Dep. of Computer Science Institute for System Architecture, Database Technology Group

Bachelor Thesis

AUTOMATIC IDENTIFICATION AND SPECIFICATION OF ATTRIBUTE LABELS IN WEB TABLES

Mark Reinke
Matr.-Nr.: 3730496

Supervised by:
Prof. Dr.-Ing. Wolfgang Lehner

and:
Katrin Braunschweig

Submitted on Jan 24th 2014

CONFIRMATION

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, Jan 24th 2014

ABSTRACT

The web contains a vast amount of relational data in the form of HTML tables. But, in contrast to databases, web tables do not carry meta-data that conveys their schema. The relation's attribute labels that we can obtain from the table header are unreliable. The challenge is to identify attribute labels if they are missing in the header, specify them if they are incomplete and substitute them if they are uninformative or false.

In this thesis we describe a two-fold approach to this problem. First, we assign class labels from a knowledge base to columns of a table by linking the column cells to entities in the knowledge base. Second, we perform information extraction techniques to find attribute labels in the context of a web table and to identify the relation name.

We conduct experiments on a web table corpus extracted from Wikipedia using the YAGO knowledge base. The experiments show that our approach in retrieving attribute labels from the contents and the context of web tables is promising.

CONTENTS

1 Motivation **6**

2 Background **8**

2.1 The Semantic Web . . . 8

2.1.1 Ontologies . . . 8

2.1.2 Linked Data . . . 9

2.1.3 YAGO . . . 9

2.2 Natural Language Processing . . . 10

2.2.1 Processing Text . . . 10

2.2.2 Part-Of-Speech Tagging . . . 10

2.2.3 Noun Phrase Chunking . . . 11

3 Related Work **12**

3.1 Web Tables . . . 12

3.2 Keyphrase Extraction . . . 13

3.3 Attribute Extraction . . . 13

4 Approach **14**

4.1 Web Table Corpus . . . 14

4.2 Problem Description 15

4.3 Strategy 17

4.3.1 Linking Columns to Classes in KB 17

4.3.2 Extracting Information from the Context 18

4.3.3 Relation Name Identification 18

5 Identifying and Specifying Attribute Labels **19**

5.1 KB Lookup Module 20

5.1.1 Preprocessing 20

5.1.2 URI Lookup 20

5.1.3 Classes and Distances 21

5.1.4 Scoring and Ranking 21

5.1.5 Algorithm 22

5.2 Information Extraction Module 24

5.2.1 Context Extraction 25

5.2.2 Text Preprocessing 25

5.2.3 Noun Phrase Extraction 26

5.2.4 Algorithms 27

5.3 Annotation Module 29

5.3.1 Candidate Selection 29

5.3.2 String Similarities 30

5.3.3 Duplicate Removal 30

5.3.4 Example 32

6 Experiments **33**

6.1 Corpus and Tools 33

6.2 Evaluation 34

6.2.1 Relation Name 34

6.2.2 Attribute Labels 34

6.2.3 YAGO Simple vs. YAGO Full 35

6.2.4 Information Extraction 36

7 Future Work **38**

8 Conclusion **39**

1 MOTIVATION

The web contains a plethora of information on a huge variety of topics. It is therefore of great significance to find ways to access, analyse, integrate or manipulate this data. Although documents on the web are generally considered to be unstructured they do contain some structured elements.

The HTML table is such an element and is of high interest because it can contain relational data. A lot of research has already gone into web tables. Algorithms have been developed [6] that successfully distinguish tables that hold relational data from those which are used for other purposes such as page layout. A huge corpus of web tables can serve as the basis of a wide range of applications. It has been used for a search system, a schema auto completion tool for databases, or an attribute synonym finding tool [6, 7].

At TU Dresden the Database Technology Group is developing DrillBeyond [9], "a novel database and information retrieval engine which allows users to query a local database as well as the web datasets in a seamless and integrated way with standard SQL". The DrillBeyond system offers the possibility to use open-world queries, i.e. to query for attributes which are not part of the local database's schema. It then uses a keyword-based search to retrieve candidate datasets from the web that may contain the queried information. Finally the system tries to find join candidates in the web tables of the retrieved pages by mapping schema and instances of the local database to the web tables.

The challenge here lies in the fact that for a table extracted from the web no proper schema knowledge is given. At best, a table header is present that can contain the labels of the concepts and attributes of the relation. But these labels may be incomplete, false, uninformative or missing.

Consider the table in example 1.1.

Name	Pop	Code	
Germany	81.9	.de	Berlin
England	53.1	.co.uk	London
France	65.7	.fr	Paris

Table 1.1: Example web table

The header of table 1.1 is a typical example of a web table header. It contains a label that is generic and uninformative (*Name*), an abbreviation (*Pop*), an incomplete label (*Code*) and one that is missing altogether. A human looking at this table will understand that all names in the first column refer to European countries. Knowing that population is a typical attribute of countries, a person will deduce that *Pop* is short for Population. From seeing the contents of the third column, one might infer that it refers to internet codes and that the last column contains capitals.
But, what is a relatively simple assignment for a human is a highly complex one for a computer. Say, we are looking for a join candidate for the *country* concept in a local database. Simply performing a string comparison between *country* and the header label *Name* will not return a match. A more sophisticated approach is necessary to process the web table and its context in such a way as to uncover its inherent schema knowledge.
This paper explores ways to automatically identify and specify attribute labels by analysing the available information. This information can be divided into two parts. One part consists of the web table and the strings stored in its column cells. The other part is the text of the web page that contains the table. The work presented in [17, 14, 12] focuses on the web table itself and does not take into account the context that it is placed in. In this thesis we will pursue a strategy that is two-fold: In the first step we will develop an algorithm that maps the strings in the table cells to individuals in an ontology and uses its taxonomic hierarchy to compute candidate classes that represent the table column. In the second step the text of the web page is analysed. This involves parsing the HTML markup to gather structural information of the page and performing information extraction techniques on the text.
While the first part of the strategy dips into the area known as the Semantic Web, the second part uses techniques from the field of Natural Language Processing (NLP). In the following section some background knowledge on these two areas of Artificial Intelligence will be provided.

2 BACKGROUND

2.1 THE SEMANTIC WEB

The World Wide Web is a vast source of information. This information is generally displayed so that it can be read and understood by humans. While it is possible for a computer to crawl the web and to parse its pages, there is no reliable way for it to process the semantics of a traditional web page. In 2001 Tim Berners-Lee et al. [4] envisioned the Semantic Web that would make this possible: "The Semantic Web is not a separate Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in co-operation."
The Web of today is still a good way off the vision that Berners-Lee portrayed. Most Web pages are regular HTML pages that carry no machine-readable meaning. In the case of web tables, for example, there is generally no meta-data given that would convey the schema of the table, allowing common database operations such as joins to be performed between tables.
On the other hand, huge developments have taken place in the field. A number of knowledge bases have been developed and are continually growing - some with the aim of representing the world's knowledge, others with the aim of storing information on a particular domain such as medicine or biology. They contain knowledge about things, the relationships between them, the abstract concepts they belong to and the relationships between these concepts. But a knowledge base (KB) does not only contain facts, it also comprises a set of inference rules. Given these rules it is possible to infer new knowledge from the given facts.

2.1.1 Ontologies

The conceptualisation that underlies a knowledge base is referred to as an ontology. An ontology is the formal representation of knowledge in a machine-readable format, most commonly RDF, and often extended by the vocabularies of RDFS or OWL. It is modelled as a graph structure, whose nodes are individuals or classes and whose edges are properties. In RDF graphs are stored

as triples that have a sentence-like structure of the form subject-predicate-object. Subjects and objects represent nodes, predicates stand for edges in the graph.
An individual in an ontology refers to an entity of the world, for example to *Dresden*. A class is a set of individuals, e.g. *Dresden* might be an instance of the class labeled *City*. Relationships, known as properties, may exist between individuals, as in the triple: *Mark Reinke - is an inhabitant of - Dresden*. Properties also exist between classes, most notably the subclass relationship that specifies one class being a subset of another. For example *City* might be a subclass of the class *Place*. The subclass property is transitive. In an OWL ontology all individuals are members of the class *OWL#Thing* and thus all classes are by definition transitive subclasses of *OWL#Thing*. The tree structure that is constructed by the subclass relationships is called a taxonomy.
The most common query language for an ontology is SPARQL which has a similar syntax to SQL.

2.1.2 Linked Data

Knowledge stored in ontologies can become even more useful when it is interlinked. This idea was also formulated by Tim Berners-Lee [3] and is known as Linked Data. Berners-Lee postulated four design rules to alleviate the interconnection of data, most importantly the use of HTTP URIs for all resources of a KB. On the one hand this ensures the uniqueness of all names. On the other hand useful lookup services and links to other data on the web can be provided under the URI's web page.

2.1.3 YAGO

The YAGO ontology [15] consists of around 5 million triples that were extracted from Wikipedia and WordNet. It therefore combines the vast encyclopaedic knowledge contained in Wikipedia with the information about words and their semantic relatedness stored in WordNet. It is modelled in the so-called YAGO model, a slight extension of RDFS. We have chosen to use YAGO as our knowledge base as it has proven successful in many Semantic Web applications (e.g. in [12]), and contains more detailed ontological knowledge than for example DBPedia that is extracted only from Wikipedia [1]. YAGO is a part of the the linked data cloud.

2.2 NATURAL LANGUAGE PROCESSING

The main challenge of Natural Language Processing is to make human languages understandable for computers, enabling a machine to derive meaning from natural language input and possibly to respond to it. NLP is a wide field with many different applications including machine translation, speech recognition and sentiment analysis. The field that is relevant for this thesis is information extraction that deals with the automatic extraction of structured information from unstructured or semi-structured documents. In the following a short overview over the relevant techniques and their terminology is given.

2.2.1 Processing Text

As a preliminary step raw text needs to be preprocessed so that operations can be performed on the text in an efficient way. This usually includes the following tasks:

Raw text is transformed into a list words and punctuation known as tokens by a process called **tokenisation**.

Normalisation involves case folding all tokens and either **stemming** or **lemmatising** the words. A stemmer crudely cuts off the ends of words by applying certain rules. The most common representative is the Porter Stemmer that follows a set of condition/action rules on the stem and the suffix of words. Lemmatisation is a more subtle process that attempts to perform a proper reduction of a word to its base stem that is stored in a dictionary such as WordNet as a so-called lemma.

Sometimes it is desirable to exclude common words that are of little value to the task. For example, when searching documents for keywords it might be favourable to eliminate words that occur in every document such as 'and','for','in', etc.. These words are referred to as **stop words**.

2.2.2 Part-Of-Speech Tagging

Part-Of-Speech (POS) Tagging is the process of annotating words in a text with their lexical class. Common lexical classes are *noun*, *verb* and *adjective*. They are defined by the word itself, its morphological form (in English most often determined by its ending) and the syntactical context of the word. A tagger generally takes all these into account. It outputs a list of tagged words. A tag is an abbreviation of a lexical class such as <N> for *noun* or <VB> for *verb*. It can be very general, but also very fine-grained. Some taggers make many distinctions within a lexical category. The Penn Treebank tag list that is used in this work [13] distinguishes between singular and plural nouns or between regular, comparative and superlative adjectives.

2.2.3 Noun Phrase Chunking

After a text has been tagged it can be chunked into sequences of words. Given a pattern or a grammar consisting of a set of patterns, a chunker extracts all word sequences that correspond to this pattern. For example, given the pattern <Adj>*<N>, a chunker would extract all phrases, that begin with zero or more adjectives and end with a noun. This is an example of a base noun phrase. “A base noun phrase is a non-recursive structure consisting of a head noun and zero or more premodifying adjectives and/or nouns.” [2]

3 RELATED WORK

3.1 WEB TABLES

In 2008 Cafarella et al. [6, 7] presented their work on the WebTables system they had developed at Google. They extracted 14.1 billion HTML tables from the web of which they estimated 154 million to be relational tables. So even though only 1.1 % of the tables were regarded relevant, the magnitude of the web surely makes this the largest collection of relational databases. These tables, of course, lack the metadata that is associated with relational databases such as keys or data types. Attribute labels are not always present and even when they are might be false, incomplete, abbreviated or too general.

This thesis focuses on the problem of automatically identifying correct attribute labels. It is inspired by the following work:

Wang et al. [17] find schemas for web tables by associating them with semantic concepts in the knowledge base Probase. The process involves two functions: One to compute the most likely concept for a set of entities in a column, the other to find the most probable concept for a set of attributes. Applying the first function to all columns generates a set of candidate schemas with confidence scores. The second function is then executed on the candidate schemas and the two scores are combined. The aim is to derive a Probase concept that represents the relation of the table as well as to retrieve a table header.

In [14] the focus is on linking information from web tables to the Semantic Web's linked open data collection, specifically the knowledge base Wikitology. Mulwad et al. present two algorithms. The first one, similarly to the first function in [17], computes the Wikitology class that represents a column by identifying and scoring the top N class labels of all strings in the column and choosing the class that maximises its score over the entire column. The second algorithm links every cell in the table to an entity in Wikitology using an SVM rank classifier.

Limaya et al. [12] use machine learning techniques to annotate web tables. A probabilistic graphical model is trained with five features. Again column cells are mapped to entities and entire columns to class labels in a knowledge base. In addition, two features measure whether cells of

different columns that appear together in one row correspond to binary relations in the KB. Limaya et al. show that determining entities and class labels is possible in polynomial time, while identifying binary relations is NP-hard.

In this paper we will develop an algorithm to compute classes for columns that is similar to the ones presented in [14, 12]. But our objective is to enhance these results with information gathered by analysing the text of the web page from which the table is extracted. All approaches that use knowledge bases can only be applied to columns that contain named entities. An approach using natural language processing techniques could not only provide further evidence for such columns, but also might yield results for columns that contain numeric or other values.

3.2 KEYPHRASE EXTRACTION

The technique we will use to analyse the context of the web table is similar to the task of extracting keyphrases from a document. Keyphrases are phrases of one or more words that summarise the contents of a document.
In [16] Peter Turney introduced Extractor, an algorithm for automatic extraction of keyphrases from text. All words in the document are stemmed to a certain word length and the most frequent ones are chosen. For those stems the most frequent sequences of up to three words are picked. Finally some phrases are filtered out due to the parts of speech they contain.
Barker and Cornacchia [2] published a modified version of this algorithm. In contrast to [16] they only considered base noun phrases. Furthermore the choice of keyphrases is based only on the frequency of the noun phrase head, not of any other words the noun phrase contains.

Elements from both these algorithms will be used to extract context information that can provide evidence concerning the attribute labels of the web table.

3.3 ATTRIBUTE EXTRACTION

In [11] Lee et al. devote themselves to the general task of Attribute Extraction irrespective of web tables. They process structured and unstructured data in the form of knowledge bases (Probase, DBPedia), web documents and query logs. Their goal is to retrieve concept-based as well as instance-based attributes. In order to acquire such information from the web, they search for a simple linguistic pattern[1] in the documents. That way they forgo the extraction of noun phrases, which would require the highly expensive task of POS tagging all web documents.

[1] *The <attribute> of (the, a, an) <concept, instance> is*, as in: 'the population of China' (instance-based) or 'the population of a country' (concept-based)

4 APPROACH

4.1 WEB TABLE CORPUS

This work begins at a point where the extraction of tables from the web has already been completed. The corpus at hand contains the following data: Each table is stored in a data structure with the contents of its columns and header, the title of the web page it was extracted from and a unique table identifier. We will proceed from the following assumptions:
Each table represents a relation R. The relation name R describes a concept C with attributes $A_1 \ldots A_n$. The table header contains (not necessarily correct) labels of those attributes. All attributes along with the (unknown) primary key of the relation constitute the relation schema. The cells of a column i contain the attribute values $V_{i,1} \ldots V_{i,m}$ of the corresponding attribute. Each row of the table other than the header row contains attribute values $V_{1,k} \ldots V_{n,k}$ of attributes $A_1 \ldots A_n$.

Name	Pop	Code	
Germany	81.9	.de	Berlin
England	53.1	.co.uk	London
France	65.7	.fr	Paris

Table 4.1: Example web table

Consider the running example shown again in table 4.1: The example relation might carry the name *European Countries*. The header consists of four labels: *Name*, *Pop*, *Code* and an empty string. The attributes they represent we might correctly label as *Country*, *Population*, *Internet Code* and *Capital*. The *Country* attribute could serve as a primary key of this relation. *Germany*, *England* and *France* are values of the attribute *Country*.

4.2 PROBLEM DESCRIPTION

The header in table 4.1 consists of a cell for each column of the table that may or may not correctly represent the contents of the column. Our goal is to write a program that will compute candidate attribute labels for these cells. We will differentiate between the following cases:

Substitution

In table 4.1 the first column is labelled *Name*. While this is a correct description of the column contents it is not very helpful, when taken out of context. It is too generic to accurately represent the column. Therefore it would be desirable to substitute *Name* with, for example, *Country*. As the relation name is not known, it would be even more accurate to retrieve the label *European Country*. A special case of substitution is to substitute an empty string, as in the fourth column of the example where one would like to replace the empty string with *Capital*.

Specification

Consider the third column of the example table. Given the context of the table a human will understand that *Code* stands for *Internet Code*. But taken by itself the label is very vague, and needs to be specified. Linguistically speaking *Code* is the head of the noun phrase *Internet Code*. So we will refer to specification when adding one or more words before a given noun that subsequently functions as the head of the resulting noun phrase.

Completion

In contrast to specification, completion involves adding words to the end of a label. Consider the example header in table 4.2 :

Population	Urban	Rural

Table 4.2: Example header

A human might guess that the words *Urban* and *Rural* refer to population, but they are not unambiguous, and even less so for a computer program. So completion here would mean to complete an adjective such as *Urban* to the noun phrase *Urban Population*. The given label could also be a noun, as in a header where *Unemployment* might in fact mean *Unemployment Rate*. The decisive factor here is that the noun phrase head is missing in the label and is appended by completion.

Expansion

With expansion we will refer to the expansion of abbreviations into their full form. In the header of 4.1 *Pop* should be expanded to *Population*. Of course, the abbreviated form is not necessarily a substring of the full form, e.g. *Avg* expands to *Average*. A header cell might contain more than one abbreviated word, as in *Avg. Sal.*.

Contrary to the other operations, expansion is executed on single tokens, not on the complete string of the header cell. Therefore combinations of Expansion with the other operations are possible. *Avg* $\rightarrow$ *Average Salary* is an example of mixing Completion with Expansion while *Pop* $\rightarrow$ *Immigrant Population* would be an example of combining Specification and Expansion.

4.3 STRATEGY

As stated earlier, we will devise a two-fold strategy to tackle the task of retrieving attribute labels. The first step is to try to link the attributes of a table to classes in a knowledge base.

4.3.1 Linking Columns to Classes in KB

Please consider again the first column of table 4.1 as depicted in 4.3:

Name
Germany
England
France

Table 4.3: Column 1

The idea is first to find individuals in the ontology with equal or similar labels to the strings in the column cells. After we have found one or many candidate labels, we can query the KB for the types of the corresponding individuals. For example, the individual *Germany* might be of type *Country*, *Place* and *OWL#Thing*. As you can see, not every class in this example represents the entity with the same accuracy. We will therefore take into account the KB's taxonomy. This taxonomy is a tree structure that specifies subclass relationships between the classes. In the example, say *Country* is a direct subclass of *Place* which is a direct subclass of *OWL#Thing*. Assigning scores to classes based on their distance to the respective individual in the tree, will favour more specific classes. Classes are retrieved in this way and the scores aggregated over all cells in the column. The top-N class labels which maximise their scores over the entire column are chosen to represent the column. In this case *Country* might be the most specific class for every column cell and thus achieve the highest score.

Proceeding to the next column shown in table 4.4, we will encounter difficulties with this approach:

Pop
81.9
53.1
65.7

Table 4.4: Column 2

Numeric values are not stored as individuals of a knowledge base. This is where the second part of the strategy comes into play - to perform NLP techniques in order to extract relevant information concerning the table from the web page.

4.3.2 Extracting Information from the Context

Searching for the substring *pop* in the web page and ordering the retrieved tokens by their frequency might yield the token *population*. For the third column in table 4.1 the task is a little more complex: We are not searching for a single token that contains another token as a substring. Here a word is given (*code*) that we would like to specify more accurately: The idea is to search the context for noun phrases that contain the string of the header cell as a substring. In this case the search might yield the noun phrase *internet code*. Let us take a closer look at both operations:

$$pop \rightarrow population$$
$$code \rightarrow internet\ code$$

Both target strings, *population* and *internet code*, are noun phrases (a single noun such as *population* is also a noun phrase). So the underlying operation can be defined as finding a noun phrase that contains a given header label as a substring.

4.3.3 Relation Name Identification

Analysing the structure and contents of the web page that contains the table might also provide clues for another important question: What is a valid name of the relation represented by the table?
The contents of a table caption, a preceding headline, the most frequent noun phrases of the text, could be of valuable help to answer this question. Returning to table 4.1, retrieving the relation name *European Countries* from the context of the table would also cast a light on the attribute labels in the header: In the context of the relation name *European countries* we would know what the word *Name* refers to. We could further infer that the population column contains the population values of the countries in the first column, not of the capitals in the fourth column.

In the next chapter the program that was developed in accordance to the presented strategy will be described in detail.

5 IDENTIFYING AND SPECIFYING ATTRIBUTE LABELS

The program we have designed to identify and specify attribute labels consists of three modules: The *KB Lookup Module*, the *Information Extraction Module* and the *Annotation Module*. Corresponding to the strategy outlined in the previous chapter, the *KB Lookup Module*, links columns to class labels in an ontology and the *Information Extraction Module* performs NLP techniques to extract information from the context of the web tables. The *Annotation Module* collects all the results and performs postprocessing before annotating the tables. Figure 5.1 gives an overview of the program and shows the tasks that are performed in the three modules.

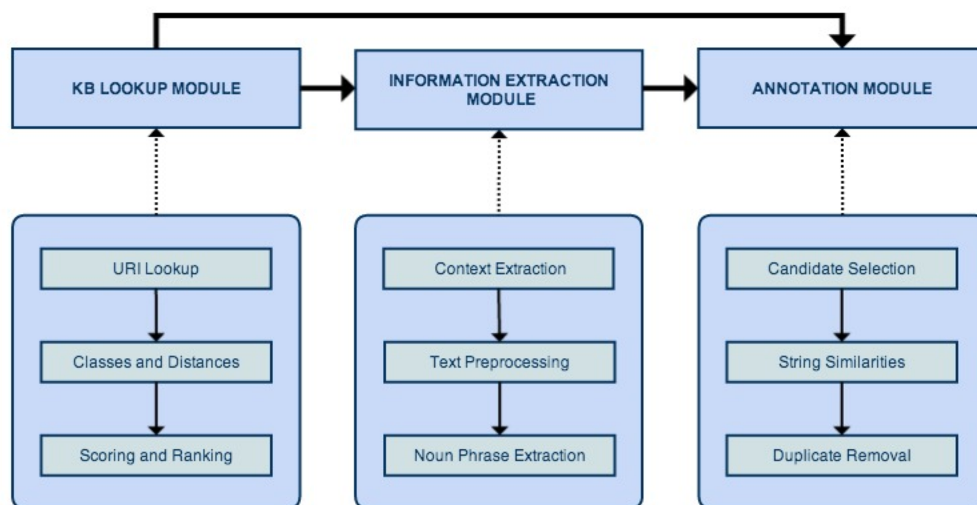

Figure 5.1: Overview

The arrows between the modules indicate that the *Information Extraction Module* takes the output of the *KB Lookup Module* as input and the *Annotation Module* collects the output from both the other modules to annotate a table.

5.1 KB LOOKUP MODULE

The assignment of the *KB Lookup Module* is to link the strings in table columns to entities (individuals) of a knowledge base and to find class labels that represent the column by analysing the class membership of the retrieved entities and the taxonomy of the KB.

5.1.1 Preprocessing

A general-purpose knowledge base such as YAGO only contains named entities as individuals. But web tables also contain data in the form of numeric values, dates, boolean values, etc.. Except in special cases, such as the number π or the date *9/11/2001*, the KB does not hold these values as individuals, so we cannot obtain any class membership information on them. To prevent a magnitude of redundant lookups in the KB, i.e. looking up column cells that do not have corresponding entities in the KB, we devised an algorithm to identify those columns that contain values that are of type *string*. The task is not as straight-forward as it may seem, as there are a lot of cases where the column cell contains numbers as well as letters. For this work it is more valuable to achieve high precision than high recall, as the corpus of web tables is huge. Therefore we opted for an algorithm that strictly filters out all columns that contain at least one cell that includes a number. In addition, columns that do not contain strings that are longer than one letter or that are non-ascii are excluded.

5.1.2 URI Lookup

The goal of the URI lookup is to link the strings in the table columns to individuals in the knowledge base. These individuals are represented by URIs. YAGO is based on DBPedia and WordNet. While its classes are derived from both sources, the individuals of YAGO are individuals of DBPedia [15]. DBPedia provides a lookup service, that returns a list of ranked URIs to a string input if it finds individuals with similar labels to the given string. The matches between the string and labels do not need to be exact, as the lookup service performs auto-completion on the string.
The rank of the returned URIs is based on the PageRank of the page within Wikipedia that corresponds to the entity, i.e. the number of links from other wikipedia pages that point to the page [10]. To account for cases where the column string is a synonym or abbreviation of the entity's label, e.g. 'EU' referring to 'European Union', the PageRank also includes Wikipedia redirects. The lookup uses a Lucene Index and combines the described relevance metric with Lucene's string similarity matching.
So the rank of the URI gives us an idea how probable it is that the URI represents a KB entity that corresponds to the column cell. We therefore retrieve the URIs along with their rank and incorporate the rank into the scoring of class labels as we will see in the next section.
It is possible to determine how many URIs one wants the lookup to return at a maximum. We set this maximum with the variable *maxRank*.

5.1.3 Classes and Distances

We use two SPARQL queries to retrieve class membership and subclass relationship information on the individuals returned by the URI lookup. The first query simply returns all classes that contain the individual directly, excluding those that contain the individual transitively by inference over subclass relationships. The second query returns all direct super classes of a class, i.e. all classes of which it is a direct subclass. Using these queries *Limited Depth First Search* (LDFS) is performed to retrieve the classes that contain the individual. The individual is the starting node, class membership or subclass relationship serve as the edges of a graph, and the classes of the KB as the nodes. This graph is now traversed with LDFS and the classes and their depth returned. The depth in the graph where a class is found is equivalent to the class's distance from the individual. The limit of the search is set with a variable $maxDist$ to make the search more efficient, and to ensure that no classes are found that are too distant from the individual and thus too general. The function is depicted in Algorithm 1.

Input: $graph \leftarrow$ ontology
$startNode \leftarrow$ URI of an individual in KB
$limit \leftarrow$ maxDist
Output: list of classes and their distances to the given individual
$classesAndDistances = LimitedDepthFirstSearch(graph, startNode, limit)$;
return *classesAndDistances*

Algorithm 1: LDFSTaxonomy

5.1.4 Scoring and Ranking

For each string in the column at most $maxRank \times maxDist$ classes are found. For every class we have stored its taxonomic distance to the entity that was found with the URI lookup ($\rightarrow$ *distance*) and the rank of the entity's URI ($\rightarrow$ *rank*). The lower the values of *distance* and *rank* the more specific the class and the more probable it is that the retrieved URI correctly identifies the string. Therefore we use a scoring function based on the inverse of *distance* and *rank* as shown in 5.1.

$$score = \frac{1}{rank} \times \frac{1}{distance} \tag{5.1}$$

Thus, the most specific class of the top ranked URI of a given string gets the score 1.
All classes found with *LDFSTaxonomy* for all URIs returned by the lookup service are scored, and the scores then aggregated over all distinct classes and added up. So, for a column containing strings for which the corresponding entities in the KB are found, only a small number of distinct classes are retrieved and achieve high scores through the aggregation. On the contrary, a column containing strings that the KB cannot make much sense of, yields many different, unconnected classes that do not accumulate high scores with the aggregation. The scores are normalised by dividing by the number of URIs found in the lookup.
When very few results are returned the results could be distorted. Say, the lookup only returns one URI for a column of ten cells. The direct class of the entity in the KB would then attain the highest possible score of 1 after normalisation and thus a class label is returned with a top score although it is only associated with one cell in the column. To prevent this from happening, the

divider for normalisation is changed to the number of column cells (in this case 10) if this value is higher than the number of URIs found.

5.1.5 Algorithm

Putting all the pieces together, we acquire the following main algorithm for the *KB Lookup Module*:

```
Input: graph ← ontology
       column of strings
       maxRank
       maxDist
Output: ordered list of class labels
classesAndScores = List();
foreach string in column do
    lookup URIs A_1 ... A_n with n ≤ maxRank ;
    for 1 ≤ i ≤ n do
        rank ← i;
        classesAndDistances = List();
        classesAndDistances ← LDFS(graph, A_i, maxDist);
        for class, distance in classesAndDistances do
            score ← 1/rank × 1/distance;
            normalise score;
            classesAndScores.append(class, score)
        end
    end
end
aggregate and add up scores of all distinct classes in classesAndScores ;
classLabelsAndScores ← get labels of classes from KB ;
sort classLabelsAndScores by score ;
return classLabelsAndScores
```

Algorithm 2: computeClassLabels

We will illustrate the *computeClassLabels* algorithm with an example. The fourth column from table 4.1 will serve as input:

Berlin
London
Paris

To keep things simple we set *maxRank* to 1. That way the URI lookup service retrieves one URI for every string in the column. Let us assume the lookup returns the following URIs:

Berlin → *http://yago-knowledge/resource/Berlin*
London → *http://yago-knowledge/resource/London*
Paris → *http://yago-knowledge/resource/ParisHilton*

In the following we will not use full URIs, but refer to them by their labels.
The next step in the algorithm is to search the KB with LDFS using the retrieved URIs as input. We will set the variable *maxRank* to 2, so that the graph is searched up to a depth of 2. Assuming the relevant excerpt of the KB is as depicted in figure 5.2, the classes and distances will be returned as shown in table 5.1.

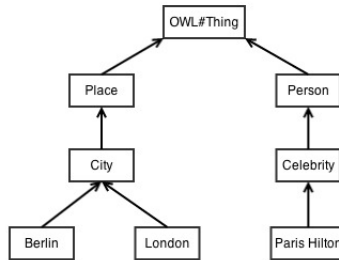

Figure 5.2: Example excerpt of KB. The leaves of the graph represent entities, all other nodes classes. The arrows denote class membership (on the bottom level) or subclass relationship (all other levels)

Column Cell	Entity Label	Class Labels (Distances)
Berlin	Berlin	City(1), Place(2))
London	London	City(1), Place(2)
Paris	Paris Hilton	Celebrity(1), Person(2)

Table 5.1: Entities, classes and distances

Including aggregation and normalisation we obtain equation 5.2 to score each class:

$$score_{class} = occurrences_{class} \times (\frac{1}{rank} \times \frac{1}{distance}) \div numURIs \tag{5.2}$$

This yields the following results with the classes ordered by their scores.

$$\begin{aligned} score_{City} &= 2 \times (1 \times \frac{1}{1}) \div 3 = \frac{2}{3} \\ score_{Place} &= 2 \times (1 \times \frac{1}{2}) \div 3 = \frac{1}{3} \\ score_{Celebrity} &= 1 \times (1 \times \frac{1}{1}) \div 3 = \frac{1}{3} \\ score_{Person} &= 1 \times (1 \times \frac{1}{2}) \div 3 = \frac{1}{6} \end{aligned}$$

The *KB Lookup Module* outputs an ordered list of class labels for every column that contains strings. For the example the list would look like this: $[City(0.66), Place(0.33), Celebrity(0.33), Person(0.17)]$

The results of the *KB Lookup Module* will now be enhanced with results obtained by the *Information Extraction Module* before being filtered by the *Annotation Module*.

5.2 INFORMATION EXTRACTION MODULE

The idea behind the *Information Extraction Module* is to find attribute labels by searching the context of the table for noun phrases (NPs) that contain headings of the table as substrings and to retrieve a relation name from the context.

Schools [edit]

The Board operates 76 elementary schools, and 16 secondary schools which are organized into the following areas

- East Area (Oakville)
- North Area (Halton Hills and Milton)
- West Area (Burlington)

Secondary school enrollment and Fraser Institute provincial rankings[2] are as follows:

HDSB secondary schools

Name	Area	Enrollment	1-year ranking of 727	5-year ranking of 693
Abbey Park High School	East	1282	34	16
Iroquois Ridge High School	East	1324	16	22
Oakville Trafalgar High School	East	1288	4	10
T. A. Blakelock High School	East	1150	63	60
White Oaks Secondary School	East	1627	126	130

Figure 5.3: Table with context

In the example we see the potential of such an approach. Figure 5.3 shows an excerpt from a web table in Wikipedia and its context. Text snippets are marked in different colours according to the header cell they are connected with.
For example the heading 'enrollment' is described more accurately by the noun phrase 'secondary school enrollment' in the preceding paragraph. This is an example of *specification* as it was defined earlier. The same goes for 'provincial rankings' which is a *specification* of 'ranking'. Furthermore the caption of the table 'HDSB secondary schools' is a correct relation name for the table, as is, in a more general sense, the headline 'Schools'. The occurrence of the noun phrases 'elementary schools' and 'secondary schools' provides further evidence that the table is related to schools. Knowing what the table is about would also reveal what is in the *Name* column. [1]
In the *Information Extraction Module* the web table's context is first extracted and then preprocessed. Finally Information Extraction techniques are applied to retrieve attribute label candidates and a potential relation name.

[1]The example also shows a difficulty of this approach: Searching the text for the header string 'Area' might yield the noun phrases 'East Area', 'North Area', 'West Area' and 'following areas'. But none of these phrases would serve as a good label for the *Area* column. This problem is discussed in section 6.2.4 .

5.2.1 Context Extraction

The context of a web table is extracted by acquiring and analysing the HTML markup of the web page that contains the table. We divide the context into three parts: *SurroundingText*, *headline* and *caption*. *SurroundingText* is the text that surrounds a web table, i.e. the paragraphs before and after a table. In figure 5.3 this would be the paragraph from "The Board ..." to"... as follows:". In the example *caption* is 'HSDB secondary schools' and *headline* is 'Schools'.
The first step is to locate the given web table on the web page. This is done by comparing the table headers. If multiple tables were extracted from one page then we also ensure that the order of the extracted tables corresponds to the order of the tables on the page. When the table is located the table caption can be easily retrieved. *SurroundingText* is obtained by extracting all plain text between the table's preceding headline and the subsequent headline if present. The preceding headline is stored as *headline*.

5.2.2 Text Preprocessing

Some preliminary steps are necessary before we search the context of the table for noun phrases: Tokenisation, POS tagging, Noun Phrase Chunking and normalisation.
First the raw text stored in *surroundingText* is tokenised. For POS tagging it is essential to keep the text segmented into sentences. So tokenisation here is performed on sentences and the tokens are not stored as one big list, but divided into lists for every sentence in the text.
For Part-Of-Speech tagging we use a tagger based on a Maximum Entropy Model that computes the most probable tag based on a set of features. A feature could be, for example, "does the word end with 'ed' " [13].
Once the text has been POS tagged, we can search for noun phrases based on a sequence of tags. There are many ways to define the pattern of a noun phrase. We have opted for a pattern of zero or more Adjectives followed by at least one noun. It would also be possible to allow, for example, determiners ('the', 'a', ...) or possessive pronouns ('my', 'our', ...) in a noun phrase, but these would not be valuable for the *specification* or *completion* of an attribute label.
This way *surroundingText* is chunked into a list of noun phrases. As the letter cases as well as the exact endings of the words are of no relevance here, the noun phrases and the header cells need to be normalised. For example one would want a search for the cell string 'school' to yield results such as 'Secondary School' or 'secondary schools'. This is ensured by lower-casing and lemmatising all tokens. We chose lemmatisation over stemming as advised in [2], in order to match different forms of a word even when they do not have a common stem ('better' → 'good', 'worse' → 'bad').

5.2.3 Noun Phrase Extraction

Now that everything is prepared we can search the list of noun phrases for occurrences of the strings in the header cells. A special case needs to be taken into account:
Web table headers often contain abbreviations. Searching for a string such as 'pos' might deliver unwanted results such as 'supposing'. Therefore we only allow the substring to be positioned at the beginning of a string. This does not prevent all artefacts, but it limits them. Furthermore we use a list of stop words to exclude common, generic strings. This list includes, for example, the very common header cell strings 'name' and 'title' [2]. These strings are too generic to truly represent the columns. As we saw in table 4.1 a *name* column might, for example, contain countries. Searching the noun phrases for the string 'name' here would not yield results that specify the label 'country', but might return totally unconnected noun phrases.
In addition to searching the noun phrases for occurrences of the header cells, we also search for the class labels that are outputted by the *KB Lookup Module*. Please consider the first column in the web table of figure 5.3. Say, the *KB Lookup Module* has the class labels 'school' and 'secondary school' as its output for this column. So instead of searching for the header cell 'name' which is a stop word, the text is searched for the two class labels. Both searches would yield, amongst others, the noun phrase 'secondary school' and thereby provide further evidence that this is the most accurate attribute label for the column.
The noun phrases are scored according to their frequency:

$$score_{NP} = occurrences_{NP} \div numNPs \tag{5.3}$$

The score refers to normalised noun phrases and thus the occurrences of normalised NPs are an accumulation of the occurrences of all the noun phrases that have been normalised to one form. For example, if 'Secondary School' occurs once and 'secondary schools' twice then the normalised version 'secondary school' has three occurrences.

[2]Roughly 10 % of the columns we extracted had these column headings.

5.2.4 Algorithms

The *findNounPhrases* algorithm (3) is presented in Pseudocode as described in the previous sections.

Input: *header* ← table header
classLabels ← class labels found for each table column with computeClassLabels
nps ← noun phrases found in surrounding text
stopWords ← list of generic strings
Output: *cellNPsAndScores* ← noun phrases that correspond to cell, scored by frequency
classNPsAndScores ←noun phrases that correspond to KB classes, scored by frequency

```
normNPs ← normalise noun phrases in nps;
foreach cell in header do
    normCell ← normalise cell ;
    normClassLabels ← normalise classLabels of the respective column;
    if normCell not in stopWords then
        cellNPs ← find noun phrases in normNPs that contain normCell as substring;
        cellNPsAndScores ← score noun phrases in cellNPs by their frequency in
        surroundingText;
    end
    classNPsAndScores = List();
    foreach classLabel in normClassLabels do
        classNPs ← find noun phrases in normNPs that contains classLabel as substring;
        scores ← score noun phrases in classNPs by their frequency in surroundingText;
        classNPsAndScores.append(classNPs, scores);
    end
end
return cellNPsAndScores, classNPsAndScores
```

Algorithm 3: findNounPhrases

In addition to searching for substrings in the document, we also extract the document's keyphrases, in order to examine if they can be of help in determining the relation name of the table. Algorithm 4 is very similar to the one presented in [2] and therefore is not described here in detail.

Input: $nps \leftarrow$ noun phrases found in surrounding text
Output: $keyphrases \leftarrow$ ordered list of keyphrases
$normNPs \leftarrow$ normalise noun phrases in nps;
get noun phrase heads of noun phrases in nps;
normalise heads;
$topKHeads \leftarrow$ find top-K most frequent normalised heads;
$keyphrases = List()$;
foreach *head in topKHeads* **do**
  $nps_{head} \leftarrow$ get corresponding noun phrases from $normNPs$;
  score noun phrases in nps_{head} by occurrence of NP in *surroundingText* multiplied by number of words in NP;
  rank nps_{head} by score and append top-M NPs to $keyphrases$;
end
sort $keyphrases$ by occurrence of NP-heads, if equal sort by occurrence of NP;
return *keyphrases*

Algorithm 4: extractKeyphrases

Summing up, the *Information Extraction Module* outputs the following for each web table in the corpus:

caption	$\leftarrow$ the table caption
headline	$\leftarrow$ the headline that precedes the table
keyphrases	$\leftarrow$ a (possibly empty) ordered list of the keyphrases of the text surrounding the table

And for every column of each table:

cellNPs	$\leftarrow$ a (possibly empty) ordered list of noun phrases that contain the header cell as a substring
classNPs	$\leftarrow$ a (possibly empty) ordered list of noun phrases that contain a class returned by the *KB Module* as a substring

5.3 ANNOTATION MODULE

When the *KB Lookup Module* and the *Information Extraction Module* have been traversed the following information has been acquired:
For every table we have obtained *caption*, *headline* and *keyphrases* and for every column of every table *classLabels*, *cellNPs* and *classNPs*. It is now the assignment of the *Annotation Module* to filter out the relevant results and to uncover connections and similarities between the results. In a last step duplicate attribute labels are removed before the table is annotated.

5.3.1 Candidate Selection

We divide the annotation problem into two different tasks:
Finding a relation name for a table and identifying attribute labels for the columns of a table.

Relation Name

For the task of finding a relation name there are three candidate sources: *caption*, *headline* and *keyphrases*. We consider it most probable that the table caption contains the relation name, followed by the headline containing it. If both are not present we choose the top scoring keyphrase of the text that surrounds the table. The method we use is straight-forward and shown in algorithm 5.

Input: *caption*, *headline* and *keyphrases*
Output: *relationName*
if *caption* **then**
| $relationName = caption$
else if *headline* **then**
| $relationName = headline$
else if *keyphrases* **then**
| $relationName =$ top scoring keyphrase in $keyphrases$
return *relationName*

Algorithm 5: findRelationName

Attribute Labels

For the task of identifying attribute labels there are again three sources: *classLabels*, *cellNPs* and *classNPs*. To filter out valid labels stored in *classLabels* we set a threshold to only allow labels above a certain score. Also, in the case that *OWL#Thing* passes above the threshold all classes that score lower than *OWL#Thing* are excluded. We argue that a class that scores lower than the most general class is probably not a valid label.
In the case of *cellNPs* and *classNPs* it is not possible to set a threshold to separate valid labels from invalid ones as the frequency based score that we use is only a very limited indication of the quality of a label. This could be more closely examined in future work.
We select the top-N labels in *classLabels*, *cellNPs* and *classNPs*. Then the scores of each distinct

result are aggregated and added up. So, if the same result is found with different methods, the score of this result is enhanced.

5.3.2 String Similarities

The *computeClassLabels* algorithm (2) in the *KB Lookup Module* that outputs candidate class labels for a column does not take the title of the column into account. But we argue that a class label that is similar to the string in the header is more probably correct than one that is not. To account for this we compute the string similarity between the class labels and the string in the header cell. The most common metric for the distance between two strings is the Edit Distance. But this method is designed to detect spelling mistakes and is therefore not a true reflection of lexical similarity. It does not attach enough weight to a high percentage of common substrings and it is not robust to a reordering of words. For example, we would want the similarity between the string 'secondary school' and the string 'school' to be rated higher than the similarity between 'state' and 'school'. This is not the case when applying the Edit Distance. We have therefore chosen an algorithm presented in [18] that computes the similarity based on the common bigrams in the two strings using the Dice Coefficient [8]. Bigrams are all sequences of two tokens in a string, the tokens in this case being letters. The formula for string similarity is depicted in 5.4:

$$sim(s1, s2) = 2 \times \frac{|bigrams(s_1) \cap bigrams(s_2)|}{|bigrams(s_1)| + |bigrams(s_2)|} \tag{5.4}$$

Those columns that have the title *name* or *title* are treated as a special case:
Here there is no point in computing the similarity between the column title and the class labels as we know the column title is a generic string that does not truly represent the contents of the column. But we argue that such a column title almost always refers to the concept described by the relation name. In the example of figure 5.3 the algorithm *findRelationName* (5) would compute the relation name 'HDSB secondary schools'. Assuming again that the *KB Lookup Module* outputted *secondary school* and *school* as class labels, computing the similarity to the relation name would correctly reflect the validity of these labels, especially the label 'secondary school'.

5.3.3 Duplicate Removal

In some cases it can happen that the same attribute labels are computed for two different columns in one table. Assuming that the tables are relational we know that one of the annotated labels must be incorrect as no two columns in a relational table can represent the same attribute. This problem might also arise in example 5.3. Here the *Information Extraction Module* would assign the noun phrase 'secondary school enrollment' to the *Name* column (supposing the class labels *school* or *secondary school* were identified by *computeClassLabels*) as well as to the *Enrollment* column. On the evidence that 'secondary school enrollment' is the only noun phrase found for the *Enrollment* column, but one of three ('elementary school', 'secondary school' and 'secondary school enrollment') found for the *Name* column, it could be discarded from the candidate list for the *Name* column.

Population	Urban	Rural

Table 5.2: Example header

Please also consider again the example shown in table 5.2. Here the same problem might occur. The *Information Extraction Module* could come up with the noun phrase ‘urban population’ and assign it to the *Population* column as well as the *Urban* column. *urban* $\rightarrow$ *urban population* is an example of *completion* while *population* $\rightarrow$ *urban population* is an example of *specification*. We argue that it is far more likely in such cases that *completion* yields the correct result while *specification* does not. It is hard to imagine a header looking like the one shown in 5.3, where ‘Total’ signifies ‘Total Population’ and ‘Population’ denotes ‘Urban Population’ :

Total	Population	Rural

Table 5.3: Example header

But there are of course exceptions to this assumption (the ‘secondary school enrollment’ example is such an exception) and it would have to be underpinned with statistical measures. This is beyond the scope of this thesis.

5.3.4 Example

To illustrate the way the results are scored and ranked, we will again use the *Name* column from example 5.3. We will assume that the *KB Lookup Module* returned the two classes *school* and *secondary school* with the scores 0.3 and 0.2. The *Information Extraction Module* does not search for noun phrases containing the header cell 'Name' as it is a stop word. So *cellNps* is empty. It does search for noun phrases containing the class labels *school* and *secondary school* though and returns the results depicted in 5.4.

classLabels	classNPs
school (0.3)	elementary school (0.1) secondary school (0.1)
secondary school (0.2)	secondary school (0.1)

Table 5.4: Class labels, noun phrases and their scores

Then the similarity scores are computed, in this case the string similarity to 'HDSB secondary schools'. This is shown in 5.5.

Results	Sim Score
secondary school(0.4)	0.8
school(0.3)	0.4
elementary school(0.1)	0.5

Table 5.5: Similarity scores

The similarity scores are added to the existing score and the end result would then be the following ordered list:
$[\text{secondary school}(1.2), \text{school}(0.7), \text{elementary school}(0.6)]$

Summing up, the *Annotation Module* collects the results from the other two modules and processes them, selecting the top results, adding similarity scores and removing duplicates. Finally it annotates every table with a relation name and every column of a table with a ranked list of the top-N labels.

6 EXPERIMENTS

6.1 CORPUS AND TOOLS

We chose to perform our experiments only on web tables extracted from Wikipedia pages.
Wikipedia pages offer the advantage that their layout follows a fixed structure. This makes it fairly simple to target specific sections of the document as we do in the *Information Extraction Module*. We incorporated ideas and programming code from [5] to extract the context of web tables. The program was written in Python. Python was chosen mainly for the one reason: It offers the NLTK and RDFLib packages.
The Natural Language Toolkit (NLTK) is a taylor-made platform for Natural Language Processing. It includes the tools for tokenising, lemmatising, POS tagging and chunking used in our program out of the box. The RDFLib package is an RDF API for Python. It makes it possible to integrate RDF concepts such as graphs and triples into the programming environment as Python data structures. It also supports the query language SPARQL and thus allows querying of an RDF ontology. RDFLib offers support of OpenLink Virtuoso, an open-source triple store [1]. So we were able to store the YAGO ontologies in the triple store and integrate them into the Python programming environment with RDFLib.

[1] a special database designed to hold RDF triples

6.2 EVALUATION

The experiments were conducted using two different YAGO ontologies. YAGO offers a full version of its taxonomy that contains roughly 366.000 classes and a reduced version holding 6543 classes. So the difference in granularity is considerable here. We will refer to the ontology based on the full taxonomy as YAGO Full, the other as YAGO Simple.
The evaluation was performed by one human judge on 50 web tables that altogether contained 115 columns. For each table a relation name was presented and for each column a maximum of 6 attribute labels ranked by their score. The judge was asked to evaluate the relation names and the top ranked attribute labels. Correct results were assigned a score of 100, results that were correct but too general 50 and false ones 0. For example, for a column listing politicians the label *Politician* would achieve a score of 100 whereas the label *Person* would be scored with 50. He was also asked to evaluate the complete list of labels presented for every column by awarding 100 if all results were correct, 50 if the majority was correct and 0 if the majority were incorrect. Furthermore the judge was asked to answer two question with yes or no: Is the top ranked result the best possible result? and: Is there at least one correct result in the list?

6.2.1 Relation Name

The Relation Name was computed correctly with a precision of 0.89. One has to bear in mind that this high result is due to the fact that Wikipedia pages are well organised and it is possible to reliably retrieve table captions and headlines. How well the method would perform on random web pages could be evaluated in future work.
All correct relation names were extracted from table captions or the preceding headline. The results originating from the keyphrases of the text surrounding the table were all incorrect. Based on this evidence one could discard this idea.

6.2.2 Attribute Labels

The results for the evaluation of attribute labels are depicted in figure 6.1. The precision of the top ranked attribute labels for the experiments with YAGO Simple is 0.66. For columns containing named entities precision is 0.73. There was at least one correct answer in the list for 70 % of the columns and the top result was considered the best possible result in 40.6 of the cases. When looking at all results for a column 55.4% were deemed correct. The reason for this value being lower than the value for top ranked results (65.8%) is due to the fact that often top ranked results were correct while the complete lists of results also included incorrect results.

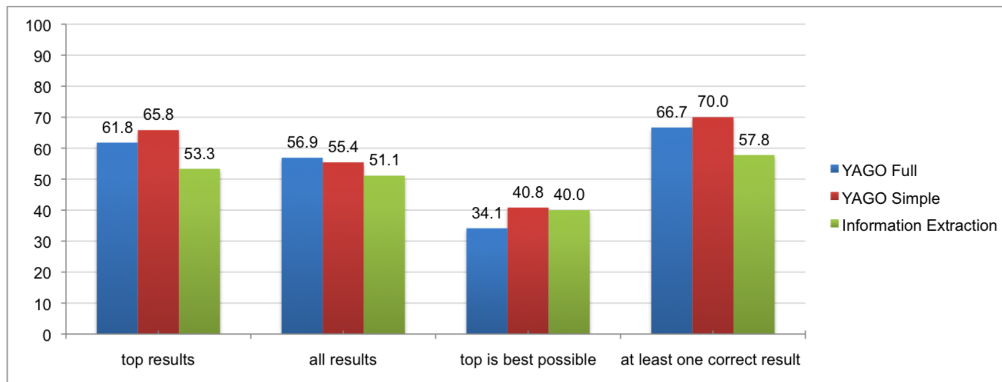

Figure 6.1: Attribute Labels

We analysed the results by distinguishing between the categories of the respective columns. We differentiated between *Organisation*, *Person*, *Place*, other named entities (*Named Entity*) and all other categories (*Other*). The results using YAGO Simple are shown in figure 6.2.

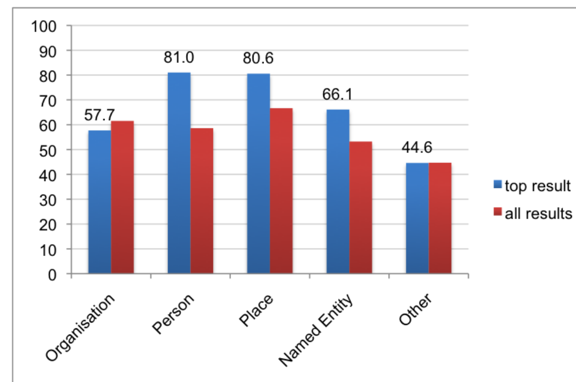

Figure 6.2: Results by Categories with YAGO Simple

We see that the program works well for places and persons, but does not perform as well for columns holding organisations or other named entities. This corresponds to the results evaluated in [14] and supports their assumption that there is less information in the KB about these kinds of entities. The result for the *Other* category that comprises datatypes such as numbers, dates, booleans and strings that are not named entities is the lowest. This reflects the fact that values stored in these columns are generally not entities of the KB. Therefore only the *Information Extraction Module* has an effect here.

6.2.3 YAGO Simple vs. YAGO Full

The results computed with YAGO Full (figure 6.3) are less accurate (61.8% top results correct) than those computed with YAGO Simple (65.8%). This is surprising due to the fact that YAGO Full holds more than 50 times as many classes as YAGO Simple. The class membership information computed with YAGO Full is much more fine-grained. But in fact this is exactly why it does not perform so well in some cases: For example, for a column containing companies that are mainly Italian the class label *Manufacturing Company of Italy* scores the highest. But this is not the

correct attribute for the column as not all of the companies are Italian. The class label is too specific for what would generally be an appropriate attribute label in a web table. Other examples for this phenomenon include labels such as *British Formula Three Championship Driver* or *Country Bordering the Atlantic Ocean*. All these class labels are extracted from Wikipedia categories. In future versions the program could take into account that these classes tend to be somewhat wayward and filter them out.

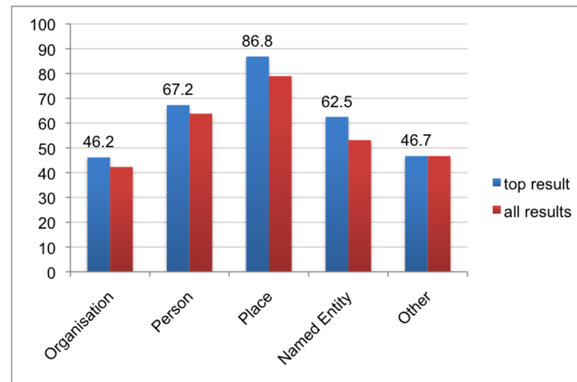

Figure 6.3: Results with YAGO Full

6.2.4 Information Extraction

As the *Information Extraction Module* implements a novel approach to the retrieval of attribute labels, we also evaluated results that are based only on the output of this module. These results achieve a precision of 0.533. The relatively low value for precision is due to the fact that we have not yet developed a reliable method to measure the quality of the retrieved noun phrases. Specifically we observe two kinds of problems that arise from this approach:

Some noun phrases are too specific and refer only to some instances of the column. The following cases show examples of header strings and the retrieved noun phrases:

$$\textit{area} \rightarrow \textit{east area}$$
$$\textit{city} \rightarrow \textit{new york city}$$

One idea how to filter out these noun phrases would be to exclude noun phrases that contain strings from the respective column as substrings. The *area* and *city* columns of the given examples contain the strings 'East' and 'New York City'. Based on this evidence we could exclude the corresponding noun phrases.

Other noun phrases do not enhance the meaning of a given header string. Examples for this are:

$$\textit{company} \rightarrow \textit{following company}$$
$$\textit{politician} \rightarrow \textit{same politician}$$

Here, an idea would be to develop a list of stop words that contains such words as 'same' and 'following'.

The diagram in figure 6.4 depicts how the correct results outputted by the *Information Extraction Module* are distributed between the operations we defined. It shows that *specification* is by far the most frequent operation performed by the module.

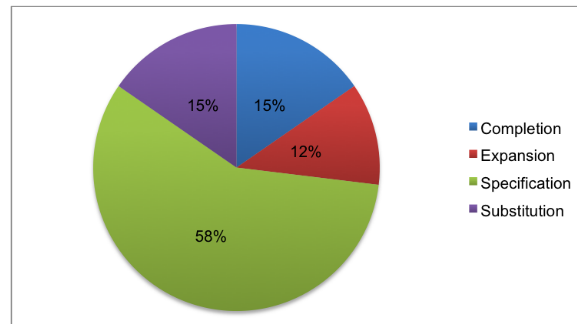

Figure 6.4: Results of the Information Extraction Module

7 FUTURE WORK

Our experiments were only conducted on web tables extracted from Wikipedia. This simplifies the task assigned to the *Information Extraction Module*. Extracting the context of a table from a random web page might not be as simple as it is for Wikipedia pages. The algorithms might need to be adjusted to account for the fact that table caption, headline and the text surrounding the table cannot be extracted as reliably.

The *Information Extraction Module* searches the context for occurrences of the strings in the web table header. This approach could be extended by taking into account the semantic relatedness between words.

We saw in this work that the approach taken in the *Information Extraction Module* has the potential to enhance the tested approach of using a knowledge base for identifying attribute labels. But the evaluation shows that precision is fairly low. As stated, this is because the quality of the results is not assessed by the programs and unwanted results not filtered out. Future work could explore a machine learning approach to the problem by training features that are based on the lexical classes of the words in the retrieved noun phrases. Furthermore the ideas presented in section 6.2.4 could be followed up.

A lot of different scoring functions are involved in our work, i.e. the scoring of classes found in the KB, the scoring of noun phrases found in documents and string similarity scores. When aggregating all the results from the different modules, there are too many factors involved to manually assign weights to the different scores. In order to obtain a ranked list of results that truly reflects their quality, a machine learning approach that learns to apply weights to the scores through training might be more feasible.

8 CONCLUSION

In this paper we developed a two phase process to identify and specify attribute labels.
First, the columns of web tables are linked to classes in a general purpose knowledge base.
Second, information is extracted from the context of the web table. A relation name is identified and attribute labels retrieved by analysing and searching the context of the tables. Experiments show that this novel approach is promising, especially for the identification of relation names. It can enhance the results identified in the first step and provide answers for columns that are not covered by the knowledge base lookup because they do not contain named entities. The quality of the retrieved noun phrases can be improved though. This problem could be explored in future work.

As tables on the web do not contain meta-data and are not necessarily labeled correctly, artificial intelligence is required. We show that an ontology based approach combined with an approach based on information extraction is a promising path to follow.

BIBLIOGRAPHY

[1] Sören Auer, Christian Bizer, Gergi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: A nucleus for a web of open data. *Lecture Notes in Computer Science*, 2007.

[2] Ken Barker and Nadia Cornacchia. Using noun phrase heads to extract document keyphrases. *Advances in Artificial Intelligence*, 2000.

[3] Tim Berners-Lee. Linked data–design issues. *W3C*, 2006.

[4] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 2001.

[5] Dana Budach. Extraktion von kontextinformationen für webtabellen. Bachelor thesis, TU Dresden, 2013.

[6] Michael J Cafarella, Alon Halevy, Daisy Zhe Wang, Eudene Wu, and Yang Zhang. Uncovering the relational web. *WebDb*, 2008.

[7] Michael J Cafarella, Alon Halevy, Daisy Zhe Wang, Eudene Wu, and Yang Zhang. Webtables: Exploring the power of tables on the web. *VLDB*, 2008.

[8] Lee Dice. Measures of the amount of ecologic association between species. *Ecology*, 1945.

[9] Julian Eberius, Maik Thiele, Katrin Braunschweig, and Wolfgang Lehner. Drillbeyond: Open-world sql queries using web tables. *BTW*, 2013.

[10] Georgi Kobilarov, Tom Scott, Yves Raimond, Silver Oliver, Chris Sizemore, Michael Smethurst, Christian Bizera, and Robert Lee. Media meets semantic web, how the bbc uses dbpedia and linked data to make connections. *ESWC*, 2009.

[11] Taesung Lee, Zhongyuan Wang, Haixun Wang, and Seung won Hwang. Attribute extraction and scoring: A probabilistic approach. 2013.

[12] Girija Limaye, Sunita Sarawagi, and Soumen Chakrabarti. Annotating and searching web tables using entities, types and relationships. *VLDB*, 2010.

[13] Gregory Malecha and Ian Smith. Maximum entropy part-of-speech tagging in nltk. *unpublished course-related report (Harvard University)*, 2010.

[14] Varish Mulwad, Tim Finin, Zareen Syed, and Anupam Joshi. Using linked data to interpret tables. *COLD*, 2010.

[15] Fabian M. Suchanek, Gjergji Kasaneci, and Gerhard Weikum. Yago: A core of semantic knowledge unifiying wordnet and wikipedia. *WWW*, 2007.

[16] Peter D. Turney. Learning algorithms for keyphrase extraction. *NPArC*, 2000.

[17] Jingjing Wang, Haixun Wang, Zhonggyuan Wang, and Kenny Q. Zhu. Understanding tables on the web. *Lecture Notes in Computer Science*, 2012.

[18] Simon White. How to strike a match.