indicator (0 or 1)]

Test = [id_Table_A, id_Table_B, match_indicator (0 or 1)]

- **Neural Network Architecture**

A simple Dense neural network classifier was trained using the entity pairs of tables A and B whose id pairs are included in the **Train** table (*Training Set*). The network's architecture consists of an input layer of 3 neurons (compatible to the 3-dim similarity vectors fed to the network), followed by one hidden layer of 10 neurons plus **0.3 Dropout** as well as an output layer of 2 neurons (compatible with match-mismatch characterization). A visual presentation of the network's architecture (*Figure 5.10*) as well as its hyperparameter configurations are provided below:

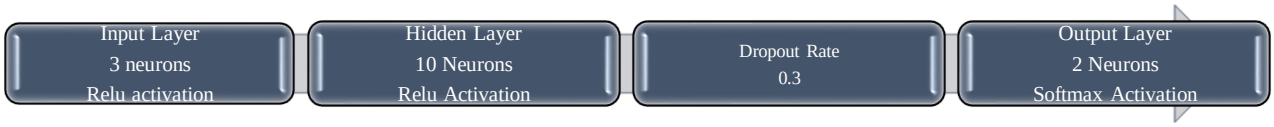

Figure 5. 10: Visual Representation of the trained Dense Model (2/4 Attribute Alignment Beer Advocate-RateBeer)

- **Training, Tuning and Learning Curves**

The architecture itself and its hyperparameters were tuned using entity pairs whose ids are included in the **Valid** table (*Validation Set*). We present the resulting tuned hyperparameter values in *Table 5.20*. After each epoch, the Network used its current weights to predict on the Validation set. Based on its accuracy, the values of the loss function directed the optimization procedure (typical **Back Propagation**). Based on this, we provide the reader with the following *Learning Curves* (*Figure 5.11*), depicting both Accuracy and Cross-Entropy Loss values during training.

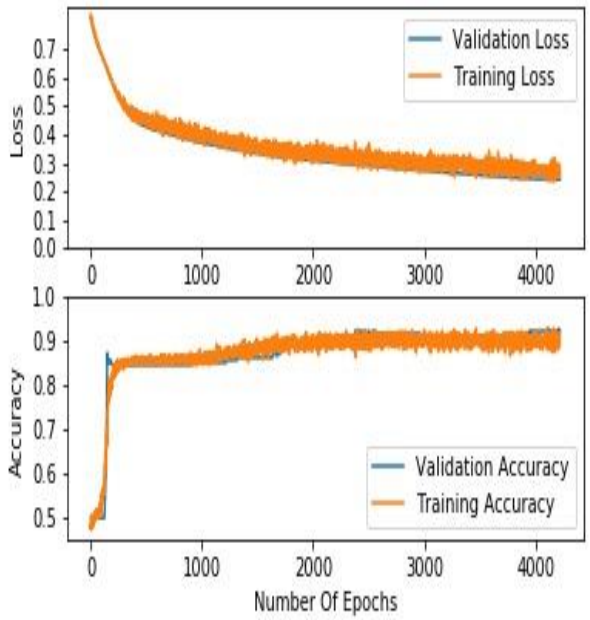

Figure 5.11: Learning Curves during Training

Hyper-Parameter	*Value*
Optimizer	Adam
Initial Learning Rate	0.0001
No of Epochs	4.221
Batch Size	268
Dropout Rate	0.3

Table 5. 20: Network Hyper-Parameter Values

- **Predicting on Validation Set**

After training, we used the classifier to predict on the Validation Set. A **Classification Report Matrix** is given in *Table 5.21*, providing all the necessary insight on the results. *Table 5.22* is a **Confusion Matrix** and it depicts the entity-pairs assignment to classes of Non-Match (0) or Match (1):

Class	**Precision**	**Recall**	**F1-Score**	**Support**
0	0.93	0.99	0.96	77
1	0.89	0.57	0.70	14
Macro Avg	0.91	0.78	0.83	91
Weighted Avg	0.92	0.92	0.92	91
Total Accuracy	0.92			

Table 5.21: Classification Report on Validation Set

N total = 91	Predicted Class 0	Predicted Class 1
Actuall Class 0	76	1
Actual Class 1	6	8

Table 5. 22: Confusion Matrix on Validation Set

- **Predicting on Test Set**

The classifier's generalization ability, and therefore its prediction power, was tested with **Test** data *Table 5.23* is a **Classification Report Matrix** on **Test** set. At the same time, the correspondent **Confusion Matrix** is given in *Table 5.24*. Finally, the AUC curve of the NN-Model is given in *Figure 5.12*:

Class	**Precision**	**Recall**	**F1-Score**	**Support**
0	0.94	0.96	0.95	77
1	0.75	0.64	0.69	14
Macro Avg	0.84	0.80	0.82	91
Weighted Avg	0.91	0.91	0.91	91
Total Accuracy	0.91	F1-Score	0.69	

Table 5. 23: Classification Report on Test Set

N total = 91	Predicted Class 0	Predicted Class 1
Actuall Class 0	74	3
Actual Class 1	5	9

Table 5. 24: Confusion Matrix on Test Set

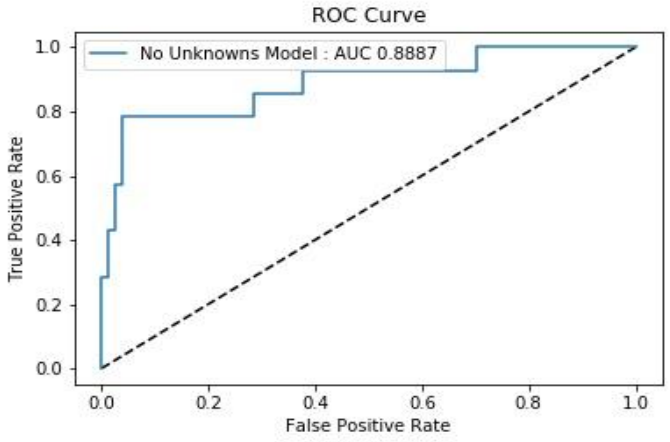

Figure 5.12: ROC Curve and Area Under Curve (AUC computation) using predictions on Test Set. The graph refers to the case of 2 out of 4 matching attributes between Table A and Table B (2/4 attribute alignment)

5.4 Attribute Similarity Approach: 1/4 Aligned Attributes

The experimental results for the case of 1/4 attribute alignment, both for DBLP-Scholar and Beer Advocate-RateBeer dataset groups, are presented in this sub-section. Aligned attributes are similarly colored. The assumed 'not-aligned' columns are merged into a single **'merged'** column. The **id** column was only used to match **Table A** and **Table B** entities according to the corresponding id pairs of **Train**, **Valid** and **Test** tables.

A. DBLP-Scholar Dataset

- ***Table Schemas and Attribute Alignement***

Table A → [id, title, merged(authors+venue+year)]

Table B → [id, title, merged(authors+venue+year)]

→ *2-d Similarity Vectors for each entity-pair*

Similarity Metric : ***Cosine Similarity***

Train = [id_Table_A, id_Table_B, match_indicator (0 or 1)]

Valid = [id_Table_A, id_Table_B, match_indicator (0 or 1)]

Test = [id_Table_A, id_Table_B, match_indicator (0 or 1)]

- **Neural Network Architecture**

A simple Dense neural network classifier was trained using the entity pairs of tables A and B whose id pairs are included in the **Train** table (*Training Set*). The network's architecture consists of an input layer of 2 neurons (compatible to the 2-dim similarity vectors fed to the network), followed by 4 hidden layers of 250 neurons, 4 hidden layers of 512 neurons , 1 hidden layer of 1000 neurons as well as an output layer of 2 neurons (compatible with match-mismatch

characterization). A visual presentation of the network's architecture (*Figure 5.13*) as well as its hyperparameter configurations is provided below:

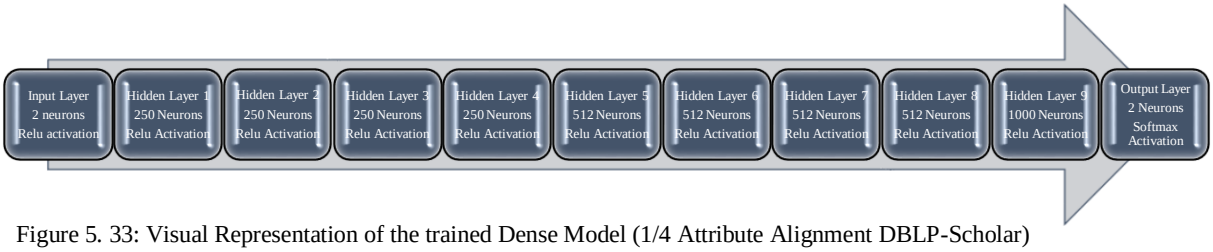

Figure 5. 33: Visual Representation of the trained Dense Model (1/4 Attribute Alignment DBLP-Scholar)

- **Training, Tuning and Learning Curves**

The architecture itself along with its hyperparameters was tuned using entity pairs whose ids are included in the **Valid** table (*Validation Set*). We present the resulting tuned hyperparameter values in *Table 5.25*. After each epoch, the Network used its current weights to predict on the Validation set. Based on its accuracy, the values of the loss function directed the optimization procedure (typical **Back Propagation**). Based on this, we provide the reader with the following *Learning Curves* (*Figure 5.14*), depicting both Accuracy and Cross-Entropy Loss values during training.

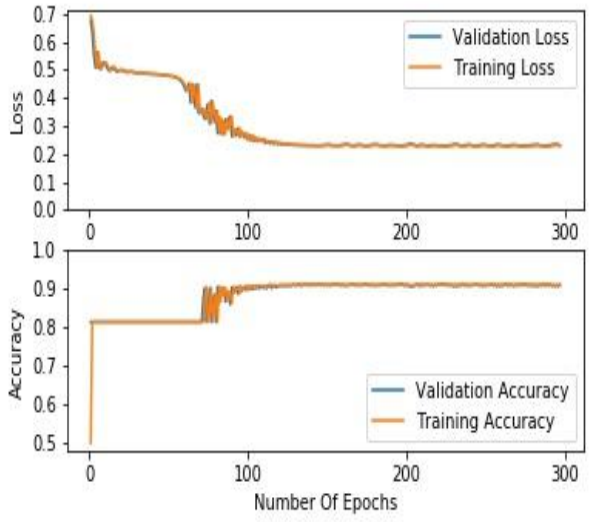

Figure 5.14: Learning Curves during Training

Hyper-Parameter	*Value*
Optimizer	Adam
Initial Learning Rate	0.0001
No of Epochs	298
Batch Size	17223

Table 5. 25: Network Hyper-Parameter Value

- **Predicting on Validation Set**

After training, we used the classifier to predict on the Validation Set. A **Classification Report Matrix** is given in *Table 5.26*. *Table 5.27* is a **Confusion Matrix** and it depicts the entity-pairs assignment to classes of Non-Match (0) or Match (1):

Class	**Precision**	**Recall**	**F1-Score**	**Support**
0	0.94	0.96	0.95	4672
1	0.79	0.73	0.75	1070
Macro Avg	0.86	0.84	0.85	5742
Weighted Avg	0.91	0.91	0.91	5742
Total Accuracy	0.91			

Table 5.26: Classification Report on Validation Set

N total = 5742	Predicted Class 0	Predicted Class 1
Actuall Class 0	4462	210
Actual Class 1	294	776

Table 5.27: Confusion Matrix on Validation Set

- **Predicting on Test Set**

The classifier's generalization ability, and therefore its prediction power, was tested with **Test** data. *Table 5.28* is a **Classification Report Matrix** on **Test** set predictions. At the same time, the correspondent **Confusion Matrix** is given in *Table 5.29*. Finally, the AUC curve of the NN-Model is given in *Figure 5.15*

Class	**Precision**	**Recall**	**F1-Score**	**Support**
0	0.93	0.95	0.94	4672
1	0.78	0.70	0.74	1070
Macro Avg	0.85	0.83	0.84	5742
Weighted Avg	0.90	0.91	0.90	5742
Total Accuracy	0.91	F1-Score	0.73	

Table 5.28: Classification Report on Test Set

N total = 5742	Predicted Class 0	Predicted Class 1
Actuall Class 0	4458	214
Actual Class 1	323	747

Table 5.29: Confusion Matrix on Test Set

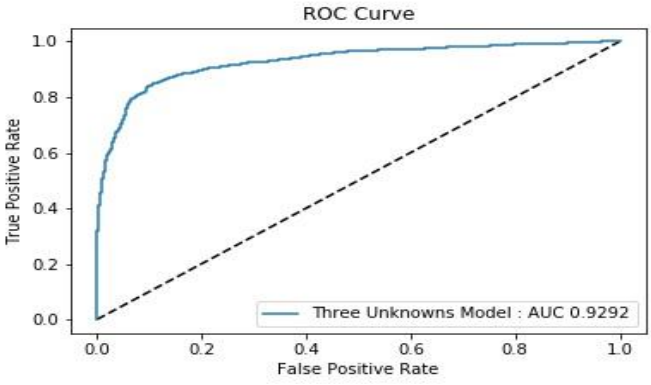

Figure 5.15: ROC Curve and Area Under Curve (AUC computation) using predictions on Test Set. The graph refers to the case of 1 out of 4 matching attributes between Table A and Table B (1/4 attribute alignment)

B. Beer Advocate-RateBeer Dataset

- ***Table Schemas and Attribute Alignement***

Table A → [id, Beer Name, merged(Brew Factory Name+Style+ABV)]

Table B → [id, Beer Name, merged(Brew Factory Name+Style+ABV)]

2-d Similarity Vectors for each entity-pair
Similarity Metric :
Cosine Similarity

Train = [id_Table_A, id_Table_B, match_indicator (0 or 1)]

Valid = [id_Table_A, id_Table_B, match_indicator (0 or 1)]

Test = [id_Table_A, id_Table_B, match_indicator (0 or 1)]

- **Neural Network Architecture**

A simple Dense neural network classifier was trained using the entity pairs of tables A and B whose id pairs are included in the **Train** table (*Training Set*). The network's architecture consists of an input layer of 2 neurons (compatible to the 2-dim similarity vectors fed to the network), followed by one hidden layer of 10 neurons plus **0.3 Dropout** as well as an output layer of 2 neurons (compatible with match-mismatch characterization). A visual presentation of the network's architecture (*Figure 5.16*) as well as its hyperparameter configurations are provided below:

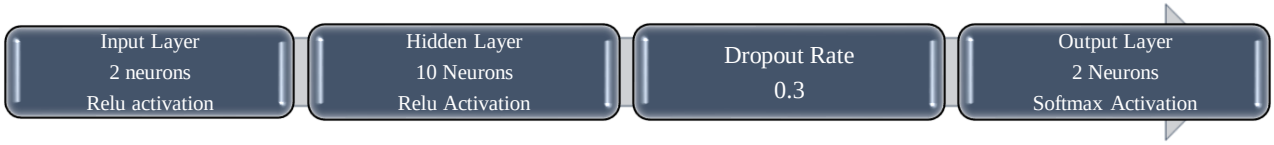

Figure 5. 46: Visual Representation of the trained Dense Model (1/4 Attribute Alignment Beer Advocate-RateBeer)

- **Training, Tuning and Learning Curves**

The architecture itself and its hyperparameters were tuned using entity pairs whose ids are included in the **Valid** table (*Validation Set*). We present the resulting tuned hyperparameter values in *Table 5.30*. After each epoch, the Network used its current weights to predict on the Validation set. Based on its accuracy, the values of the loss function directed the optimization procedure (typical **Back Propagation**). Based on this, we provide the reader with the following *Learning Curves* (*Figure 5.17*), depicting both Accuracy and Cross-Entropy Loss values during training.

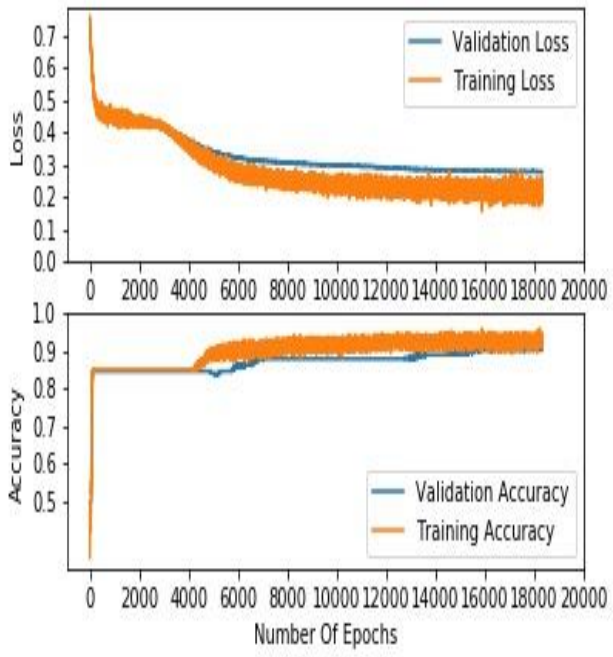

Figure 5.17: Learning Curves during Training

Hyper-Parameter	*Value*
Optimizer	Adam
Initial Learning Rate	0.001
No of Epochs	18.330
Batch Size	268
Dropout Rate	0.3

Table 5.30: Network Hyper-Parameter Values

- **Predicting on Validation Set**

After training, we used the classifier to predict on the Validation Set. A **Classification Report Matrix** is given in *Table 5.31*, providing all the necessary insight on the results. *Table 5.32* is a **Confusion Matrix** and it depicts the entity-pairs assignment to classes of Non-Match (0) or Match (1):

Class	Precision	Recall	F1-Score	Support
0	0.93	0.97	0.95	77
1	0.80	0.57	0.67	14
Macro Avg	0.86	0.77	0.81	91
Weighted Avg	0.91	0.91	0.91	91
Total Accuracy	0.91			

Table 5.31**:** Classification Report on Validation Set

N total = 91	Predicted Class 0	Predicted Class 1
Actuall Class 0	75	2
Actual Class 1	6	8

Table 5.32**:** Confusion Matrix on Validation Set

- **Predicting on Test Set**

The classifier's generalization ability, and therefore its prediction power, was tested with **Test** data *Table 5.33* is a **Classification Report Matrix** on **Test** set. At the same time, the correspondent **Confusion Matrix** is given in *Table 5.34*. Finally, the AUC curve of the NN-Model is given in *Figure 5.18*:

Class	Precision	Recall	F1-Score	Support
0	0.93	0.97	0.95	77
1	0.80	0.57	0.67	14
Macro Avg	0.86	0.77	0.81	91
Weighted Avg	0.91	0.91	0.91	91
Total Accuracy	0.91	F1-Score	0.66	

Table 5.33: Classification Report on Test Set

N total = 91	Predicted Class 0	Predicted Class 1
Actuall Class 0	75	2
Actual Class 1	6	8

Table 5.34: Confusion Matrix on Test Set

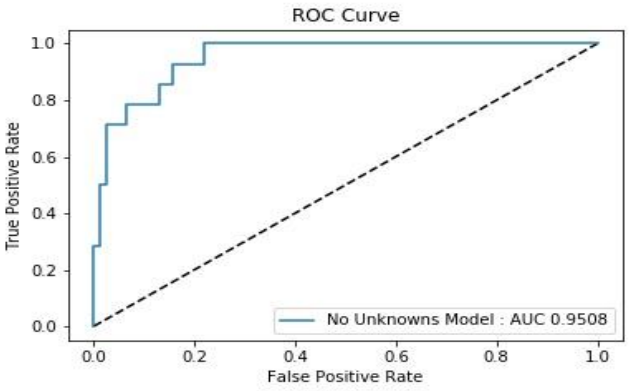

Figure 5.18: ROC Curve and Area Under Curve (AUC computation) using predictions on Test Set. The graph refers to the case of 1 out of 4 matching attributes between Table A and Table B (1/4 attribute alignment)

5.5 Attribute Similarity Approach: No Aligned Attributes

The experimental results for the case of no attribute alignment, both for DBLP-Scholar and Beer Advocate-RateBeer dataset groups, are presented in this sub-section. The assumed 'not-aligned' columns are merged into a single **'merged'** column. The **id** column was only used to match **Table A** and **Table B** entities according to the corresponding id pairs of **Train**, **Valid** and **Test** tables.

A. DBLP-Scholar Dataset

- ***Table Schemas and Attribute Alignement***

Table A → [id, merged(title+authors+venue+year)]

Table B → [id, merged(title+authors+venue+year)]

→ *1-d Similarity Vectors for each entity-pair*

Similarity Metric : ***Cosine Similarity***

Train = [id_Table_A, id_Table_B, match_indicator (0 or 1)]

Valid = [id_Table_A, id_Table_B, match_indicator (0 or 1)]

Test = [id_Table_A, id_Table_B, match_indicator (0 or 1)]

- **Neural Network Architecture**

A simple Dense neural network classifier was trained using the entity pairs of tables A and B whose id pairs are included in the **Train** table (*Training Set*). The network's architecture consists of an input layer of 1 neuron (compatible to the 1-dim similarity vectors fed to the network), followed by 4 hidden layers of 250 neurons, 4 hidden layers of 512 neurons , 1 hidden layer of 1000 neurons as well as an output layer of 2 neurons (compatible with match-mismatch

characterization). A visual presentation of the network's architecture (*Figure 5.19*) as well as its hyperparameter configurations is provided below:

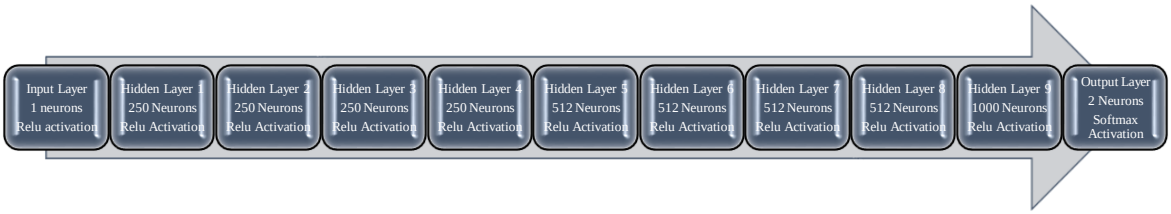

Figure 5. 19: Visual Representation of the trained Dense Model (0/4 Attribute Alignment DBLP-Scholar)

- **Training, Tuning and Learning Curves**

The architecture itself along with its hyperparameters was tuned using entity pairs whose ids are included in the **Valid** table (*Validation Set*). We present the resulting tuned hyperparameter values in *Table 5.35.* After each epoch, the Network used its current weights to predict on the Validation set. Based on its accuracy, the values of the loss function directed the optimization procedure (typical **Back Propagation**). Based on this, we provide the reader with the following *Learning Curves* (*Figure 5.20*), depicting both Accuracy and Cross-Entropy Loss values during training.

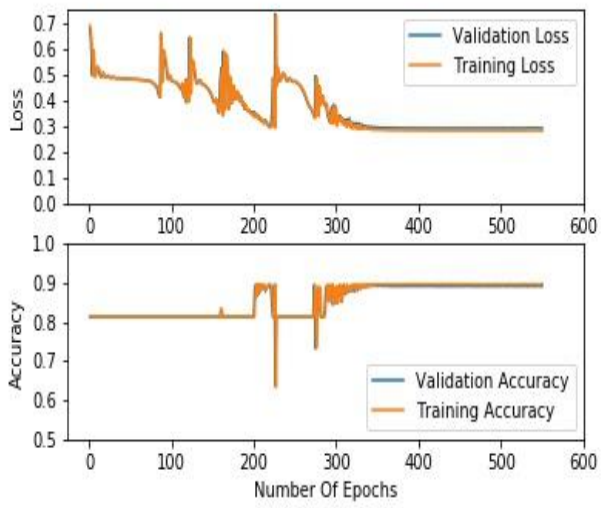

Figure 5.20: Learning Curves during Training

Hyper-Parameter	*Value*
Optimizer	Adam
Initial Learning Rate	0.0001
No of Epochs	550
Batch Size	17223

Table 5.35: Network Hyper-Parameter Value

- **Predicting on Validation Set**

After training, we used the classifier to predict on the Validation Set. A **Classification Report Matrix** is given in *Table 5.36*. *Table 5.37* is a **Confusion Matrix** and it depicts the entity-pairs assignment to classes of Non-Match (0) or Match (1):

Class	**Precision**	**Recall**	**F1-Score**	**Support**
0	0.90	0.98	0.94	4672
1	0.87	0.51	0.64	1070
Macro Avg	0.88	0.75	0.79	5742
Weighted Avg	0.89	0.89	0.88	5742
Total Accuracy	0.89			

Table 5.36: Classification Report on Validation Set

N total = 5742	Predicted Class 0	Predicted Class 1
Actuall Class 0	4592	80
Actual Class 1	524	546

Table 5.37: Confusion Matrix on Validation Set

- **Predicting on Test Set**

The classifier's generalization ability, and therefore its prediction power, was tested with **Test** data. *Table 5.38* is a **Classification Report Matrix** on **Test** set predictions. At the same time, the correspondent **Confusion Matrix** is given in *Table 5.39*. Finally, the AUC curve of the NN-Model is given in *Figure 5.21.*

Class	**Precision**	**Recall**	**F1-Score**	**Support**
0	0.90	0.99	0.94	4672
1	0.89	0.51	0.65	1070
Macro Avg	0.90	0.75	0.79	5742
Weighted Avg	0.90	0.90	0.89	5742
Total Accuracy	0.90	F1-Score	0.65	

Table 5.38: Classification Report on Test Set

N total = 5742	Predicted Class 0	Predicted Class 1
Actuall Class 0	4606	66
Actual Class 1	523	547

Table 5.39: Confusion Matrix on Test Set

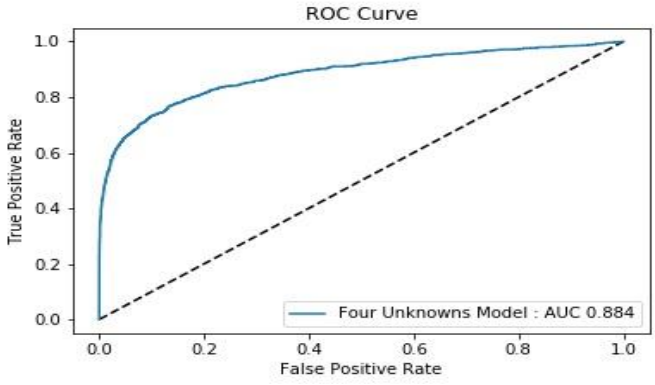

Figure 5.21: ROC Curve and Area Under Curve (AUC computation) using predictions on Test Set. The graph refers to the case of no matching attributes between Table A and Table B

B. Beer Advocate-RateBeer Dataset

- ***Table Schemas and Attribute Alignement***

Table A → [id, merged(Beer Name+ Brew Factory Name+Style+ABV)]

Table B → [id, merged(Beer Name+Brew Factory Name+Style+ABV)]

1-d Similarity Vectors for each entity-pair
Similarity Metric : ***Cosine Similarity***

Train = [id_Table_A, id_Table_B, match_indicator (0 or 1)]

Valid = [id_Table_A, id_Table_B, match_indicator (0 or 1)]

Test = [id_Table_A, id_Table_B, match_indicator (0 or 1)]

- **Neural Network Architecture**

A simple Dense neural network classifier was trained using the entity pairs of tables A and B whose id pairs are included in the **Train** table (*Training Set*). The network's architecture consists of an input layer of 1 neuron (compatible to the 1-dim similarity vectors fed to the network), followed by one hidden layer of 10 neurons plus **0.3 Dropout** as well as an output layer of 2 neurons (compatible with match-mismatch characterization). A visual presentation of the network's architecture (*Figure 5.22*) as well as its hyperparameter configurations are provided below:

Figure 5. 22: Visual Representation of the trained Dense Model (0/4 Attribute Alignment Beer Advocate-RateBeer)

- **Training, Tuning and Learning Curves**

The architecture itself and its hyperparameters were tuned using entity pairs whose ids are included in the **Valid** table (*Validation Set*). We present the resulting tuned hyperparameter values in *Table 5.40*. After each epoch, the Network used its current weights to predict on the Validation set. Based on its accuracy, the values of the loss function directed the optimization procedure (typical **Back Propagation**). Based on this, we provide the reader with the following *Learning Curves* (*Figure 5.23*), depicting both Accuracy and Cross-Entropy Loss values during training.

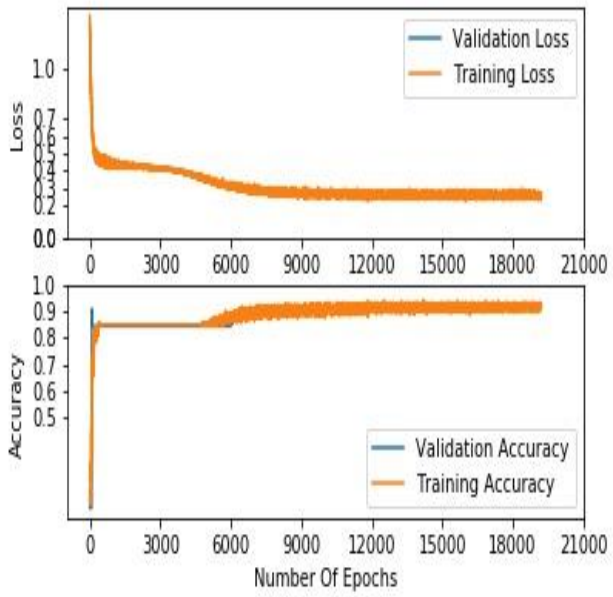

Figure 5.23: Learning Curves during Training

Hyper-Parameter	*Value*
Optimizer	Adam
Initial Learning Rate	0.0001
No of Epochs	19.200
Batch Size	268
Dropout Rate	0.3

Table 5.40: Network Hyper-Parameter Values

- **Predicting on Validation Set**

After training, we used the classifier to predict on the Validation Set. A **Classification Report Matrix** is given in *Table 5.41*, providing all the necessary insight on the results. *Table 5.42* is a **Confusion Matrix** and it depicts the entity-pairs assignment to classes of Non-Match (0) or Match (1):

Class	Precision	Recall	F1-Score	Support
0	0.92	0.99	0.95	77
1	0.88	0.50	0.64	14
Macro Avg	0.90	0.74	0.79	91
Weighted Avg	0.91	0.91	0.90	91
Total Accuracy	0.91			

Table 5.41**:** Classification Report on Validation Set

N total = 91	Predicted Class 0	Predicted Class 1
Actuall Class 0	76	1
Actual Class 1	7	7

Table 5.42: Confusion Matrix on Validation Set

- **Predicting on Test Set**

The classifier's generalization ability, and therefore its prediction power, was tested with **Test** data *Table 5.43* is a **Classification Report Matrix** on **Test** set. At the same time, the correspondent **Confusion Matrix** is given in *Table 5.44*. Finally, the AUC curve of the NN-Model is given in *Figure 5.24*:

Class	Precision	Recall	F1-Score	Support
0	0.95	0.96	0.95	77
1	0.77	0.71	0.74	14
Macro Avg	0.86	0.84	0.85	91
Weighted Avg	0.92	0.92	0.92	91
Total Accuracy	0.92	F1-Score	0.74	

Table 5.43: Classification Report on Test Set

N total = 91	Predicted Class 0	Predicted Class 1
Actuall Class 0	74	3
Actual Class 1	4	10

Table 5.44: Confusion Matrix on Test Set

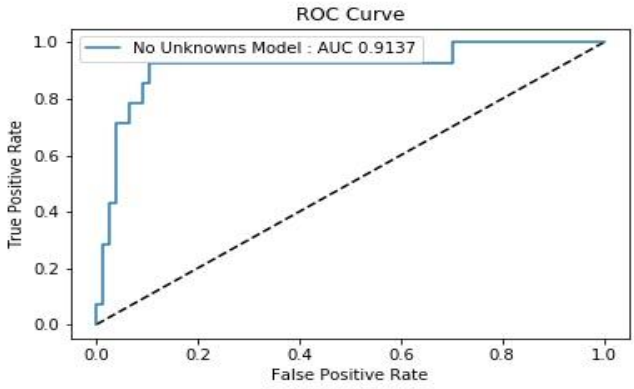

Figure 5.24: ROC Curve and Area Under Curve (AUC computation) using predictions on Test Set. The graph refers to the case of no matching attributes between Table A and Table B

5.6 Concatenated Strings Approach

The modeling results of the **Concatenated Strings** approach, as described in **5.1** are presented here. Before depicting the classification results, it would be convenient for the reader to consider the following:

a) **Data Augmentation** was applied to all matching entity-pairs included in the **Training Set** of the Beer Advocate-RateBeer Dataset. The experimentations showed that the best results were achieved when the actual matching pairs included in the training set were augmented x5. To be more accurate, instead of only 40 matching pairs from the total 268 training points, a new training set was created using the original 228 non-matching training entity pairs and 200 matching pairs (augmented by the original 40). As a result, a total amount of 468 training data points were used to train the classifier.

b) Instead of a Simple Dense Network, a **CNN Neural Network Architecture** was applied for the Beer Advocate-RateBeer Dataset.

A. DBLP-Scholar Dataset

- ***Table Schemas and String Concatenation***

The original schemas of Tables A and B are, of course, the same as before:

Table A → [id, title, authors, venue, year)] **Table B** → [id, title, authors, venue, year)]

For a candidate entity pair ($\mathbf{t_a}$,$\mathbf{t_b}$), we concatenate the merged text of all the attributes of $\mathbf{t_a}$ with the merged text of all the attributes of $\mathbf{t_b}$. The resulting concatenated single text represents the candidate pair and it is mapped to a **300-dimensional vector** by applying *Algorithm 2.1* to it. The procedure is explained in depth in **5.1**: String Concatenation Approach. The **Train**, **Valid** and **Test** tables are of the same schema as before.

- **Neural Network Architecture**

A simple Dense neural network classifier was trained using the entity pairs of tables A and B whose id pairs are included in the **Train** table (*Training Set*). The network's architecture consists of an input layer of 300 neurons (compatible to the 300-dim vectors fed to the network in this approach), followed by 1 hidden layer of 256 neurons, 1 hidden layers of 512 neurons , 1 hidden layer of 1024 neurons as well as an output layer of 2 neurons (compatible with match-mismatch characterization). Between each hidden layer**, Dropout** was used to reduce overfitting. A visual presentation of the network's architecture (*Figure 5.25*) as well as its hyperparameter configurations is provided below:

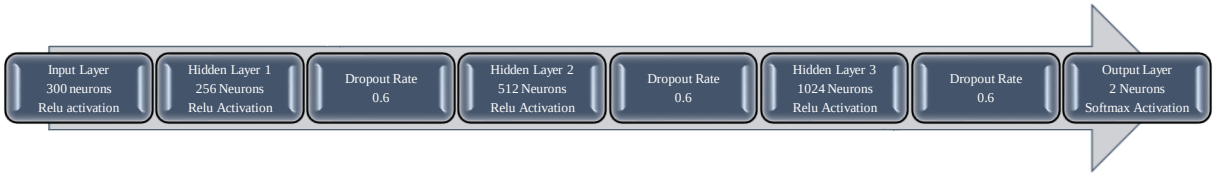

Figure 5. 25: Visual Representation of the trained Dense Model (Concatenated Strings DBLP-Scholar)

- **Training, Tuning and Learning Curves**

The architecture itself along with its hyperparameters was tuned using entity pairs whose ids are included in the **Valid** table (*Validation Set*). We present the resulting tuned hyperparameter values in *Table 5.45*. After each epoch, the Network used its current weights to predict on the Validation set. Based on its accuracy, the values of the loss function directed the optimization procedure (typical **Back Propagation**). Based on this, we provide the reader with the following *Learning Curves* (*Figure 5.26*), depicting both Accuracy and Cross-Entropy Loss values during training.

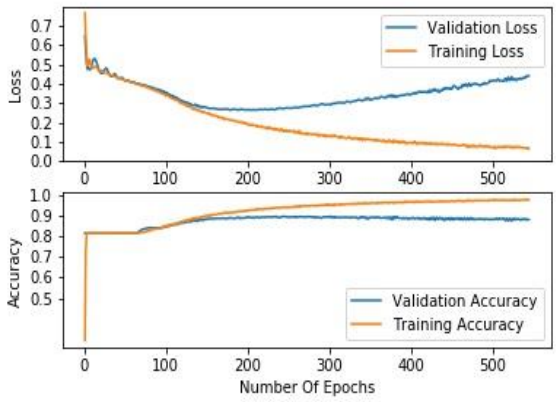

Figure 5.26: Learning Curves during Training

Hyper-Parameter	*Value*
Optimizer	Adam
Initial Learning Rate	0.0001
No of Epochs	545
Batch Size	17223

Table 5.45: Network Hyper-Parameter Value

- **Predicting on Validation Set**

After training, we used the classifier to predict on the Validation Set. A **Classification Report Matrix** is given in *Table 5.46*. *Table 5.47* is a **Confusion Matrix** and it depicts the entity-pairs assignment to classes of Non-Match (0) or Match (1):

Class	**Precision**	**Recall**	**F1-Score**	**Support**
0	0.94	0.93	0.94	4672
1	0.72	0.73	0.72	1070
Macro Avg	0.83	0.83	0.83	5742
Weighted Avg	0.90	0.90	0.90	5742
Total Accuracy	0.90			

Table 5.46: Classification Report on Validation Set

N total = 5742	Predicted Class 0	Predicted Class 1
Actuall Class 0	4366	306
Actual Class 1	293	777

Table 5.47: Confusion Matrix on Validation Set

- **Predicting on Test Set**

The classifier's generalization ability, and therefore its prediction power, was tested with **Test** data. *Table 5.48* is a **Classification Report Matrix** on **Test** set predictions. At the same time, the correspondent **Confusion Matrix** is given in *Table 5.49*. Finally, the AUC curve of the NN-Model is given in *Figure 5.27:*

Class	**Precision**	**Recall**	**F1-Score**	**Support**
0	0.93	0.93	0.93	4672
1	0.71	0.71	0.71	1070
Macro Avg	0.82	0.82	0.82	5742
Weighted Avg	0.89	0.89	0.89	5742
Total Accuracy	0.89	F1-Score	0.70	

Table 5.48: Classification Report on Test Set

N total = 5742	Predicted Class 0	Predicted Class 1
Actuall Class 0	4357	315
Actual Class 1	312	758

Table 5.49: Confusion Matrix on Test Set

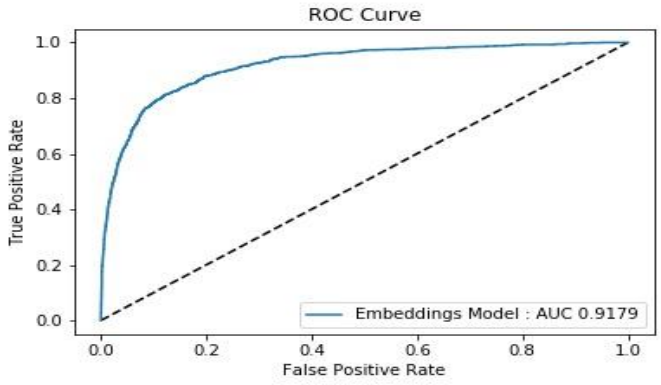

Figure 5.27: ROC Curve and Area Under Curve (AUC computation) using predictions on Test Set for the **Conatenated Strings** approach.

B. BeerAdvocate-RateBeer Dataset

- ***Table Schemas and String Concatenation***

The original schemas of Tables A and B are the same as before:

Table A → [id, Beer Name, Brew Factory Name, Style, ABV]

Table B → [id, Beer Name, Brew Factory Name, Style, ABV]

For a candidate entity pair ($\mathbf{t_a}$,$\mathbf{t_b}$), we concatenate the merged text of all the attributes of $\mathbf{t_a}$ with the merged text of all the attributes of $\mathbf{t_b}$. The resulting concatenated single text represents the candidate pair and it is mapped to a **300-dimensional vector** by applying Algorithm 2.2.2 to it.

Neural Network Architecture

A 1-dim Convolutional Neural Network (**CNN**)-based classifier was trained using the entity pairs of tables A and B whose id pairs are included in the **Train** table (*Training Set*). The network's architecture consists of an input layer of 300 neurons (compatible to the 300-dim vectors fed to the network in this approach), followed by a **Convolutional** layer with two filters, each equipped with a **12-dim sized kernel**. After the Convolutional layer, we enhance the model with a **MaxPooling** Layer of **4-dim pool size**. Between the MaxPooling and the Convolutional layers, we applied **Dropout** of rate equal to 0.4 in order to reduce overfitting. Finally, a Flattening Layer was stacked to the output of the MaxPooling Layer, which in turn leads to the output layer of 2 neurons (compatible with match-mismatch characterization). A visual presentation of the network's architecture (*Figure 5.28*) as well as its hyperparameter configurations is provided below:

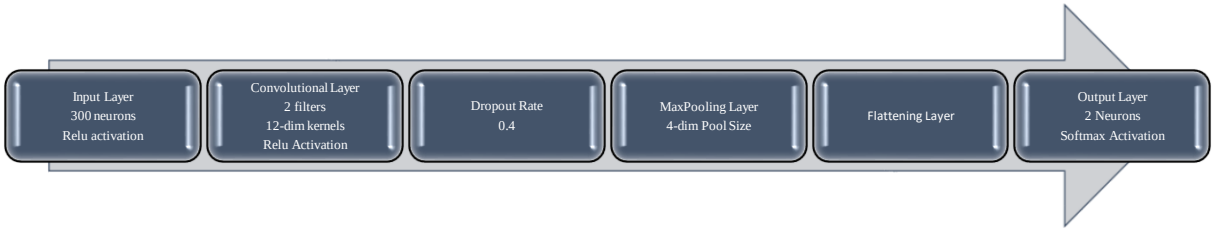

Figure 5. 28: Visual Representation of the trained Dense Model (Concatenated Strings Beer Advocate-RateBeer)

- **Training, Tuning and Learning Curves**

The architecture itself along with its hyperparameters was tuned using entity pairs whose ids are included in the **Valid** table (*Validation Set*). We present the resulting tuned hyperparameter values in *Table 5.50.* After each epoch, the Network used its current weights to predict on the Validation set. Based on its accuracy, the values of the loss function directed the optimization procedure (typical **Back Propagation**). Based on this, we provide the reader with the following *Learning Curves* (*Figure 5.29*), depicting both Accuracy and Cross-Entropy Loss values during training.

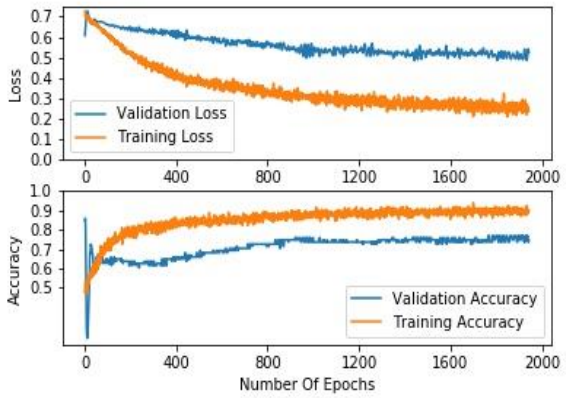

Figure 5.29: Learning Curves during Training

Hyper-Parameter	*Value*
Optimizer	Adam
Initial Learning Rate	0.01
No of Epochs	1944
Batch Size	468

Table 5.50: Network Hyper-Parameter Value

- **Predicting on Validation Set**

After training, we used the classifier to predict on the Validation Set. A **Classification Report Matrix** is given in *Table 5.51*. *Table 5.52* is a **Confusion Matrix** and it depicts the entity-pairs assignment to classes of Non-Match (0) or Match (1):

Class	**Precision**	**Recall**	**F1-Score**	**Support**
0	0.95	0.75	0.84	77
1	0.37	0.79	0.50	14
Macro Avg	0.66	0.77	0.67	91
Weighted Avg	0.86	0.76	0.79	91
Total Accuracy	0.76			

Table 5.51: Classification Report on Validation Set

N total = 91	Predicted Class 0	Predicted Class 1
Actuall Class 0	58	19
Actual Class 1	3	11

Table 5.52: Confusion Matrix on Validation Set

- **Predicting on Test Set**

The classifier's generalization ability, and therefore its prediction power, was tested with **Test** data. *Table 5.53* is a **Classification Report Matrix** on **Test** set predictions. At the same time, the correspondent **Confusion Matrix** is given in *Table 5.54*. Finally, the AUC curve of the NN-Model is given in *Figure 5.30:*

Class	**Precision**	**Recall**	**F1-Score**	**Support**
0	0.94	0.79	0.86	77
1	0.38	0.71	0.50	14
Macro Avg	0.66	0.75	0.68	91
Weighted Avg	0.85	0.78	0.80	91
Total Accuracy	0.78	F1-Score	0.5	

Table 5.53: Classification Report on Test Set

N total = 91	Predicted Class 0	Predicted Class 1
Actuall Class 0	61	16
Actual Class 1	4	10

Table 5.54: Confusion Matrix on Test Set

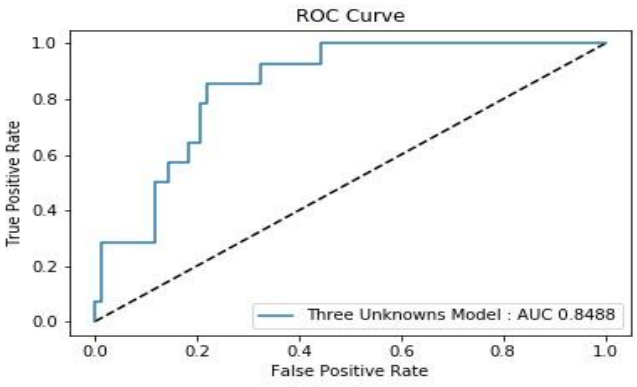

Figure 5.30: ROC Curve and Area Under Curve (AUC computation) using predictions on Test Set for the **Conatenated Strings** approach.

6. Applying LSH to the Deep Learning Framework

In the previous section, we experimented with the cases of partial or complete attribute misalignment between the two Tables whose entities we are trying to match. We have managed to show that the ER task can be solved efficiently in cases of partial or complete schema agnosticism, with the case of total alignment being, however, the most efficient in terms of classification performance.

We will not go into any further detail here (the modeling results are discussed in detail in **section 7**). However, it is important for us to remind the reader the duality of this thesis' purpose.

1. Test the ability of a Deep Learning framework in cases of partial or complete schema agnosticism. That is exactly what **section 5** is all about.

2. Test whether the process of matching id pairs between two given tables can be done in a distributed and more efficient way than checking on all possible pair combinations between the two tables. Towards this goal, we introduced the concepts of Locality Sensitive Hashing and LSH Forest, as well as the idea of implementing these techniques in the form of a document recommendation engine in order to significantly lower the amount of computations of the framework.

In this section, we will provide the reader with the established our established workflow: In sub-section 6.1, an in-depth exposure of how an LSH Recommendation Engine was combined with a pre-trained Deep Learning network to predict on DBLP-Scholar and Beer Advocate-RateBeer dataset will be given. In the same sub-section (6.1), we will also provide the reader with insight on how we worked in order to obtain some metrics on our framework's actual ability to solve the ER task. After that, we will provide our experimental results on both datasets.

6.1 Creating 'recommended-candidate pairs' with LSH forest and predicting with a Neural Network

Let us consider the case of having a dataset group similar to those that we are experimenting with (DBLP-Scholar, BeerAdvocate-RateBeer). To be more specific, we suppose that we have two **Tables A** and **B** along with a **Training**, a **Validation** and a **Test set**.

Table A and Table B include entities and along with their attribute values and a unique **id identifier** in the following form:

$\mathbf{t_A}$ = [id, Attribute_Value_1, Attribute_Value_2,……,Attribute_Value_k], $\mathbf{t_A} \in$ set[A] and k ∈ **R.**

$\mathbf{t_B}$ =[id,Attribute_Value_1,Attribute_Value_2,…..,Attribute_Value_j], $\mathbf{t_B} \in$ set[B] and j ∈ **R.**

On the other hand, the Training, Validation and Test sets consist of entities that include ids of entities $\mathbf{t_a} \in$ set[A], ids of entities $\mathbf{t_b} \in$ set[B] along with a matching indicator (match (1) or no-match (0)).

To make it more clear, an entity **e** that is included in any of the above sets (Training, Validation or Test) looks more or less like this:

e = [id_tableA,id_tableB, matching_indicator (0 or 1)]

In other words, this is exactly the same dataset group format that we have for DBLP-Scholar and Beer Advocate-RateBeer datasets that we used in **section 5**. The reader should be able by now to understand that it is trivial to train, validate and test a classifier C using the above dataset group. What is not trivial is this: Even though C is trained, validated and tested using the respective Training, Validation and Test sets, it is not able to solely achieve the ultimate goal of the ER task:

‘’ **Given two Tables A and B, find the matching entity pairs between them**’’

That is because the Training, Validation and Test sets only consist of some portion of possible candidate pairs between the two tables. In other words, there are many more entity-pairs that we need to check. As a matter of fact, we have already stated that the obvious and most complete way to do that is to check for all possible combinations after training C (exactly as described in Algorithm 2.2). It is, however, inefficient.

An alternative way to check for matching pairs between the two tables is the LSH Forest in the form of document recommendation, as we have already stated. Now, it is time to show the reader how this can be achieved.

Solving ER with LSH Forest: Explanation

Let the aforementioned dataset group: Tables A and B, a Training Set, a Validation Set and a Test Set. We can then train, validate and test a Classifier **C** under the procedure that is indicated by **Algorithm 2.2** (steps 1 to 7). The reader should feel comfortable on understanding this procedure by now. After this, we are left with a trained Classifier C that, given an input candidate pair, predicts whether the pair consists a matching pair or not.

The next move is this: We can go on and concatenate each entity's attributes **separately** for Tables A and B in a manner similar to what was described in the Concatenated Strings approach (Concatenated Strings Approach: sub-section 5.1). After doing so, we are left with two new tables A' and B'**.** For completeness, we depict the two tables:

TABLE A'	
Entities	Concatenated Attribute A (**Concat**[A])
$\mathbf{t_{a1}}$	$\mathbf{t_{a1}}[A_1] + \mathbf{t_{a1}}[A_2] + \ldots\ldots + \mathbf{t_{a1}}[A_m]$
$\mathbf{t_{a2}}$	$\mathbf{t_{a2}}[A_1] + \mathbf{t_{a2}}[A_2] + \ldots\ldots + \mathbf{t_{a2}}[A_m]$
$\mathbf{t_{a3}}$	$\mathbf{t_{a3}}[A_1] + \mathbf{t_{a3}}[A_2] + \ldots\ldots + \mathbf{t_{a3}}[A_m]$
……	……..

TABLE B'	
Entities	Concatenated Attribute B (**Concat**[B])
$\mathbf{t_{b1}}$	$\mathbf{t_{b1}}[B_1] + \mathbf{t_{b1}}[B_2] + \ldots\ldots + \mathbf{t_{b1}}[B_j]$
$\mathbf{t_{b2}}$	$\mathbf{t_{b2}}[B_1] + \mathbf{t_{b2}}[B_2] + \ldots\ldots + \mathbf{t_{b2}}[B_j]$
$\mathbf{t_{b3}}$	$\mathbf{t_{b3}}[B_1] + \mathbf{t_{b3}}[B_2] + \ldots\ldots + \mathbf{t_{b3}}[B_j]$
……	……

Table 6. 1: Pair of tables example for the Concatenated Strings Approach (Re-exlained)

In other words, for each entity $t_{ai} \in$ set[A] and for each entity $t_{bi} \in$ set[B], we apply the following mapping:

$$\mathbf{t_{ai}} = [\ t_{ai}[A_1]\ ,\ t_{ai}[A_2]\ ,\ \ldots\ldots,\ t_{ai}[A_m]\ ] \rightarrow [\ t_{ai}[A_1] + t_{ai}[A_2] + \ldots\ldots + t_{ai}[A_m]\ ]$$

$$\mathbf{t_{bi}} = [\ t_{bi}[B_1]\ ,\ t_{bi}[B_2]\ ,\ \ldots\ldots,\ t_{bi}[B_j]\ ] \rightarrow [\ t_{bi}[B_1] + t_{bi}[B_2] + \ldots\ldots + t_{bi}[B_j]\ ]$$

Each entity $\mathbf{t_{ai}}$ of Table A is mapped to a **text document entity** of the new table A'. The same stands for all entities $\mathbf{t_{bi}}$ of B. It is crucial to mention that each attribute value $\mathbf{t_{ai}}[A_k]$ of A is subject to text converting, as explained in **4.3**. The same stands for every attribute value $\mathbf{t_{bi}}[B_k]$ of B.

Now, we need to use C in order to come up with all the matching pairs between Tables A' and B'. However, we do not want to test for every combination of entity pairs. Here comes the tricky part:

The fact that each entity $\mathbf{t_{ai}}$ of Table A is mapped to a text document entity of the new table A' and each entity $\mathbf{t_{bi}}$ of Table B is mapped to a text document entity of the new table B' enables us to use **Algorithm 3.2 :** Considering a document entity $\mathbf{t_{ai}} \in$ set[A'] as a text query and set[B'] as a set of text documents, we can use LSH Forest exactly as described in Algorithm 3.2 in order to retrieve the n-nearest $t_{bi} \in$ set[B'] with respect to $\mathbf{t_{ai}} \in$ set[A'] for this particular entity $\mathbf{t_{ai}}$, with 'n' being a pre-defined user parameter.

Let us consider the above **n-nearest retrieved entities t_b(retrieved_i) ∈ set[B']** for a text query $\mathbf{t_{ai}}$. They form a new text set that directly relates this particular $\mathbf{t_{ai}}$ with its nearest neighbors on set[B']. We denote this set as $\mathbf{Ret_n[t_{ai}]}$, for which it stands that:

$Ret_n[t_{ai}] = \{ [t_b(retrieved_1), t_b(retrieved_2),.., t_b(retrieved_n)]: t_b(retrieved_k) \in set[B'] \; \forall \; k, n=$pre-defined parameter that indicates the number of retrieved documents$\}$

Our approach is this: Given an entity $t_{ai} \in$ set[A'] and its related set $Ret_n[t_{ai}]$, we use Classifier C to predict only for those $t_{bi} \in$ set[B'] $\cap$ $Ret_n[t_{ai}]$. In other words, for the entity t_{ai}, we use C to predict only on its n-nearest documents, as given by *Algorithm 3.2*.

It is straightforward to realize that, by repeating the above procedure for every entity $t_{ai} \in$ set[A'], we solve the problem of ER without testing for all pair combinations. All in all, **Algorithm 6.1 summarizes our approach for solving the ER task**:

Algorithm 6.1

1: Input: Table A, Table B, training set S

2: Output: Prediction of matching pairs between A and B

3: // Training

4: for each pair of tuples (t_a; t_b) in S, where $t_a \in$set[A] and $t_b \in$set[B], do:

5: Train a classifier C using either Attribute Similarity Approach or Concatenated Strings Approach and true labels.

6: // Predicting

7: for each entity t_a of A, use LSH Forest Recommendation and compute $Ret_n[t_a]$ on set[B].

8: For each entity t_a of A and its related set $Ret_n[t_a]$, use C to predict for match or no-match $\forall\ t_b \in Ret_n[t_b]$

Algorithm 6. 1: Combining DL with LSH

Using any of the approaches of section 5 and combining it with LSH Forest technique to tackle the ER problem. All in all, the algorithm provides an efficient Deep Learning Approach to solving the ER problem.

.

Let us suppose that Table A has m-rows and Table B has j-rows , as well as a pre-defined number of recommendations n for every row-entity of Table A, with n<<j:

Complexity of predicting matching pairs amongst all combinations = O(m*j)

Complexity of predicting matching pairs using 6.1 = O(m*n) + O(LSH Forest Recommendation Procedure)

Of course, taking into account that LSH is an extremely efficient algorithmic procedure and the fact that n<<j, we can safely conclude that:

Complexity of predicting matching pairs amongst all combinations >> Algorithm 6.1

Practical Implemenation in DBLP-Scholar and Beer Advocate-RateBeer: Explanation of the Framework's workflow and Testing Explanation

Both DBLP-Scholar and Beer Advocate-RateBeer datasets are of the same format as described above. It is straightforward to use Algorithm 6.1 on both of them. Before we provide the results, the reader should keep in mind some aspects:

1. We are free to choose any of the classifiers of section 5.1 as C. However, since we have already made experimentations on the differences between models that assume different number of aligned attributes, we will only choose one of those models to feed on 6.1. We chose the model that exhibited the best performance, which is the one that assumes complete attribute alignment. After all, what we care about here is to test whether LSH Recommendation Forest has the ability to predict matching pairs.

2. We can either proceed on applying 6.1 using t_{ai} and $Ret_n[t_{ai}]$ $\forall$ $t_{ai} \in$ set[A] or t_{bi} and $Ret_n[t_{bi}]$ $\forall$ $t_{bi} \in$ set[B]. For each dataset case, we will only experiment with the latter case: For every entity in Table B, we will get recommendations for this entity from Table A and predict using our trained classifier.

3. Evaluating the performance of the framework is difficult: Even if we apply 6.1 and get a set of matching pair of ids, we cannot automatically evaluate whether all of the predicted pairs are indeed true matches. That is because, we only have true labels for a limited number of pairs: Those included in the Training, Testing and Validation Sets. However, we can overcome this evaluation problem and come up with a metric that shows whether or not our framework detects matching pairs efficiently in the following way: We consider **all the true matching pairs** included in our Test Set along with the pair of id's included in the set. How many of those matching pairs? were really detected by the framework (True Positives)? How many of them were not detected (False Negatives)? **This evaluation metric actually enables us to answer the following question: Given a set of known (true) matches amongst two tables A and B, how many of them is our framework able to detect and how many of them remained undetected? What is the ratio r** $= \frac{\boldsymbol{True\ Posirives}}{\boldsymbol{False\ Negatives}}$ **?**

For completeness, we repeat the above metric evaluation on both Test and Validation sets.

4. It is straightforward to realize that the number of recommendations 'n' makes a difference. The higher the n, the more chances there are to detect a matching pair. For this reason, we will repeat the procedure for n=2,5,10,15 and compare the results. We will try to see how the choice of n affects the results.

6.2 Evaluating the Deep Learning framework with LSF Forest Recommendation: Results

We proceed on presenting the experimental results for DBLP-Scholar and Beer Advocate-Advocate RateBeer dataset. For each dataset, we apply **Algorithm 6.1** (considering the case of complete attribute alignment as C = trained classifier) and evaluate on both Testing and Validation Set true positives as described in sub-section 6.1: We count the number of true matches included in Validation / Test sets that are detected by the framework (**True Positives**), as well as the number of true matches included in the Validation/Test sets that the framework was unable to detect (False Negatives). We repeat the procedure for a various number of recommendations: n=2,5,10,15 and we present the results.

Results For DBLP-Scholar Dataset

In Table 6.2, we provide our experimental results on the **Validation Set**: the number of **Total True Matche**s included in the Validation Set, the number of **True Positive** and **False Negative Predictions**, along with the ratio **r** = $\frac{True\ Posirives}{False\ Negatives}$ **.** What is more, Figure 6.1 depicts the differentiation of the ratio **r** for different number of recommendations (n):

Validation Set (DBLP-Scholar)				
Recommendations	Total Number of True Matches	True Positives	False Negatives	r=$\frac{True\ Positives}{False\ Negatives}$
n=2	1070	625	445	1.404
n=5	1070	666	404	1.648
n=10	1070	684	386	1.772
n=15	1070	694	376	1.845

Table 6. 2: Framework Results on Validation Set(DBLP-Scholar Dataset)

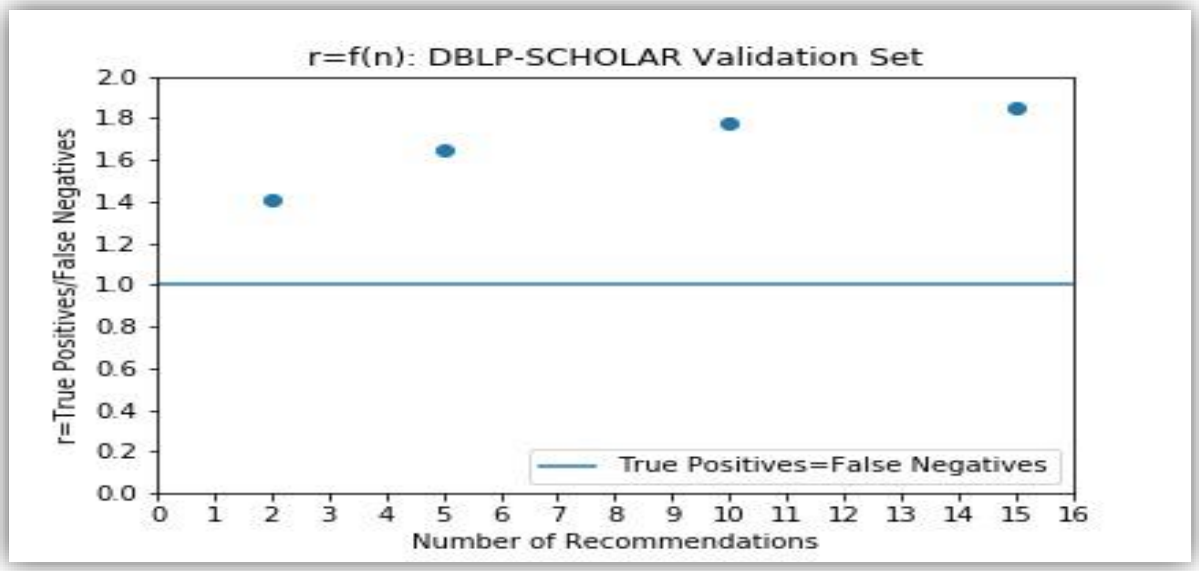

Figure 6. 1: r=f(n) r=$\frac{True\ Positives}{False\ Negatives}$, n=No of recommendations on Validation Set (DBLP-Scholar Dataset)

Similar info regarding the Test Set is also provided in *Table 6.3* and *Figure 6.2*:

Test Set (DBLP-Scholar)				
Recommendations	Total Number of True Matches	True Positives	False Negatives	r=$\frac{True\ Positives}{False\ Negatives}$
n=2	1070	633	437	1.448
n=5	1070	679	391	1.736
n=10	1070	697	373	1.868
n=15	1070	706	364	1.939

Table 6. 3: Framework Results on Test Set (DBLP-Scholar Dataset)

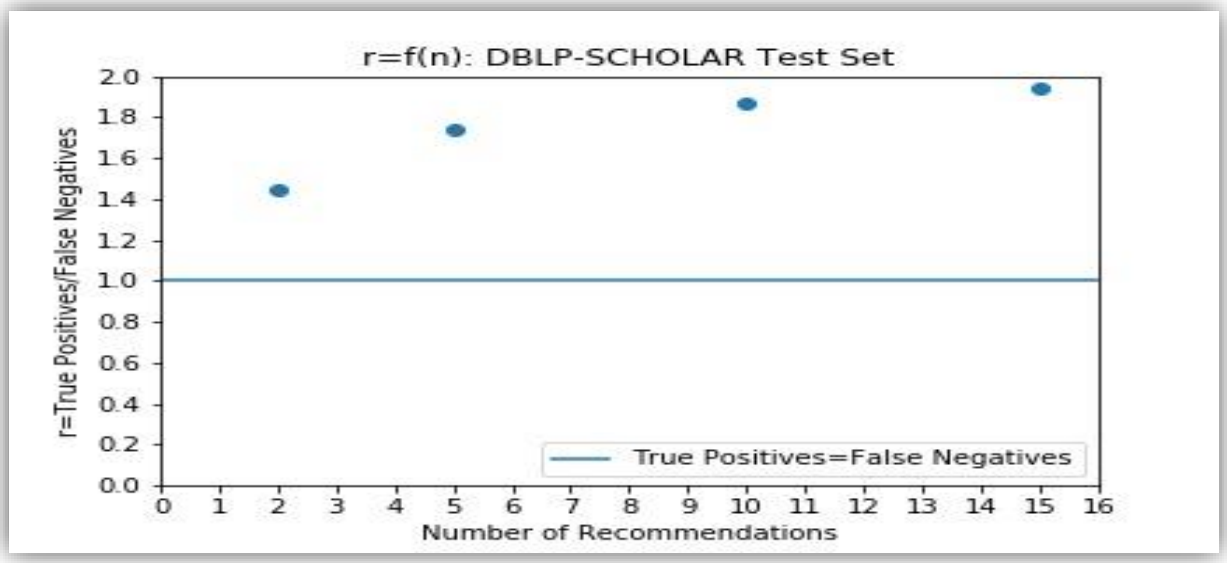

Figure 6. 2: r=f(n) r=$\frac{True\ Positives}{False\ Negatives}$, n=No of recommendations on Test Set (DBLP-Scholar Dataset)

Results For Beer Advocate-RateBeer Dataset

In Table 6.4, we provide our experimental results on the **Validation Set**: the number of **Total True Matche**s included in the Validation Set, the number of **True Positive** and **False Negative Predictions**, along with the ratio **r** = $\frac{True\ Positives}{False\ Negatives}$. What is more, Figure 6.3 depicts the differentiation of the ratio **r** for different number of recommendations (n):

Validation Set (Beer Advocate-RateBeer)				
Recommendations	Total Number of True Matches	True Positives	False Negatives	r=$\frac{True\ Positives}{False\ Negatives}$
n=2	14	7	7	1.000
n=5	14	7	7	1.000
n=10	14	7	7	1.000
n=15	14	7	7	1.000

Table 6. 4: Framework Results on Validation Set (BeerAdvocate-RateBeer Dataset)

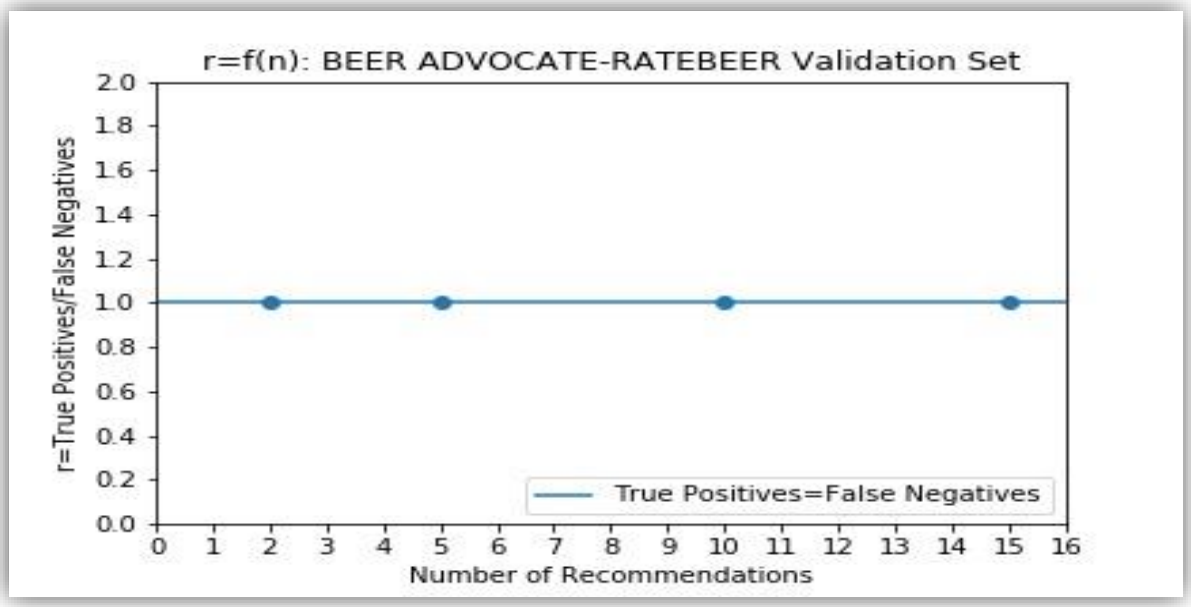

Figure 6. 3: r=f(n) r=$\frac{True\ Positives}{False\ Negatives}$, n=No of recommendations on Validation Set (Beer Advocate-RateBeer)

Similar info regarding the Test Set is also provided in Table 6.5 and Figure 6.4:

Test Set (Beer Advocate-RateBeer)				
Recommendations	Total Number of True Matches	True Positives	False Negatives	r=$\frac{True\ Positives}{False\ Negatives}$
n=2	14	7	7	1.000
n=5	14	9	5	1.800
n=10	14	9	5	1.800
n=15	14	9	5	1.800

Table 6. 5: Framework Results on Test Set (BeerAdvocate-RateBeer Dataset)

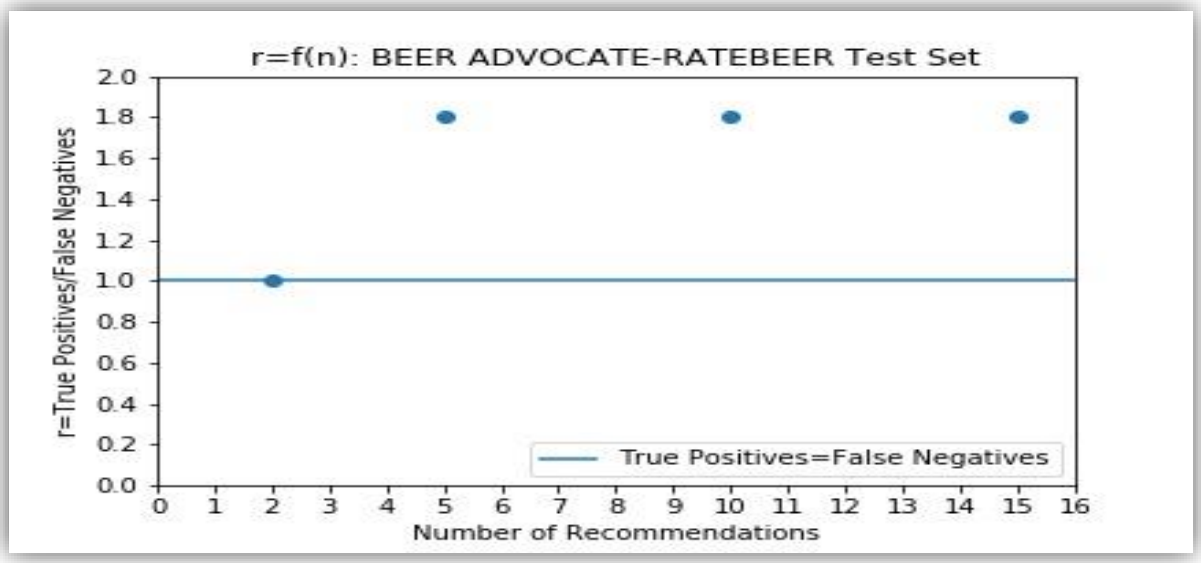

Figure 6. 4: r=f(n) r=$\frac{True\ Positives}{False\ Negatives}$, n=No of recommendations on Test Set (Beer Advocate-RateBeer)

7. Reviewing the Results: Conclusions and Future Work

In this section, we comment on the results for both section 5 (**Deep Learning Approach on Entity Resolution**) and section 6 (**Applying LSH to the Deep Learning Framework**).

7.1 Deep Learning Classification on Entity Resolution: Review and Thoughts

This sub-section is all about reviewing the Deep Learning Classification results (sub-Sections 5.2 to 5.6) and commenting on them. It is crucial to remind the reader about our main goal here: We need to test the ability of a Deep Learning based classifier to deal with the Entity Resolution task for two tables A and B in schema agnostic cases (partial or total agnosticism). In order to do so, the core Test Set classification metrics will be examined for each and every one of the above frameworks. The results for DBPL-Scholar datasets are provided in *Table 7.1*, while the respective results for Beer Advocate-RateBeer dataset are provided in *Table 7.2*:

1) **Modeling Accuracy**
2) **F1-Score**
3) **Average Weighted Recall**
4) **AUC**

DBLP-Scholar Dataset

	Similarity Approach: 4/4 Allignment	Similarity Approach: 2/4 Allignment	Similarity Approach: 1/4 Allignment	Similarity Approach: 0/4 Allignment	Concatenated Strings Approach
Accuracy	0.94	0.94	0.91	0.90	0.89
F1-Score	0.82	0.82	0.75	0.65	0.70
Avg Recall	0.94	0.94	0.91	0.90	0.89
AUC	0.97	0.96	0.93	0.88	0.92

Table 7.1: Concentrated classification results for all of the experimental methods on DBLP-Scholar dataset

It can be seen from the above Table that the best model is the one that assumes total attribute alignment between Table A and Table B. In fact, as the number of misaligned attribute increases, the classification performance of the respective Deep Learning framework drops steadily. However, the important thing is this: **The decrease in the classification performance is minor!**

Taking into account that the last two models (**Attribute Similarity approach with 0/4 attribute alignment** and **Concatenated Strings approach**) make no assumptions about the schemas of the two tables, we should be very pleased with the results: We showed that the difference between Ebrahem's DeepER framework and a **completely Schema Agnostic framework** is actually small, at least in this case! In other words, one could say that we 'paid' a little bit of prediction power in order to 'buy' the luxury of making no assumptions about the tables' schemas.

Beer Advocate-RateBeer

	Similarity Approach: 4/4 Allignment	Similarity Approach: 2/4 Allignment	Similarity Approach: 1/4 Allignment	Similarity Approach: 0/4 Allignment	Concatenated Strings Approach
Accuracy	0.92	0.91	0.91	0.92	0.76
F1-Score	0.74	0.69	0.66	0.74	0.75
Avg Recall	0.92	0.91	0.91	0.92	0.76
AUC	0.92	0.88	0.95	0.91	0.84

Table 7.2 Concentrated classification results for all of the experimental methods on Beer Advocate-RateBeer dataset

For the Beer Advocate-RateBeer dataset, the results were a bit unexpected. For example, there is a clear drop in the classification performance between the 4/4 and 3/4 alignment assumptions. It can also be seen that there is a big difference in the classification performance between the Similarity approach (for any number of assumed attribute alignment) and the Concatenated Strings approach. However, the fact that the case of 0/4 attribute alignment, which is a case of complete agnosticism regarding the dataset's schema, provided the exact same results with the case of 4/4 attribute alignment (even though the cases of 2/4 and 3/4 attribute alignment presented slightly weaker classification power) is surprising. What is more, the Concatenated Strings approach exhibited the highest F1-Score between all models, even if its overall performance was the weakest. Looking back at the results of the **Concatenated Strings** approach at sub-section **5.6**, this can be explained: The CNN-Model was able to capture the highest number of matching pairs compared to all other approaches. However, it also provided a substantial number of False Positives.

Taking into account the results for both DBLP-Scholar and Beer Advocate – RateBeer datasets, we can conclude the following:

1. The case of complete attribute alignment between Tables A and B is the one that achieves the highest classification performance. That is to be expected, of course, since this framework makes the most assumptions about the schemas of the Tables in question.

2. There is some tradeoff between classification performance and schema agnosticism: The more agnosticism is inserted regarding the tables' schemas, the lower the performance of the Neural Network is. However, this tradeoff is acceptable in both dataset cases. In other words, the difference in the classification performance between the cases where we assume complete attribute alignment and the cases where we assume partial or complete agnosticism about the dataset schemas is quite low. **The Entity Resolution task can be solved efficiently in cases where we do not make any assumptions about the tables' attribute alignment**.

3. Unfortunately, the Neural Network architecture that addresses the Entity Resolution Task is dataset specific: We did not manage to come up with a single Deep Learning model that manages to provide good results for both datasets. In fact, for the case of Beer Advocate-RateBeer dataset, we had to call forth a CNN model to enable the Concatenated Strings approach.

4. **For the two different cases of complete schema agnosticism** (**Similarity Approach: 0/4 Alignment** and **Concatenated Strings approach**), **there is no clear answer to what method works better**. For DBLP- Scholar, the Concatenated Strings approach was able to distinguish the matching id pairs better. However, for the Beer Advocate-RateBeer dataset, the Similarity Approach model performed better.

7.2 Reducing the Search Space with LSH and Predicting: Review and Thoughts

In this section, we comment on the results as they are exposed in **Section 6**. Let us remind the reader about our original purpose regarding the application of LSH Forest Recommendation on top of a Deep Learning Classifier for the Entity Resolution Task:

Our initial goal in **Section 6** was to combine a Deep Learning framework with the distributional characteristics of LSH and test its ability to efficiently deal with the Entity Resolution Problem.

It is important to state that, in essence, **the framework consisted of two parts**: First, the **LSH part** of Algorithm 6.1, where the framework provides recommendation documents as answers to an entity-query from one table to another. Secondly, the **classifier's prediction part**, where a pre-trained Deep Learning classifier is used to predict whether the aforementioned entity-query and its recommended document neighbors consist a match or not.

Keeping this in mind, it is straightforward to realize that there are **two obstacles** that the Framework needs to overcome in order to be able to detect matching entity pairs:

A. The LSH algorithmic part needs to accurately provide the true matching entity-documents as recommendations to a specific query. In simpler words, **our Deep Learning classifier would have never been able to detect a True Positive matching pair, and therefore predict on it, if the recommendation LSH part did not manage to provide the True matching pairs correctly**.

B. If a True matching pair indeed manages to pass from the previous LSH Recommendation stage, the classifier has to be able to accurately classify the pair as a 'matching pair' in order to obtain a final True Positive.

In order to test the ability of the framework, we decided to check on the framework's capability of detecting the actual matching entities of the Validation and Test sets, for which we have True Labels in the first place. In order to do so, we checked the number of the two sets' known true matching entities that the framework was able to detect. Finally, we computed the ratio r = $\frac{True\ Positives}{False\ Negatives}$. It is important to understand that this single number depicts the ability of both parts (LSH and Classification) to detect matching entities. We conducted the above procedure for both DBLP-Scholar and Beer Advocate-RateBeer Datasets.

For both datasets, the results seem encouraging: **In any case, it stood that r≥1**. That means that **the framework was able to detect the true matching pairs more often than not**. What is more, **for the majority of dataset instances, r increases when (n) increases**, as expected. This was not the case, however, when predicting on the Validation Set of Beer Advocate-RateBeer dataset. For this particular case, it stood that**: r=f(n)=1, ∀ n∈{2,5,10,15}**.

To be more informative, it can be observed from the respective Tables and Figures of Section 6 that **for the DBLP-Scholar** dataset, in both Validation and Test sets, **r=f(n) is an ascending function for which it stands that r>1 ∀ n∈{2,5,10,15}**. This means that the number of matching pairs that the framework managed to capture was always higher than the number of undetected matches. **It can also be seen from the graphs that the relationship r=f(n) appears to exhibit a particular pattern**.

For Beer Advocate-RateBeer dataset, it can be observed that, for the Validation Set, the number of True Positive predictions is equal to the number of False Negatives ∀ n∈{2,5,10,15}. This is not, however, the case for the experimentations on the Test Set, for which we observe a similar upward trend of r as the number of recommendations (n) increases (possibly still indicating the same pattern as for the DBLP-Scholar Dataset).

All in all, we could say that the framework is able to accurately detect most of the True Matches that exist in the vast majority of the experimental sets (for pairs included in Train and Test Sets respectively). However, by no means it presents perfect results.

7.3 Future Work

Our experimentations on a Distributed Deep Learning Framework on the Entity Resolution problem have a lot of space for improvement and further investigation. Firstly, we provide the reader with some of the main problems that we faced during our experimentations and, in exchange, possible improvements:

1. Beer Advocate-RateBeer dataset exhibited unexpected results on some occasions. Firs of all, the case of 0/4 Alignment Similarity Approach exhibited the same results with 4/4 Alignment Similarity Approach, even though 2/4 and 1/4 approaches performed worse. This might be indicative of the fact that the classification results are dependent on the combination of merging attributes. As a result, **further studies should be conducted by assuming different combinations of merged-misaligned attributes.**

2. For the same reason, it is crucial to **experiment upon more datasets**. Beer Advocate-RateBeer dataset might have some specific peculiarities that make it hard for the framework to perform well.

3. **Experimentations should be made with RNN-based word embeddings**. Even though Spacy is indeed an exceptional tool, the particularities of each dataset might be an obstacle on the mapping procedure of a word to a vector. This could be solved with Recurrent Neural Networks LSTM or GRU cells. In this case, the word vectors should be able to capture these particularities because of the fact that they focus on mapping words to vectors based on a vocabulary that is dataset specific.

4. **The relationship r=f(n) should be furtherly investigated**. Unexpectedly, a pattern seems to have emerged for this relationship during the experimentations. Being able to extract such a mathematical relationship will enable us to choose the appropriate number of recommendations for the desired classification performance.

References

E.Rahm, E.Peukert , 2019. "Large Scale Entity Resolution"

C. Zhao, 2018. Extending DeepER: "A Closer Examination of Deep Learning for Entity Resolution"

V. Christophides, V. Efthymiou, T. Palpanas, G. Papadakis, K. Stefanidis, 2019. "End-to-End Entity Resolution for Big Data: A Survey"

M. Ebraheem, S. Thirumuruganathan, S. Joty, M. Ouzzani, N. Tang, 2018. "Distributed Representations of Tuples for Entity Resolution"

Dražen Oreščanin, 2019. "Conceptual Framework for Entity Integration from Multiple Data Sources"

V. Yadav, S. Bethard, 2018. "A Survey on Recent Advances in Named Entity Recognition from Deep Learning models"

S. Mudgal, H. Li, T. Rekatsinas, A. Doan, Y. Park, G. Krishman, R. Deep, E. Arcaute, V. Raghavendra, 2018. "Deep Learning for Entity Matching: A Design Spacle Exploration"

M. Bawa, T. Condie, P. Ganesan,2005. "LSH Forest: Self-Tuning Indexes for Similarity Search"

K.M. Lee, 2012. "Locality-Sensitive Hashing Techniques for Nearest Neighbor Search"

J. Wang, H. T. Shen, J. Song, and J. Ji, 2014. "Hashing for Similarity Search: A Survey"

T. Mikolov, I. Sutskever, K. Chen, G. Corrado, J. Dean, 2013. "Distributed Representations of Words and Phrases and their Compositionality"

J. Pennington, R. Socher, C.D. Manning, 2014. Glove. "Global Vectors for Word Representations"

Y.LeCun, Y. Bengio, G. Hinton, 2015. "Deep Learning"

J. Schmidhuber, 2015. "Deep Learning in Neural Networks"

P. Domingos, 2012. "A Few Useful Things to Know About Machine Learning"

K. O'Shea, R.Nash, 2015. "An Introduction to Convolutional Neural Networks"

M. Abadi, A. Agarwal, P. Bargham et al., 2015. "Tensorflow: Large Scale Machine Learning on Heterogenous Distributed Systems"

M. Abadi, P, Bargham et al., 2016. "Tensorflow: A System for Large Scale Machine Learning"

E. Loper, S. Bird, 2016. "NLTK: The Natural Language Toolkit"

W. McKineey, 2011. "Pandas: a Foundational Python Library for Data Analysis and Statistics"