University of Stavanger

Tabularized Search Results

Johan Le Gall

supervisor:
Krisztian Balog

June 15, 2016

Abstract

This thesis focuses on the problem of generating search engine results as a table. These are tables describing a list of entities in terms of their features. The goal of making these tables is to make it easier and faster for users to finding information related to their search queries. In resent years we have stated to see changes to the Search Engine Result Pages (SERP). These changes are often centered around presenting entities to the user.

We introduce the term "entity table" to describe such tables, and the term is further explained. The task of making entity tables is formalized, and we identify and explore the challenges that this task introduces. The problem is broken down in to two main task: identifying what rows the table should have, and identifying the most appropriate columns. For the former of these two problems, we explore the task of filtering out inappropriate entities. For the latter, we explore ways of ranking the properties in terms of how well it would fit as a column in the table. Approaches to these task are explained and implemented.

To evaluate our approaches, a gold standard is made for both tasks, listing the ideal output for a set of test queries. An evaluation of the approaches shows us which work the best for our tasks. Lastly, we demonstrate how we can use our implementations to visualise the "entity tables" on a web page.

Contents

Chapter 1

Introduction

Results from search on the web can be showed to the user in several different ways, the most common one being a list of hyperlinks each followed by some text serving as a preview of the document/web page. Although effective document retrieval methods can give us high likelihood of finding the information we are looking for in the first few results, this is often times not the optimal way of presenting the data. What we want is a simple and comprehensive way of showing the data that the user might be interested in given a certain search query. This can be done using a table that list entities and some of their features. This will allow the user to easily look up information and compare the different entities. We call such tables *entity tables*, and creating these tables is the main objective for this thesis.

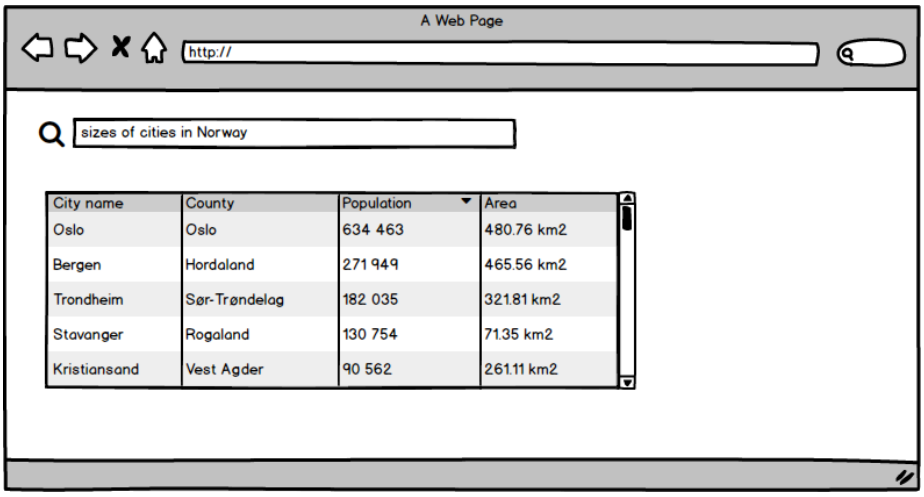


Figure 1.1: An example of search results as an entity table

Achieving our task does present significant challenges. We identify four chal-

lenges: (1) When is it appropriate to present the data in such a way? That is, how can we tell that the information the user is looking for can be presented in a table or how can we tell if the user would benefit from a comparison of entities. One example of when this would be appropriate is for the search query "sizes of cities in Norway". (2) What entity type is the user looking for? For the "Norwegian cities" example the table should show a list of entities with the type "city". It is important that we are able to identify this type, as it is an important part of choosing what entities are included in the table. (3) Which entities should be listed? In our example this would be cities located in Norway. (4) Which properties should be shown? We want to select properties that both answers the query, and summarizes the entities. Relevant attributes for city sizes could be the area or population. These challenges are explored throughout this thesis.

1.1 Contrebutions

The contributions of this thesis are as follows:

- We introduce the task of generating entity tables.
- We propose an approach to solving this task by identifying sub tasks.
- We explore two sub task of our main objective: ranking property, and identifying the target type of a query.
- We propose different approaches to these two tasks.
- We produce a test collection for evaluation these two tasks.
- We evaluate and compare the various approaches to these two sub tasks.
- We demonstrate how an entity table can be shown on a web page.

1.2 Outline

In **Chapter 2**, we give an introduction into the field of information retrieval and some common method of solving information retrieval tasks, as well as discussing previous work that relate to our task. In **Chapter 3** we propose an approach to the task of generating entity tables. Then, we introduce our experimental setup in **Chapter 4**, and the results of these experiments in **Chapter 5**. In **Chapter 6**, we show how these tables can be visualised on a web page. Finally, in **Chapter 7**, we conclude the thesis.

Chapter 2

Background

In this chapter, we give an introduction into the field of Information Retrieval, and discuss some methods used in this field as they relate to our task.

2.1 Information Retrieval

Information Retrieval (IR), a research field concerned with the structure, analysis, organization, storage, searching, and retrieval of information [16], has become of high relevance in the last few decades. Searching through immense collections of data has become an everyday task for us, either with search engines on the web or on smaller domains. An IR task will often consist of finding text documents, although information can also come in other forms, such as images or videos [7].

One central challenge in IR is to retrieve relevant data. Here, the term *relevant* can be ambiguous; a comparison between the language of a search terms and a document might indicate that a document is relevant, even though a user would consider it not to be. Therefore a *retrieval model* must be developed while considering the users' perception of relevancy. Furthermore, the retrieval system must be able to return results that are not only relevant to the query, but also to whatever underlying work task the user is trying to achieve by performing the search. This can be a challenge with limited user input.

2.2 Document Scoring

Many measures for scoring documents relevancy have been proposed. In this section we present some common ones.

2.2.1 Scoring With TF-IDF

We can score a document by the tf and idf of its terms. $tf_{t,d}$ indicates the *term frequency* of term t in document d, and idf_t, the *inverse document frequency* of t tells us how uncommon t is in all documents [7]. Using these two measures, we can calculate how representative a term is of a document like this:

$$tfidf_{t,d} = tf_{t,d} \cdot idf_t \tag{2.1}$$

We can then use this function to calculate the score of a document d in terms of its relevancy to query q, consisting of a set of terms:

$$score_q(d) = \sum_{t \in q} tfidf_{t,d} \cdot tfidf_{t,q} \tag{2.2}$$

2.2.2 Language Modeling

One common way to score entities is in terms of its probability of being relevant to the query by generating a language model of the document. This language model indicate what words are used in the document and how frequent these words are. An example of this is the standard *language modeling* (LM) ranking where $P(q|d)$, the probability that query q is relevant to document d, can be calculated like this:

$$P(q|d) = \prod_{t \in q} P(t|\theta_d) \tag{2.3}$$

where θ_d is a language model of d. we can formulate $P(t|\theta_d)$ like this:

$$P(t|\theta_d) = (1 - \lambda)P(t|d) + \lambda P(t|C) \tag{2.4}$$

Here, $P(t|d)$ indicate how common t is in d, and $P(t|C)$ indicate how common t is in all documents. We use λ to "smooth out" the language model. It is commonly set to 0.1.

2.2.3 BM25

An other common method is BM25, where document scores can be calculated like this:

$$score_q(d) = \sum_{t \in q} \frac{f_{t,d}(1 + k_1)}{f_{t,d} + k_1(1 - b + b\frac{|d|}{avgdl})} idf_t \tag{2.5}$$

where b and k_1 are constants.

2.3 Evaluation

An important part of information retrieval is the evaluation of our algorithms. By performing an IR task and analysing the results, we can determine how good a system is. The obtained measurements can be used to compare the performance of different IR systems. It can allow us to determine if changes to the system actually are improvements. Evaluation is also used for tuning our systems: we can change some of the parameters or parts of the algorithm, evaluate, and see the effects on retrieval performance. In fact, a typical way of working with IR is to first create a baseline system (this can be an existing system, or one we make ourselves) and to make improvements to this system. We can then evaluate and compare the results to the baseline and see the effects of our changes.

There are two ways of performing evaluation: *user testing* and *automated evaluation*. User testing consist of letting a user use the IR application and observe how the system performs in terms of helping the user achieve his/her task. To determine how good the system is we can use data such as where the user is clicking, where the user is looking, or have the user rate the program on certain aspects, such as ease of use or difficulty to learn. This types of testing can tell us a lot about how an application performs in terms of user experience. However, these can be slow and costly, and might not tell us as much about how well the underlying algorithms perform. These are reasons for conducting automated evaluation.[7]

2.3.1 Effectiveness and Efficiency

Most often in IR, our goal is to increase *effectiveness* of our algorithms. Effectiveness is the measure of how well an algorithm performs in terms of retrieving relevant information. To determine effectiveness of an algorithm, we need a way of determining whether the retrieved results are relevant or not with a metric that we can use to compare it to other retrieval systems. We focus less on *efficiency* (how fast the algorithm is) since today's system are usually fast enough once they have been optimized.[7]

2.3.2 Cranfield Paradigm

The most common way of performing evaluation in IR is by using the *Cranfield paradigm* [7]. Here, we have what we call a *test collection*. The test collection consist of a set of topics, a set of documents (a corpus), and relevance judgements for each topic-document pair. A topic represent an information need, which is formulated as a search query. The relevance judgements tell us what documents are relevant for each topic. These are usually binary values (relevant or not relevant) but can also be graded (e.g. an integer value from 0 to 3). These relevance judgments constitute the *ground truth*, which is the ideal ranking that we trying to obtain automatically by using our IR algorithms. Using this test collection we can evaluate how well our IR system performs: we retrieve a set of

documents form the corpus using a query corresponding to a topic, and compare the results against the ground truth. We can then compute the performance of the system on the given query using some formula. This test can be repeated for other queries in the collection to calculate the overall performance of the system and summarize it as a single score.

2.3.3 Precision and Recall

To rate the effectiveness of our system, we can look at how well it performs in terms of the set-based metrics *precision* and *recall*. Precision is the fraction of the retrieved documents that are relevant. The intuition is that we don't want to retrieve irrelevant documents. Recall, on the other hand, is the fraction of relevant documents that are retrieved. Here, the intuition is that we don't want to miss relevant documents, but want to retrieve as many of them as possible. These two metrics are not enough to evaluate an ER system, but can serve as a basis for other evaluation metrics.

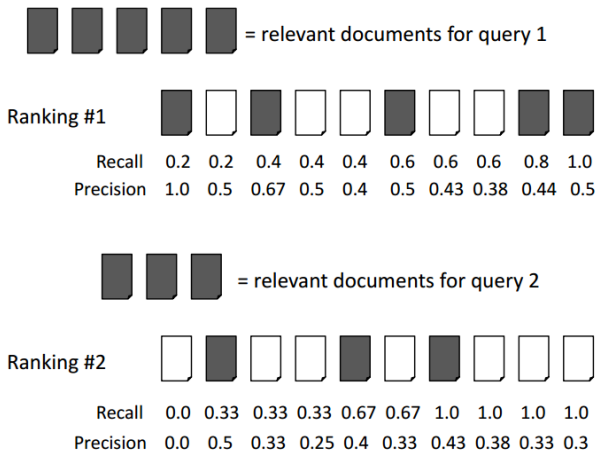


Figure 2.1: Recall and Precision calculated at each document for two queries. Image taken from [7].

2.3.4 Mean Average Precision

Mean Average Precision (MAP) is the most commonly used metric for evaluating an IR system. It is computed by calculating the average precision over each of the points at which we retrieved a relevant document, and taking the mean of this measure for a set of queries. Taking the example rankings from Figure 2.1, MAP can be calculated as follows [7]:

$$AP_1 = (1.0 + 0.67 + 0.5 + 0.44 + 0.5)/5 = 0.62$$
$$AP_2 = (0.5 + 0.4 + 0.43)/3 = 0.44$$
$$MAP = (0.62 + 0.44)/2 = 0.53$$

More generally:

$$MAP = \frac{1}{n}\sum_{i=1}^{n} AP_i, \tag{2.6}$$

where n is the number of queries and AP_i is the average precision for query i. This formula favors results where the relevant documents have high rankings. This corresponds to the intuition that a user wants to get relevant documents at the top of the search results.

2.3.5 Normalized Discounted Cumulative Gain

An other popular metric is *Discounted Cumulative Gain* (DCG). It differs from MAP in that it does not consider relevance to be binary, but can take graded relevance judgments. It measures the IR system based on these two intuitions: (1) the user benefits more from examining a highly relevant document, more so than a less relevant document, and (2) the user benefits from having more relevant documents at a higher rank [7]. We can calculate DCG at rank p like this:

$$DCG_p = rel_1 + \sum_{i=2}^{p} \frac{rel_i}{log_2 i}, \tag{2.7}$$

where rel_i is the relevance of document i as an integer score (for example 0 to 5 where 0 is not relevant and 5 is the highest possible relevancy) [7].

In order to calculate the average DCG over a set of queries, we need to normalize this measure. To do this, we first calculate DCG for the ideal results (iDCG). This is the results where the documents are ordered in descending order of relevance. We can then calculate *Normalized Discounted Cumulative Gain* (NDCG) by dividing DCG of our results by the iDCG: $NDCG_p = \frac{DCG_p}{iDCG_p}$. This gives us a number between 0.0 and 1.0 for any rank cut-off p, as opposed to DCG that can have any number above 0.[7]

2.3.6 Mean Reciprocal Rank

Mean Reciprocal Rank (MRR) indicates how well ranked the first relevant document is on average[7]. It can tell how many documents we would have to inspect from a list of results before we see a relevant document. This is often a good measure when there only is one relevant document. It is calculated like this:

$$MRR = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{rank_i}, \tag{2.8}$$

where $rank_i$ is the rank of the first relevant document for query q_i.

2.4 Entity Retrieval

In the field of *Entity Retrieval* (ER) we are concerned with presenting search results as entities or entity properties as opposed to a list of documents. This comes with some significant challenges, such as (1) how do we represent an entity given a collection of data, and (2) how can we rank these entities based on the user query.

An entity is considered to be an object or thing with the following properties: some unique identifier, name(s), type(s), features describing the entity, and relations to other entities. We can retrieve such entities from a *knowledge base*, such as *DBPedia*[1], where entities are described as a list of features extracted from Wikipedia[2].

The type of an entity is important for categorizing them. These types are often described in an ontology, such as the *DBpedia ontology*[3] where each entity is concidered to be an instace of the type "Thing", but can also have more specific types, such as "Organization", "Person", "Athlete", "Place", "Populated Place", "Settlement" or "Event".

2.4.1 Target Type Identification

As an extension to entity retrieval, we can implement *Target Type Identification* and *type filtering*. This can allow us to only retrieve entities of a certain type. The problem of identifying the target type of a query is the topic of [1]. In this paper Balog and Neumayer [1] explores the problem as it relates to a hierarchical ontology and identify two general models to identifying the most appropriate type from: *type-centric model* and *entity-centric model*.

Type-centric model: One approach is to represent each type as a document and rank each type using common document ranking algorithms. The documents are generated by concatenating the description of all entities associated with that type. The types are then ranked in terms of relevance to a query $q = (w_1, ..., w_{|q|})$ using a language modeling approach:

$$P(q|t) = \prod_{i=1}^{|q|} P(w_i|\theta_t) = \prod_{i=1}^{|q|}((1-\lambda)P(w_i|t) + \lambda P(w_i)) \tag{2.9}$$

[1] `http://wiki.dbpedia.org/`
[2] `https://www.wikipedia.org/`
[3] `http://wiki.dbpedia.org/services-resources/ontology`

Where θ_t is the language model of the document of type t.

Entity-centric model: An other approach to this problem is to retrieve all entities relevant to the query, and then determining the most appropriate type by considering the type of these entities. This is the approach used in [18]. Here, a snapshot of Wikipedia was used as a knowledge base, with entities distributed across 64 types. This was done by first scoring entities based on relevance to the query using a Kullback-Leibler distance based ranking algorithm. They then compute E_q, a list of entities sorted by decreasing scores. The score of a type is then calculated a such (the notation has been changes to fit our own):

$$score_q(t) = \sum_{e \in E_q} \begin{cases} w_q(e), & \text{if } type(e) = t \\ 0, & \text{otherwise.} \end{cases} \tag{2.10}$$

where $w_q(t, i)$ is a weight function. The authors present four weight functions:

- $count_q(e)$, which give equal weight to all entities
- $score_q(e)$, where we use the relevancy score of e for q
- $pos_q(e)$, which considers the rank of the entity. It is calculated like this: $|E_q| - rank_q(e)$, where $rank_q(e)$ is the index of e in E_q when sorted by score
- $pos_q^2(e)$, which is $(pos_q(e))^2$

The results of this paper shows that pos_q^2 gave the best results, and that $count_q$ got the lowest score. This tells us that the target type is more prevalent in the top results since pos_q^2 give higher weight to these results. The authors also experimented with only considering the top 70 results. This gave better results, which further confirms the fact that the target type occurs more often in the top results.

Balog and Neumayer [1] also proposes an entity-centric model. This approach is similator to the one in [18] but with a different entity retrieval method.

2.5 Entity Summarization

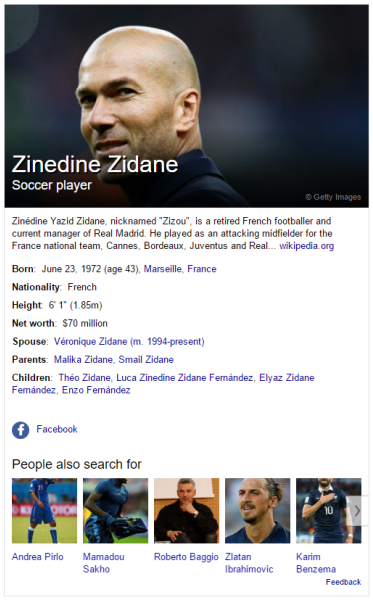
As mentioned before, an entity can be described as list of its features. However, when presenting an entity to the user, such a list can be too long and overwhelming. For this reason, it can be beneficial to present the entity as a short list of features that serves as a summery of the entity.

Cheng et al. [5] define the problem of *entity summarization* as producing a more concise version of an entity, that still contains enough information to determine the underlying entity. Here, a model called RELIN is proposed, which

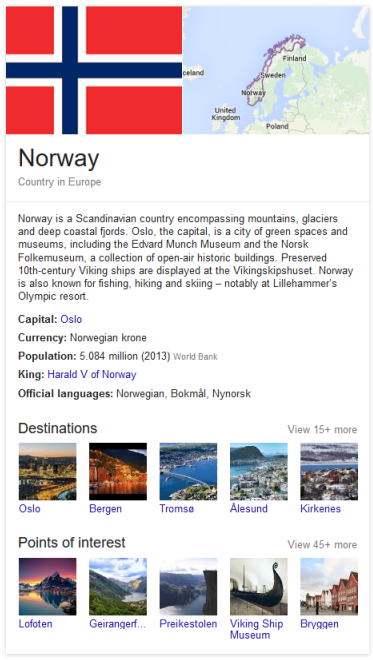
performs entity summarization by ranking an entity's features by a measure of centrality, and choosing the k top-results. This is done using a random surfer model [15].

2.5.1 Entity Summarization on Popular Search Engines

Examples of entity summarization can bee seen on popular search engines such as Google[4] or Yahoo[5] when searching for an entity, such as a person or a country. We are then given a summary of the entity in the form of an info box that can containing pictures, a short description, a list of features including relations to other entities, and other related entities.



(a) An entity summarization from yahoo.com.



(b) An entity summarization from google.com.

Figure 2.2: Entity summarizations on popular search engines.

2.5.2 FACES

In [10], the FACES algorithm is proposed. It performs *diversity-aware entity summarization* by clustering similar attributes into facets using the Cobweb

[4] `https://www.google.com`
[5] `https://www.yahoo.com`

algorithm [9], and summarizing each facet by ranking its attributes. This approach makes it so attributes are chosen based on diversity, popularity, and uniqueness. The input to the Cobweb algorithm is $FS(e)$, the feature set of an entity e. The feature set is the set of all feature (or property-value pairs) that an entity has. The Cobweb algorithm uses the feature set to cluster the features based on semantic similarity. The property-value pairs that describe the features alone are not enough perform the clustering, however. This is why the word set $WS(f)$ is generated for each feature f. The word set contains a set of words describing the feature on a more abstract level, and is useful for clustering similar features. The word set of a feature is generated by taking the words from the property and the type of the value, performing tokenization and stop word removal, and adding hypernyms of the term to the set. A hypernyms of a term is its type (for example a hypernym for "yellow" is "color"). Adding these hypernyms makes the word set a more abstract representation of the feature.

After the word set are generated, the Cobweb algorithm partitions the feature set into facets. Then, the *faceted-entity-summarization* $FSumm(e, k)$ is generated by taking the features that best summarise the facets, where k is the length of the summary. In $FSumm(e, k)$ more then one feature can be chosen from each facet only if k is larger then the number of facets. Otherwise, at most one feature is taken from each facet. To rank the features, a ranking algorithm inspired by tf-idf is utilized. The informativeness $Inf(f)$ of a facet is used as idf, and the popularity of the facets valuer $Po(v)$ is used as tf.

In [10], FACES is evaluated by comparing the results to a set of ideal summaries of length 5 and 10 for 50 randomly selected entities from DBpedia. The ideal summaries were made by 15 judges with experience in Semantic Web. The system was also evaluated based on user evaluation. This evaluation showed that FACES outperformed all entity summarization systems before it in terms of speed and quality of summaries.

2.5.3 LinkSUM

Thalhammer et al. [17] propose another algorithm for entity summarization. It is called LinkSUM and is relevance-oriented as opposed to FACES, which is diversity-oriented. LinkSUM consist of to stages, the first of which being *resource selection*. Here, the goal is to create a ranked list of entities that is connected to the target entity (the entity that we want to summarize). This is done by combining two link-measures: PageRank [4] that accounts for the importance of the connection, and Backlink [19] that tells us rather or not the entities have a strong connection. The two values can then be weighted differently based on parameters of the algorithm.

The links that are considered in this stage are not typed; that is, we don't know what kind of relation the entities have. Often times, an entity will have multiple links to the same entity, and this is the reason for the second stage: *Relation selection*. In order to select the best relation, three factors are defined: *frequency*, *excucivity*, and *description*. Different combinations of these can be

used to rate the relations.

In [17] LinkSUM is evaluated in the same manner as FACES, and it is showed to perform significantly better; both when comparing to the ideal summaries, and by user evaluation.

Chapter 3

Approach

In this chapter, we will look at the underlying system we use for building entity tables. We will not be discussing any challenges related to presenting the data in a GUI, as this is the topic of Chapter 6

In Chapter 1, we looked at the challenges in creating entity tables. Let us formalise these problems:

Problem 1: Entity Retrieval
Ranking entities by their relevancy to the query, and retrieving the top k results.

Problem 2: Query Classification
Identifying rather a user would benefit from seeing an entity table as part of the search results based on the query.

Problem 3: Target Type Identification and Type Filtering
Identifying the appropriate type for the entities in the entity table, and filtering out entities that does not fit this type.

Problem 4: Property Ranking
Providing a ranked list of properties indicating how well it would fit as a column in the entity table.

In this section, we will look at how these problems can be solved using a modular approach, where each module handle one of the above mentioned problems. Figure 3.1 illustrates this approach. We identify four modules in this pipeline: an ER module, a *Query Classification* (QC) module, a *Target Type Identification* (TTI) module, and a *Property Ranking* (PR) module.

The input to the model is the user provided query. The ER module solves Problem 1, and performs an ER task based on this query and returns a list of entities along with their relevancy rating. Then, the QC module determines rather the results should be shown in a table; this handles Problem 2. If the query results from the ER module and the query seem appropriate for an entity

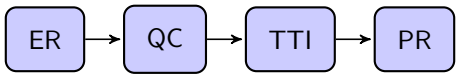


Figure 3.1: Pipeline for presenting entity search results as tables.

table, then these are passed on to the TTI module. The TTI module handles Problem 3 by analysing the query and the entity rankings, and filtering out entities accordingly. Finally, the PR module analyses the filtered list of entities and the query to determine what columns should be seen in the entity table, handling Problem 4.

The main focus of this thesis is the last two modules: the TTI module and the PR module. For our, demonstration we will be using an *oracle* ER module, meaning it has knowledge of the ground truth for a set of test queries and will always give us relevant entities for these test queries. As for the QC module, it is removed from the model, and we consider it future work. This leaves us with a pipeline model as shown in Figure 3.2.

Figure 3.2: Pipeline without query classification.

The problem statement is set accordingly: "Given a query for which the results can be shown as a table of entities, what entities should populate the table (as rows), and which features should be displayed in the table (as columns)?"

In this chapter we will be exploring two research questions:

RQ 1: How can we identify the target type of a query requesting a list of entities, and filter out any entity that does not fit this type?

RQ 2: How can we best select properties that can serve as columns in our entity table?

3.1 Definitions

Let us define the terms and notation used in the following chapters:

Data Set: The data set G consist of a set of entities E_G, a set of properties P_G, a set of values $V_G \in E_G \cup L_G$ where L_G is a set of literals (such as textual or numerical values), and a set of triples $Triples_G$.

Triple: A $triple \equiv (s\ p\ o)$ describe a subject-predicate-object relation where $o \in E_G$, $p \in P_G$, and $o \in V_G$.

Entity: An entity $e \in E_G$ is a uniquely identifiable object. It is described by its feature set $FS(e)$.

Feature: A feature f is a property-value pair. We denote the property of a feature with $Prop(f) \in P_G$, and its value as $Val(f) \in V_G \cup V'$ where $V' \subset V_G$. That is the value of a feature can be a set of values.

Ontology: The ontology O is a hierarchical graph consisting of a set of type nodes T_O and a set of edges A_O indication child-parent relations between types. The ontology has a root type $t_{root} \in T_O$, and any type $t \in T_O$ is a subtype of t_{root}, either by being a child node of t_{root}, or indirectly through one or more nodes.

Entity Types: Each entity has a type $t = type(e) \in T_O$.

In some related literature [10, 17], a feature can have only one value. This is not the case here. For single value features, we can identify the features of an entity as such: a feature f exists in $FS(e)$ if there exists a $triple \equiv (e\ Prop(f)\ Val(f)) \in Triples_G$. We change this so that all triples that share the same subject and predicate become one feature where the value of this feature is a set of values. We allow for multi-value features so that a property is not repeated within an entity for multiple features. Having unique properties for an entity makes it easier to rank properties and determine columns for our entity tables. One can also identify features by looking at triples where the entity is the object of the triple ($f \in FS(e)$ if $triple \equiv (Val(f)\ Prop(f)\ e) \in Triples_G$). This is done in LinkSUM [17], but we avoid this to simplify our task.

3.2 Entity Retrieval

As mentioned, we will be using an oracle ER module. The ER module outputs a set of entities and their relevancy scores $score_q(e)$. We denote the set of entities retrieved for query q with $E_q \subset E_G$. As it is an oracle module, it will never output any entity that is not relevant to the query. We will explain in Section 4.4 how this module is made for our test collection.

3.3 Type Filtering

After retrieving E_q, the set of entities relevant to query q, and their rankings, we still need to filter this list, as all entities still might not fit into the table. In our case we would want to filter the entities by type. This entails: identifying the target type (the type of entities that the user is looking for), and filtering

by this type. In a live application, we would need more filtering than this, as discussed in Section 7.2, however in our case this will be enough considering our oracle ER module.

We can employ different approaches to identify the type that the user is looking for. For this task, we look at the two methods presented in [1], type-centric and entity-centric. We also introduce a *term-centric* approach.

Once we have identified the target type t, we need to filter the entities by this type. With a flat ontology structure, one would simply pick all the entities where $type(e) = t$. But with a hierarchical ontology however, we must include all types that are instance of the target type. In other words, any entity that have a subtype of the target type is also included. We call the output of the TTI module E_q^t, denoting the list of entities retrieved for query q and filtered by type t.

$$E_q^t = \{e | e \text{ “}is\ instance\ of\text{”}\ t\ and\ e \in E_q\} \tag{3.1}$$

We can write the problem of target type identification as such: “for a query q requesting a list of entities and a set of entities E_q retrieved by their relevance to q, identify the target type t that is broad enough for E_q^t to contain all entities $e \in E_q$ that answers query q, and specific enough to filter out as many other entities as possible.”

$$t = TargetType(q) \tag{3.2}$$

The target type is identified by ranking all types in the ontology and selecting the one with the highest rank. The function $score_q(t)$ denotes the rank of type t for query q.

3.3.1 Entity-Centric

Recall from Section 2.4.1 that we can calculate the score of a type by looking at the types of the entities and their relevancy to the query, as demonstrated by Vallet and Zaragoza [18].

$$score_q(t) = \sum_{e \in E_q} \begin{cases} w_q(e), & \text{if } type(e) = t \\ 0, & \text{otherwise.} \end{cases} \tag{3.3}$$

where w_q is one of these weigh functions:

- $count_q(e)$
- $score_q(e)$
- $pos_q(e)$
- $pos_q^2(e)$

The effectiveness of this method relies on the effectiveness the ER method used to gather E_q and their rankings based on $score_q(e)$. In the next chapters we will evaluate this method performes with or oracle ER module.

3.3.2 Type-Centric

In Section 2.4.1 we saw how we can rank types by making a document representation of each type, and ranking these document using a probabilistic language model, as demonstrated by Balog and Neumayer [1]. This method can be implemented with any document retrieval method. We implement it TF-IDF, and BM25. We use Whoosh[1] to perform the document retrieval task.

As proposed in [1], we construct the document representation of each type by concatenation the descriptions of all entities with this type. To improve our results we performer some operations on the documents: we make the document all lowercase, we tokenize to identify all terms in the document by separating them on spaces an any other non alphanumeric character, we remove all stop words (words such as "the", "an" or "for"), and we perform stemming using Porter stemming algorithm provided through the Natural Language Toolkit (NLTK)[2]. We also perform the same operations on the query text before the document retrieval.

With this approach, we can also limit ourselves to considering only the types present in the results or any super type of these. We will see the effects of doing this in Chapter 5

3.3.3 Term-Centric

An other approach is to attempt to identify the type as a term in the query. When studying the queries that request a list of entities, we can see that the type is often specified as a noun in plural form. In these cases we can identify the target type by identifying these nouns. One challenge here is to determine what type in the ontology best represents the type specified in the query. We formalise this problem as such: for a noun n, what type t in our ontology best represents n. We propose two approaches to this problem:

Type Label Comparison: In this approach, we simply compare the terms to the label of the type. That is, the target type is t if $n = label(t)$. Of course, this approach will not always work, as the type will often be specified with a different term then the type label.

Type Label Comparison with Synonyms: To handle the problem with the previous approach, we create a word set $Syn(n) = s_1, ..., s_m$ containing the noun n and its synonyms. This set of synonyms are gathered using WordNet[3] through NLTK. We then compare the synonyms in $Syn(n)$ with the la-

[1] `https://whoosh.readthedocs.io`
[2] `http://www.nltk.org/`
[3] `https://wordnet.princeton.edu/`

bels of the types in the ontology, and identify a set of possible target types $T_n = \{t|\exists t, label(t) = n \; and \; n \in Syn(n)\}$.

Both approaches can identify 0, 1, or multiple possible target types, since there can be multiple terms in plural form in the query, and multiple matching synonyms. In some cases we also need to include terms that are comprised of multiple words. This occurs when a noun in plural has one or more nouns before it in. An example of this is the query "Give me all launch pads operated by NASA.", where we would include the terms "pad" and "launch pad".

For these approaches we rank types like this:

$$score_q(t) = \begin{cases} 1, & \text{if } t \in T_q. \\ 0, & \text{otherwise.} \end{cases} \tag{3.4}$$

where T_q is a set of possible target types as identified using type label comparison with or without synonyms. This approach is not enough to identify a single target type on its own, but it can be used along side other methods to give an indication of what types are more likely to be the target type.

3.4 Property Selection

Property selection is the problem of choosing what properties are the most appropriate to show as columns in our entity tables. These properties should summarise the entities in the table, and in cases where the query asks for a specific property, then these properties should be selected. The method for selecting properties is similar to that of retrieving documents: we first score properties and retrieve the top k ranks. This gives us an ordered list of properties of the desired length.

We start by limiting the properties that we need to consider. We only score the properties that can be found within the set of entities that are included in the table. We call this set of properties $P(E)$ for entity set E. Any property not in $P(E)$ will get a score 0. At this point in the pipeline, the entities that are considered are E_q^t, the entities retrieved for query q and filtered by type t.

3.4.1 Entity Summarization Approach

Our approach is mainly focused on summarizing the entities in the table, and not as much on answering the query directly. This approach consist of first summarizing each entity, and then use these summarizations as the basis for scoring the properties. As discussed in Section 2.5, entity summarization is the problem of ranking features of an entity in terms of how well it summarizes the underlying entity and selecting the top k features to be included in the summary. In our case we can consider the feature ranking task to be the same as property ranking for one entity since all features have unique properties for that entity.

In order to rank properties for the entities in E_q^t, we first calculate the summaries $Summ(e)$ for all entities $e \in E_q^t$. In order to compute these summaries we can use a variety of summarization methods. In this section we propose a summarization/feature ranking method, and we will also see in Chapter 5 how the LinkSUM [17] method performs as a summarization method for property ranking. Once we have the summaries, we employ a voting like method for ranking properties based on the summaries. The ranks of a property for query q can be calculates as such:

$$score_q(p) = \sum_{e \in E_q} w_q(e) w_e(p) \tag{3.5}$$

where w_q and w_e are weight functions. We use the same weight functions as in Section 3.3.1: *count*, *rank*, *pos*, and pos^2. For w_e, we consider the score or position of properties within a summary. That is, $w_e(p)$ is the weight of p relative to $Summ(e)$.

Baseline: As a baseline, we implement this method without any symmetrization method, and in stead, rank all properties of an entity the same. That is $score_e(p) = 1$. We can choose any of the weight functions for w_q. However for w_e, only *count* can be used. This is because the other weight functions consider either the score or the position of the property within the summaries, which are both equal for all properties with our baseline method.

Informativeness-Popularity Ranking: We can rank features similarly to how it is done in FACES [10]. Recall from Section 2.5.2 that this is a tf-idf inspired measure considering $Inf(f)$, the informativeness (or uniqueness) of feature f for all $e \in E_G$, and $Po(v)$, how common the value is in $Triples_G$. In our case, where the goal is to rank properties for a set of entities E, we can introduce two more measures: $Inf_E(f)$ and $Po_E(p)$. $Inf_E(f)$ is the informativeness of the feature f for E or how rare f is in E, while $Po_E(p)$ is the popularity of property p in E, or how common it is for an entity in E to have a feature with $Prop(f) = p$. These four measures are calculated as such:

$$Inf(f) = log(\frac{|E_G|}{|\{e | f \in FS(e)\}|}) \tag{3.6}$$

$$\begin{aligned} Po(v) = log|\{triple\ t | \exists\ e, f : t \in Triples_G\ and \\ t \equiv (e\ Prop(f)\ Val(f))\ and\ Val(f) = v\}| \end{aligned} \tag{3.7}$$

$$Inf_E(f) = log(\frac{|E|}{|\{e | f \in FS(e)\ and\ e \in E\}|}) \tag{3.8}$$

$$\begin{aligned} Po_E(p) = log|\{triple\ t | \exists\ e, f : t \in Triples_G\ and \\ t \equiv (e\ Prop(f)\ Val(f))\ and\ Prop(f) = p \\ and\ e \in E\}| \end{aligned} \tag{3.9}$$

As explained by Gunaratna et al. [10], ranking features based on $Inf(f)$ emphasises more interesting features, while $Po(v)$ makes the summary more human readable. The intuition for $Inf_E(f)$ is that we want columns in out table with varying values, as these will give us a good basis for comparing differences between the entities. $Po_E(p)$ gives higher scores to properties that are common for the entities, avoiding blank spaces in the table. However, this intuition might already be covered by the voting system, as common properties will get more votes from different entities.

We can calculate the rank of feature f for an entity set E using these measures like this:

$$rank_e(f) = Inf(f)^{\lambda_1} \cdot Po(Val(f))^{\lambda_2} \cdot Inf_E(f)^{\lambda_3} \cdot Po_E(Prop(f))^{\lambda_4} \quad (3.10)$$

We can set the λ parameters to put emphasis on different measures. We provide the λ parameters in form of a vector $\Lambda = \{\lambda_1, \lambda_2, \lambda_3, \lambda_4\}$, and we will see in Chapter 5 how different Λ vectors can affect the algorithm. For multi-valued features, we consider one value at a time and average the score for all values of the features. We call this approach the *Informativeness-Popularity* (I-P) ranking.

Chapter 4

Experimental Setup

In this chaper, we present our experimental setup. We describe the data that is used, we describ how we have med the gold standads for our tasks, and we present the experiments for which the results are presented in Chapter 5.

4.1 DBpedia

To achieve our task we will be using DBpedia as our knowledge base. DBpedia is a massive knowledge base consisting of 3 billion RDF triples (pieces of information) describing 38.3 million entities [6]. DBpedia is made from extracting structured data from Wikipedia. This allows DBpedia to cover a wide range of domains while still staying up-to-date, an otherwise costly task. DBpedia has become a central hub in the emerging web of data as an increasing number of data publishers have started to set RDF links from their data sources to DBpedia [6]. We will be using the English version which includes 580 million RDF triples and 4.58 million entities [6].

In DBpedia each entity is described by a list of properties-value pairs. Figure 4.1[1] shows an example of this. Here we see the entity Stranger described by a list of properties in the left column and their values to the right.

To determine the type of an entity we can look at its *rdf:type* property (Figure 4.2). It usually holds a list of types retrieved from various databases. We will be using the ones denoted with *dbo*. These types are from the DBpedia ontology. In our example in Figure 4.2, there are multiple types from this ontology, dbo:Settlement being the most specific one, the others being super types of this type. The DBpedia ontology is generated from Wikipedia info boxes[2]. It is a directed-acyclic graph, and not a hierarchical tree as described in 3.1. However, in our instance of this ontology, there is only one type that brakes the structure of a tree, and we simply remove one of its links to fix this.

[1] `http://dbpedia.org/page/Stavanger`

[2] `http://wiki.dbpedia.org/services-resources/ontology`

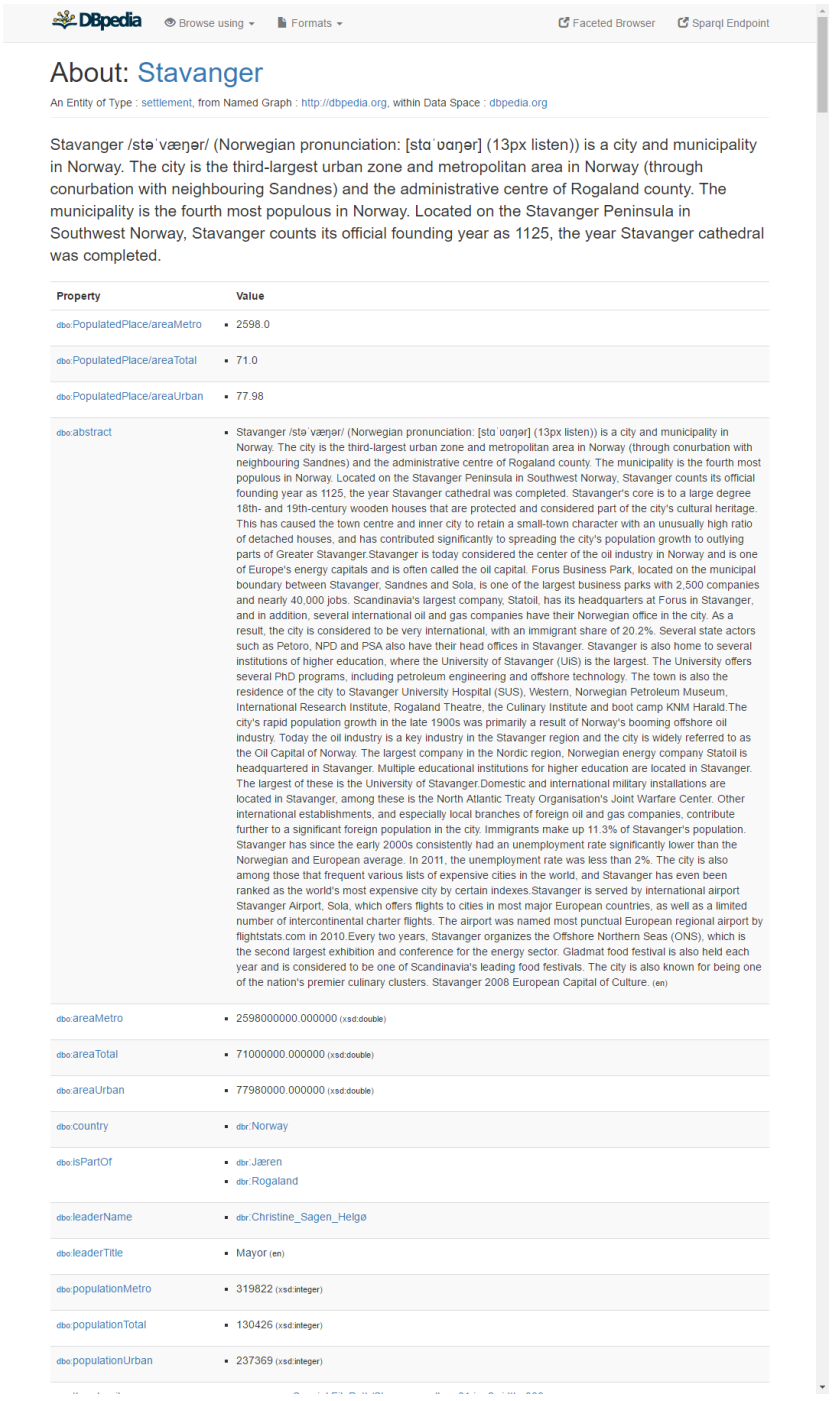


Figure 4.1: The DBpedia page for the Stavanger entity.

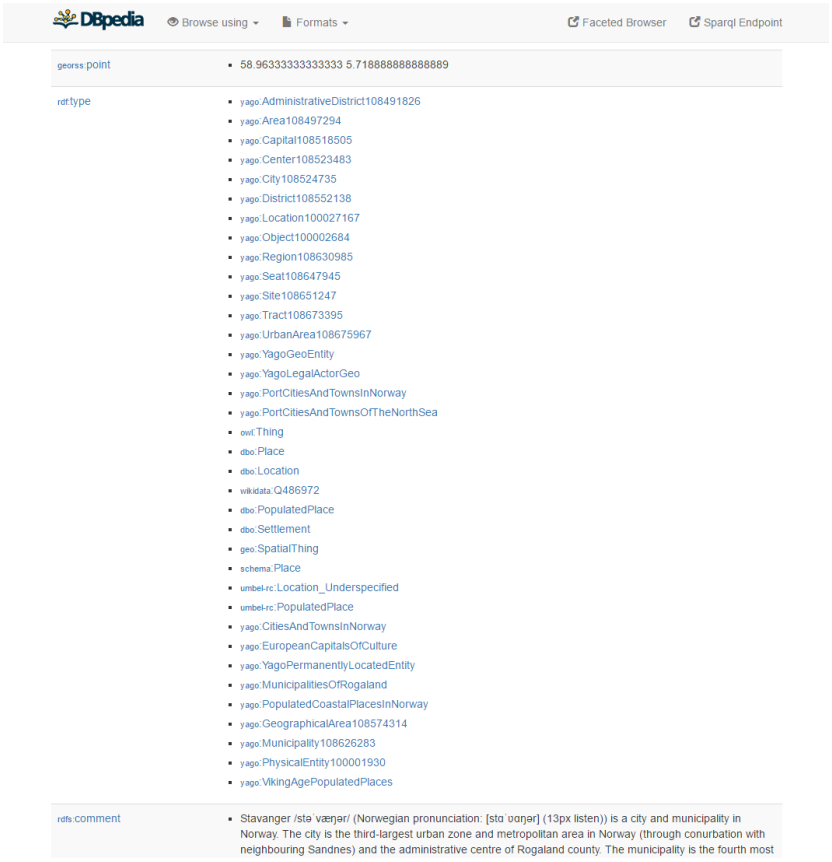


Figure 4.2: The types of Stavanger in DBpedia.

4.2 Test Queries

To test our implementation, a list of queries with relevant results are needed. We will be using the test set presented in [13]. This is a test collection for entity retrieval based on DBpedia. It contains a wide range of queries taken from previous benchmarking evaluation campaigns. These are: INEX-XER [8], TREC Entity [2], SemSearch ES [3, 11], SemSearch LS [3], QALD-2 [14], and INEX-LD [20]. In total this set contains 485 queries. Out of these queries, we select the once where the results can be shown as an entity table. This is done manually by selection the queries that ask for a list of entities. This gives us 163 queries. Here are some examples of such queries:

“List of films from the surrealist category”
“matt berry tv series”
“List all boardgames by GMT.”

"Give me all books written by Danielle Steel."
"major leagues in the united states"
"Chefs with a show on the Food Network."
"famous historical battlefields opponents fought"
"female rock singers"
"wonders of the ancient world"
"Swiss cantons where they speak German"
"In which military conflicts did Lawrence of Arabia participate?"

4.3 Gold Standard

To evaluate our approach, we need a Gold Standard for each of the modules, indicating the ideal output.

For the ER module, we need relevance judgements for each query-entity pair. This ground truth provided to us by [13], where relevant entities get a score of 1, and all others, a score of 0.

4.3.1 Relevance Judgements for Target Types

To evaluate the TTI module, we need to identify the correct target types for each query. This is done by looking at types presented in the ground truth results of the ER task, and picking the correct target type. Recall from Section 3.3 that the correct target type is the most specific type that includes all relevant results. In certain cases, some of the results does not have specific types. As an example of this, all but one of the "Seven Wonders of the Ancient World" are "Things", the root type of the ontology. Therefore, the target type for the query "wonders of the ancient world", is "Thing". This is not ideal, as this would also include all other retrieved entities in the entity table, but for this data set, this is the correct target type.

We employ *strict* as proposed in [1], meaning only the most correct type gets a relevance score 1 and all others 0.

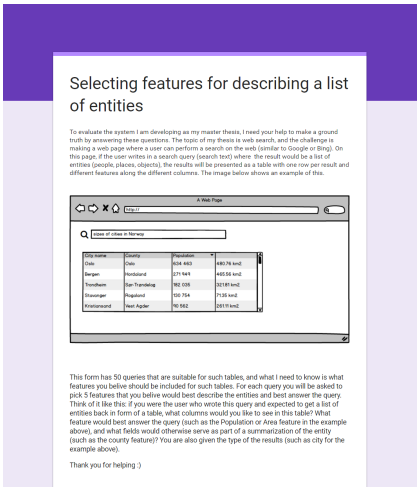
4.3.2 Relevance Judgements for Property Ranking

To evaluate the the results for the choosing property problem, a gold standard was made for 50 queries. These queries were chosen at random from our 163 that are appropriate for showing results as an entity table. 5 people participated in user evaluation, and for each of the 50 queries, they were asked to pick 5 properties that they believed were most appropriate for the query. They were asked to pick properties that were relevant to the query (such as the Population or Area for the example "sizes of cities in Norway") and to pick other features that they believed would be important to include in order to summarize the entities in the table.

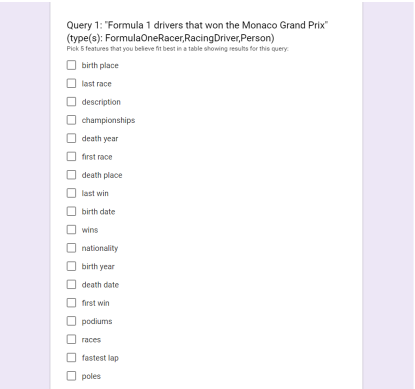
Google Forms[3] was used to to gather these relevance judgements. A form

[3] `https://www.google.com/intl/nb_no/forms/about/`

was generated using Apps Script[4]. This form contained an introduction to make the user familiar with the problem and their task, and a check box question for each of the 50 queries where the user was asked to select 5 features.



(a) Form introduction



(b) Exmple of a query

Figure 4.3: The form used to gather relevance judgments.

The list of properties that the user could select from was constructed by taking the resulting entities for the query, filtering by the appropriate type, and including all the features that occurred in more that 25% of the entities. The last step was included in order to avoid having features that only few of the entities had, this would cause the table to have a lot of blank spaces.

As an example, the the query "tango culture movies" gets these relevancy ratings:
"Country": 3
"Starring": 4
"Music Composer": 2
"Producer": 1
"Release Date": 5
"Language": 4
"Runtime": 2
"Director": 4

[4] `https://developers.google.com/apps-script/`

4.4 Oracle Modules

To evaluate the output of a module, we will provide it with an ideal input. We do this by building oracle modules for the modules before it in the pipeline. These modules operate by accessing the ground truth, and will always output the correct output for any of the test queries.

For evaluating the TTI module, we make an oracle ER module that will always return only entities that are relevant to the query. These are not only the entities that fit into the entity table as a row however, since an entity can be relevant to the query without answering it. Correctly identifying the target type, and filtering the entities by this type should give us a more correct list of entities to serve as row selection.

This approach allows us to test our TTI with an ideal input, but it does present a new problem: the relevance judgments from the ground truth are binary (relevant, or not relevant), and recall from Section 3.3.1 and 3.4.1 that we rely on relevance scores of entities from the ER modules in both the TTI an PR modules. This is solved by scoring the entities in the ground truth using LM scoring on the entities description.

4.5 Evaluation Methods

In this section we present the different evaluation methods we will employ. We will first evaluate the TTI module, then the PR module.

4.5.1 Target Type Identifier Evaluation

We start of by evaluating the different approaches to the TTI modules individually, each with different settings. We have three approaches. These are: entity-centric, type-centric, and term-centric. To evaluate how these approaches perform in terms of identifying the correct type (selecting the top rank), we use the success rate at rank 1 (S@1) and MRR.

4.5.2 Properly Ranker Evaluation

We evaluate the PR module with different summarization methods. these are: the baseline approach, summarization with informativeness-popularity ranking, and LinkSUM [17]. We will look at how they perform with different weighting functions. For the informativeness-popularity ranking approach, we will look at the effect of using different Λ vectors. We first analyse the results using the default setting $\Lambda = \{1, 1, 1, 1\}$ and different combinations of weight functions. We then look at the results when using different Λ, and discuss any notable results.

For LinkSUM, summaries of length 20 are retrieved from their web API[5], providing us with an ordered list of features for each entity. This does not

[5] `http://km.aifb.kit.edu/services/link/`

provide us with the ranks of the feature, and so the *score* function can not be used as a weight function here. These summaries also contains features where the target entity is the object (not the subject). These features appears with a property with this format: "is [Predicate] of". These are removed from the summaries, and all other subsequent features are moved up accordingly. LinkSUM summaries does not contain any literals, only entity values. Therefor, we also remove all properties with literal values from the ground truth when evaluation this method, and from the other methods when comparing them to LinkSUM.

We will be using NDCG@5 and NDCG@10 to evaluate this module. This will indicate how good the systems is as selecting the columns that should be in our entity tables.

Chapter 5

Results

In this section we present the results of our TTI and PR approaches. Recall that we are trying to answer these two research questions:

RQ 1: How can we identify the target type of a query requesting a list of entities, and filter out any entity that does not fit this type?

RQ 2: How can we best select properties that can serve as columns in our entity table?

We compare the different methods used in both, and discuss their benefits and some possible improvements. We will see how the different weight functions affect our results. Recall that the four weight functions are:

- *count*: all have equal weight
- *score*: weight equal the score
- *pos*: weighting based on rank
- pos^2: the square of *pos*

5.1 Target Type Identification

We have three different approaches to identifying the target type: the entity-centric approach, the type-centric approach, and the term-centric approach.

5.1.1 Entity-Centric

For the entity-centric approach we can use the weight functions for the entities (w_q):

w_q	S@1	MRR
count	**0.7607**	**0.8176**
score	0.7361	0.8049
pos	0.7423	0.8088
pos^2	0.7484	0.8109

Table 5.1: Results for the entity-centric approach.

count give us the best result here. The S@1 results tell us that this system is able to identify the target type for 76% of the cases, and the MRR result indicate that the target has an average rank of $0.8176^{-1} = 1.2230$. These result are not bad, but it should be noted that these results rely heavily on the effectiveness of the ER module, and in our case, we use the oracle ER module.

5.1.2 Type-Centric

The type-centric approach, consisting of using document retrieval method on document representations of the types. It can be done with any document retrieval method. We use the methods TF-IDF and BM25.

retrieval method	S@1	MRR
TF-IDF	**0.2944**	**0.4463**
BM25	0.2331	0.3885

Table 5.2: Results for the type-centric approach.

TF-IDF performs the best here with a sucsess rate of 29%, and an average rank for the target type of $0.4463^{-1} = 2.2406$. We can improve our results by only considering the types that are present in the entities:

retrieval method	S@1	MRR
TF-IDF	**0.6748**	**0.7018**
BM25	0.6625	0.6821

Table 5.3: Results for type-centric approach when considering only types present in E_q.

Again we are relying on the ER module, and as expected we get significantly better results.

5.1.3 Term-Centric

Recall that the term-centric approach consists of identifying possible terms in the query that could refer to the target type, and identify possible types that would represent this word. Since this approach can give us no, one, or multiple types, it is not enough to select *one* target type by its self, and we evaluate this method in terms of how well it can improve the results of the entity-centric method. To do this we give different weights to the score using the entity-centric approach, $entityScore_q(t)$, and the score using the term-centric approach, $termScore_q(t)$ using this equation:

$$score_q(t) = entityScore_q(t) \cdot (1 - \lambda) + termScore_q(t) \cdot \lambda \tag{5.1}$$

We use $\lambda = 0.5$, meaning we all types with $termScore_q(t) = 1$ are ranked higher than those with $termScore_q(t) = 0$. We first look at the results for type label comparison without synonyms:

w_q	S@1	MRR
count	0.7546	0.8166
score	0.7361	0.8080
pos	0.**7730**	**0.8285**
pos^2	0.7668	0.8241

Table 5.4: Results for entity-centric approach using type label comparison ($\lambda = 0.5$).

When comparing to the results in 5.4, we see that we can get better results using this approach. Although, in some cases, the results are lower. This is caused by the terms-centric approach identifying the wrong types in some cases.

Next, we look at the terms-centric approach using synonyms. From Table 5.5, we can tell that this approach give us worst results. Again, this is because of wrong types being accosted with the terms, and this case, more types are identified by with the terms-centric approach .

w_q	S@1	MRR
count	**0.7484**	0.8134
score	0.7361	0.8080
pos	**0.7484**	**0.8145**
pos^2	0.7361	0.8069

Table 5.5: Results for entity-centric approach using type label comparison with synonyms ($\lambda = 0.5$).

5.1.4 Discussion

These results show that with a strong ER module, we can identify the target type with a relatively high success rate, using simple methods. We also saw that we are able to identify the target type as a term in the query. We saw that we need to be conservative when trying associating these terms with types, which only gives us minor improvements. To really make use of such an approach, we would need a better way to associate terms to types. One example where this approach does not work is for the query "Which U.S. states are in the same timezone as Utah?". Here we see that the target type is "State", but the correct type here would be "Administrative Region", as this is the type of U.S. states in DBpedia. Such problems could be solved by linking all types in the ontology to the corresponding term in WordNet. This would also allow us to further utilize WordNet to score target types, for example by calculating relatedness between the therms from the queries and the terms representing the types.

5.2 Property Ranking

Next, we look at the results for the property ranking task. We start of by looking at the baseline, then using the informativeness-popularity ranking, and finally with LinkSUM.

5.2.1 Baselines

Table 5.6 shows the results for our baseline method using different weight function for w_q. Here we can see that $w_q = pos$ performs the best, indication that the preferred properties are more present in the more relevant entities. Both *score* and pos^2 perform worst than *pos*. These functions emphasises the top results even more, favoring the properties present in these entities as opposed to a more even weighting across the entities, and selects rarer properties.

	with literals		without literals	
w_q	NDCG@5	NDCG@10	NDCG@5	NDCG@10
count	0.2443	0.3648	0.4848	**0.5525**
score	0.2444	0.3647	0.4844	0.5492
pos	**0.2556**	**0.3751**	**0.4892**	0.5500
pos^2	0.2491	0.3674	0.4824	0.5503

Table 5.6: Results for our baseline approach.

5.2.2 Informativeness-Popularity Ranking

Table 5.7 show the results for the default setting ($\Lambda = \{1, 1, 1, 1\}$). Here, it seems *pos* performs the best for w_e when including literals. Without literals, it

seems $w_e = pos^2$ give us the best results.

	with literals		without literals	
w_q	NDCG@5	NDCG@10	NDCG@5	NDCG@10
count	0.2696	0.3455	0.4199	0.5082
score	0.2659	**0.3478**	0.4183	0.5139
pos	**0.2728**	0.3462	0.4293	0.5136
pos^2	0.2557	0.3365	**0.4330**	**0.5224**

(a) $w_e = score$

	with literals		without literals	
w_q	NDCG@5	NDCG@10	NDCG@5	NDCG@10
count	**0.2921**	**0.3985**	0.4488	0.5389
score	0.2885	0.3970	0.4510	0.5386
pos	0.2886	0.3940	0.4668	0.5470
pos^2	0.2803	0.3846	**0.4683**	**0.5567**

(b) $w_e = pos$

	with literals		without literals	
w_q	NDCG@5	NDCG@10	NDCG@5	NDCG@10
count	**0.2936**	0.3888	0.4394	0.5405
score	0.2823	**0.3895**	0.4437	0.5469
pos	0.2773	0.3732	**0.4553**	**0.5508**
pos^2	0.2682	0.3662	0.4516	0.5488

(c) $w_e = pos^2$

Table 5.7: Results for property ranking using I/P with $\Lambda = \{1, 1, 1, 1\}$. $w_e =$ *count* is not included since it functions the same as the baseline method.

To look at the effects of each of the different measures, we use appropriate Λ vectors to only use one measure at a time. Table 5.8 show us that $Po(v)$ gives the best results when only using one measure with literals included. We see that without literals included, $Inf(f)$ has the highest effect.

To optimize these parameters we would use a learning algorithm, something we consider future work, and a possible improvement to our system. Table 5.9 shows a few more Λ vectors, and their effect.

5.2.3 Property Ranking with LinkSUM Summaries

Looking at Table 5.10, we see that this approach performs best with the weight function $w_q = count$ and $w_e = pos$ for $k = 5$, and $w_q = score$ and $w_e = count$ for $k = 10$. It perform significantly better than the other methods, indicating that it select properties that fit well as columns.

	with literals		without literals	
Λ	NDCG@5	NDCG@10	NDCG@5	NDCG@10
$\{1, 0, 0, 0\}$	0.2560	0.3698	0.4888	**0.5786**
$\{0, 1, 0, 0\}$	**0.3148**	**0.3917**	0.4888	0.5536
$\{0, 0, 1, 0\}$	0.2584	0.3698	0.4888	0.5578
$\{0, 0, 0, 1\}$	0.2679	0.3746	0.4888	0.5536

Table 5.8: Results for property ranking with only one of the four measures at a time. The results are maximum score for all combinations of weight functions.

	with literals		without literals	
Λ	NDCG@5	NDCG@10	NDCG@5	NDCG@10
$\{0, 1, 0, 1\}$	0,3131	0,4105	0,4848	0,5525
$\{1, 2, 1, 2\}$	**0,3315**	**0,4229**	0,4848	0,5567
$\{1, 0, 1, 0\}$	0,2444	0,3676	0,4848	**0,5638**
$\{2, 1, 2, 1\}$	0,3254	0,4115	0,4848	0,5525

Table 5.9: Results for property ranking using I/P with various Λ vectors. The results are maximum score for all combinations of weight functions.

5.3 Discussion

As we can tell from the results, It seems that in our attempt at using summarization techniques for property ranking, LinkSUM perform the best. Reacall thet LinkSUM empesises values with a high PageRank and a strong link to the entity. Our I-P method seem to be laking some optimization, and still has some improvement potensial. It also has the advantage of being able to rank literals as values.

One improvement that could be done would be to differentiate between how we rank literals and entity values. Gunaratna et al. [10], the creators of the faces algorithm have proposed an approach to this[1]. We could also experiment with utilizing facets as it used in FACES.

[1] `http://wiki.knoesis.org/index.php/FACES#Ranking_.28features_.26_facets.29`

	without literals	
w_q	NDCG@5	NDCG@10
count	0.5150	0.5927
score	**0.5171**	**0.5972**
pos	0.5078	0.5898
pos^2	0.5031	0.5890

(a) $w_e = count$

	without literals	
w_q	NDCG@5	NDCG@10
count	**0.5212**	**0.5934**
score	0.5092	0.5837
pos	0.5018	0.5801
pos^2	0.5024	0.5809

(b) $w_e = pos$

	without literals	
w_q	NDCG@5	NDCG@10
count	**0.5121**	**0.5853**
score	0.4943	0.5773
pos	0.4935	0.5765
pos^2	0.4923	0.5759

(c) $w_e = pos^2$

Table 5.10: Results for property ranking using LinkSUM.

Chapter 6

Result Presentation

To demonstrate what entity tables look like when being generated by our algorithm, we develop a web page. Once the query classification task has been achieved, this web page could be expended to show a regular list of search results and include an entity table when this is appropriate. We start of with an induction to the framework use to develop the site, and then look at how the site is built.

6.1 Django

The web page has been developed using Django, a "high-level Python web framework that encourages rapid development and clean, pragmatic design"[1]. Django is based around the *Model-View-Controller* (MVC) ref paradigm, although it is often referred to as an MTV (Model-Template-View) framework, as these are therms more commonly used for Django [12].

The view in MTV has a similar role to the controller in MVC: when a user requests a web page from the server using a URL, a view will generate the response. We can set different views to be called for different URL paths, and the view can access parameters from the URL when generating the reply. A view will reply to the user in the form of a template. A template is what is referred to as a view in the MVC model. This is essentially the HTML document along with styling and scripts, and indicates what the user will see in their browser. In MTV, the model refers to the database access part of the framework. Django is well suited for *database-driven web sites*, where the web site is essentially a way for a user to access a database. For instance, a database resource can be specified by the user as a URL path, and the view can reply with a template describing the resource. [12]

[1] `https://www.djangoproject.com/`

6.2 Building our Web Page

Our task when building the web page is relatively simple: we want the user to be able to specify a query, then have an entity table generated and shown to the user. To achieve this, we will provide the query to a view as a URL parameter, and have it give the query as an input to an implementation of the pipeline as described in Chapter 3. This will give us a list of ranked entities and a list of ranked properties, which we use to generate the template. We will need to format the different types of values to fit as a cell in the table. We also want to include a way for the user to query the server using a text field. Using this text field should generate the appropriate URL and request a new page.

6.2.1 Recognizing URL Patterns

First, we make an application in our domain called "search". This application will handle any URL path starting with "`/search`", and any request matching this pattern will be handled by the "index" view function of this sub application. A query can be given as a parameter to the view like this: "`/search/?query=EU%20countries`"

6.2.2 Generating the Table Data

To make the table, we first need to initialise the pipeline:

```
# make the pipeline:
er = OracleER()
tti = OracleTTI()
pr = IPPR()
pipeline = Pipeline(er, tti, pr)
```

Listing 6.1: Initializing the pipeline.

Our pipeline uses an instace of the `OracleER` class. Of course, this module will only work for the test queries, as these are the only queries it has the ground truth for, but we could easily swap out this module with an actual ER system once it has been developed. We use the pipeline in the view function to retrieve the required data.

```
# get the entity and property scores for the query
entity_scores, property_scores = pipeline.make_table(query)
```

Listing 6.2: Scoring entities and properties using the pipeline.

At this point, we sort the entities and properties by their score, and select the top ranks. We set the cut of point for entities at 10, and the cut for properties at 5. We then pass on all needed information onto the template as the *context*.

6.2.3 Generating the Template

We generate the entity table in as an HTML table. This is done using Django Template Language (DTL)[2], which allows us to generate HTML dynamically. Listing 6.3 shows how this is done. Variables such as `entities` are retrieved from the context provided by the view. We use Bootstrap[3] for styling.

```
<h1>
    {{ query }}
</h1>
<table class="table table-striped">
    <thead>
    <tr>
        <th></th>
        {% for property in property_labels  %}
        <th>{{ property }}</th>
        {% endfor %}
    </tr>
    </thead>
    <tbody>
        {% for entity in entities  %}
         <tr>
            <td>{{ entity.display_name }}</td>
            {% for property in properties  %}
            <td>{{ entity|get_item:property }}</td>
            {% endfor %}
         </tr>
        {% endfor %}
    </tbody>
</table>
```

Listing 6.3: Generating the entity table wit DTL

6.2.4 Formatting Values

Features can have different types of values. We can format these values in different ways within the cells. For entity values, we simply show the label of the entity. Recall that in DBpedia any entity relates to a Wikipedia page, and we can make these label into hyperlinks to the corresponding Wikipedia pages, making it easy for the user to quickly get more information about the entity.

For literals, we simply show the values. Some more work could be done here to indicate the appropriate unit for numerical values. These units can be found in the DBpedia ontology.

For multi-valued features, we simply choose the first one in the list. We do not possess any indication as to what value is the best one to select here. One possibility here is to rank the features as single valued only, and choosing the best rank.

[2] `https://docs.djangoproject.com/en/1.9/topics/templates/`
[3] `http://getbootstrap.com/`

EU countries

	capital	language	official language	currency	government type
Portugal	Lisbon	Portuguese language	Portuguese language	Euro	Unitary state
Finland	Helsinki	Finnish language	Finnish language	Euro	Unitary state
Poland	Warsaw	Polish language		Polish złoty	Parliamentary republic
Latvia	Riga	Latvian language	Latvian language	Euro	Parliamentary republic
Estonia	Tallinn	Estonian language	Estonian language	Euro	Parliamentary republic
Republic of Ireland	Dublin	Irish language		Euro	Unitary state
Slovenia	Ljubljana	Slovene language	Slovene language	Euro	Unitary state
Lithuania	Vilnius	Lithuanian language	Lithuanian language	Euro	Unitary state
Hungary	Budapest	Hungarian language	Hungarian language	Hungarian forint	Unitary state
Sweden	Stockholm	Swedish language	Swedish language	Swedish krona	Unitary state

Figure 6.1: An example of an entity table

6.2.5 Sending Queries to the Server

At this point, the user can send a query by specifying it in the URL. This isn't the most user friendly solution. What we want is a way for the user to write query in a search field on the web page. As mentioned, the system only work for the test queries at this point, and so we use a drop down box where the user can select one of the 163 test queries in stead of a text field. This drop down could easily be replaced with a text field if a fully functioning ER module was used.

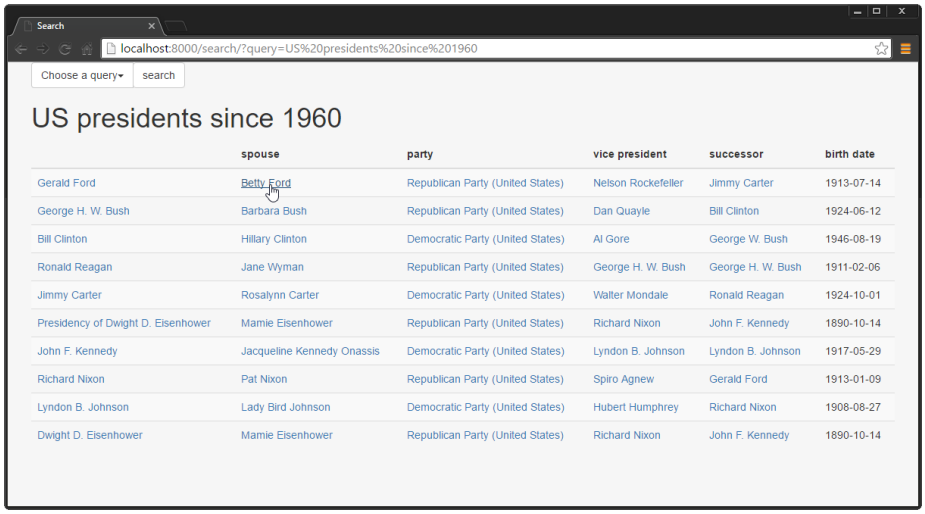


Figure 6.2: A sample of the web page.

Chapter 7

Conclusion and Future Work

In this final chapter, we give a short summary of the report and what we have done. We than discuss future work as well as some possible improvements.

7.1 Conclusion

We started of the report by introducing the task of generating entity tables and identify some of the challenges related to this task. In Chapter 3, we formalized these problems, and presented two research questions that we wanted to answer. We take a retrospective look at these questions and discuss them:

RQ 1: How can we identify the target type of a query requesting a list of entities, and filter out any entity that does not fit this type?

Having experimented with three different approach, it seem they all show potential, although it has yet to be seen how these will work with a live entity retrieval system. We introduced a simple term-centric approach and showed how we can identify the type as a noun in the query; however, this gave us little in terms of results as it was perhaps a bit to simple, and the approach would need to be explored further.

RQ 2: How can we best select properties that can serve as columns in our entity table?

When evaluating up against relevance judgements gathered from a group of five users, we showed that a summarization based approach to this problem can work. We saw that LinkSUM was the best summarization method for this, and that the informativeness-popularity ranking approach has potential for improvement given some fine tuning of the parameters.

We also demonstrated how our tables can be illustrated on a web page using the Django framework, giving us an idea of how entity tables could look on a search engine result page.

7.2 Future Work and Possible Improvements

We are well on our way to achieving our goal of building a search engine that presents entity tables when appropriate, but there still are some parts missing. Most notable, is the absents of the query classification module, which would determine when entity table are suited.

We are also missing a working ER system, and even though our extension to our ER module, the target type identifier, has a good success rate, this is largely due to us using such a strong ER module in our experiments. In Section 3.3.2, we showed an attempt at using NLTK to identify the target type. However, such an approach could be more viable if we were to identify the term in WordNet that best represent the types in the DBpedia ontology. This would make us able to utilise NLTK much better.

For the target type identification task, we could also make use of the hierarchical structure in of our ontology. Perhaps a type should get a better score if its subtypes have good scores?

One important aspect of building a working ER system for building our entity tables would be to identify the *target feature*. This entails identifying what is the feature that the entities in the table should have in common. For instance, for the query "Which countries in the European Union adopted the Euro?", we would be looking for entities with the feature "Currency: euro".

One other possible improvement to our system would be to give higher rank to properties that the user specify in the query. This would include the task of identifying *target properties*. We could then sort the entities in the tables by this property, as the user likely would want to compare the values of its column.

Bibliography

[1] Krisztian Balog and Robert Neumayer. Hierarchical target type identification for entity-oriented queries. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 2391–2394. ACM, 2012.

[2] Krisztian Balog, Pavel Serdyukov, and Arjen P de Vries. Overview of the trec 2010 entity track. In *Proceedings of the Test Retrieval Conference*, 2010.

[3] Roi Blanco, Harry Halpin, Daniel M Herzig, Peter Mika, Jeffrey Pound, Henry S Thompson, and T Tran Duc. Entity search evaluation over structured web data. In *Proceedings of the 1st international workshop on entity-oriented search workshop (SIGIR 2011), ACM, New York*, 2011.

[4] Sergey Brin and Lawrence Page. Reprint of: The anatomy of a large-scale hypertextual web search engine. *Computer networks*, 56(18):3825–3833, 2012.

[5] Gong Cheng, Thanh Tran, and Yuzhong Qu. RELIN: Relatedness and Informativeness-based Centrality for Entity Summarization. In *Proceedings of the 10th International Conference on The Semantic Web - Volume Part I*, ISWC'11, pages 114–129. Springer-Verlag, 2011.

[6] Georgi Kobilarova Sören Auerb Christian Beckera Richard Cyganiakc Sebastian Hellmannb Christian Bizera, Jens Lehmannb. DBpedia - A crystallization point for the Web of Data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7:154–165, 2009.

[7] W Bruce Croft, Donald Metzler, and Trevor Strohman. *Search engines: Information retrieval in practice*. Pearson Education, Inc., 2015.

[8] Gianluca Demartini, Tereza Iofciu, and Arjen P De Vries. Overview of the inex 2009 entity ranking track. In *Focused Retrieval and Evaluation*, pages 254–264. Springer, 2009.

[9] Douglas H Fisher. Knowledge acquisition via incremental conceptual clustering. *Machine learning*, 2(2):139–172, 1987.

[10] Kalpa Gunaratna, Krishnaprasad Thirunarayan, and Amit Sheth. FACES: Diversity-aware Entity Summarization Using Incremental Hierarchical Conceptual Clustering. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, number 3 in AAAI'15, pages 116–122. AAAI Press, 2015.

[11] Harry Halpin, Daniel M Herzig, Peter Mika, Roi Blanco, Jeffrey Pound, Henry S Thompson, and Duc Thanh Tran. Evaluating ad-hoc object retrieval. In *Proc. of the Intl. Workshop on Evaluation of Semantic Technologies*, 2010.

[12] Adrian Holovaty and Jacob Kaplan-Moss. *The definitive guide to Django: Web development done right*. Apress, 2009.

[13] Robert Neumayer Krisztian Balog. A test collection for entity search in DBpedia. *36th international ACM SIGIR conference on Research and development in Information Retrieval (SIGIR 2013)*, page 737–740, 2013.

[14] Vanessa Lopez, Christina Unger, Philipp Cimiano, and Enrico Motta. Evaluating question answering over linked data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 21:3–13, 2013.

[15] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[16] Gerard Salton. *Automatic information organization and retrieval*. McGraw-Hill, 1968.

[17] Andreas Thalhammer, Nelia Lasierra, and Achim Rettinger. Linksum: Using link analysis to summarize entity data. In *Proceedings of the 16th International Conference on Web Engineering (ICWE 2016). To appear*, 2016.

[18] David Vallet and Hugo Zaragoza. Inferring the most important types of a query: a semantic approach. In *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 857–858. ACM, 2008.

[19] Jörg Waitelonis and Harald Sack. Towards exploratory video search using linked data. *Multimedia Tools and Applications*, 59(2):645–672, 2012.

[20] Qiuyue Wang, Jaap Kamps, Georgina Ramirez Camps, Maarten Marx, Anne Schuth, Martin Theobald, Sairam Gurajada, and Arunav Mishra. Overview of the inex 2012 linked data track. In *CLEF (Online Working Notes/Labs/Workshop)*, 2012.