From tag to concept

Ketelaars, G.J.A.M.

Award date:
2008

Eindhoven University of Technology
Department of Mathematics and Computer Science

From Tag to Concept

By
G.J.A.M. Ketelaars

Supervisors:

prof.dr.ir. G.J.P.M. Houben
ir. K.A.M. van der Sluijs

Eindhoven, February 2008

Abstract

Semantic Web technologies help machines 'understand' and easily share more information on the Web. It provides promising approaches which enable us to categorize and annotate multimedia objects. Although Semantic Web comes with nice technologies and useful tools for structuring and annotating information, it can be difficult to the average user. Tags, on the other hand, are a simple form of annotation and are used easily to categorize objects.

This research examines the possibility to combine tagging and existing ontologies in such a way that the simplicity of tagging is preserved and the expressiveness of ontologies is obtained. Mapping tags to concepts will allow us to exploit the semantics of ontologies. The main goal of this project is to find high-quality relations between tags and concepts in Semantic Web ontologies.

For this purpose the TagConceptualizer is designed. This framework extends tag information with concept or tag suggestions that are used by other applications. The TagConceptualizer is tested with the GTAA ontology as an information source. The results show that the designed framework produces good tag suggestions. This implies that high-quality relations between tags and concepts are made.

Preface

This thesis concludes my studies at the Department of Mathematics and Computer Science at Eindhoven University of Technology. My project was carried out in the Business Information Systems area of expertise.

I would like to thank my supervisors, Geert-Jan Houben and Kees van der Sluijs, for their support and valuable input. I also would like to thank the other member of the assessment committee, Ad Aerts, for reviewing my work.

Further I would like to thank my fellow graduation students for the pleasant work environment and interesting discussions. And of course a word of thanks goes to my family and friends for their encouragements and belief in a good ending of my project.

Finally, I would like to thank my girlfriend Joyce for all her love, motivation and support.

Gertjan Ketelaars

February 2008

Table of contents

Section 1: Introduction

Nowadays, a lot of information is available on the web. The number of data-driven web applications is growing, which results in even more information. An increasing part of this information are multimedia objects. Unfortunately a lot of these multimedia objects are not equipped with metadata. The metadata is necessary to index the information to enable browsing and searching through multimedia collections. Therefore, metadata has to be created.

To make the multimedia collections available to users, they must be annotated. This is typically a manual process that is done by professionals. However, there are few professionals and a lot of data. There is too much work for the available time. A solution to that, would be to let users participate in the annotating process. Semantic Web technologies are promising approaches which enable us to categorize and annotate multimedia objects.

1.1 Semantic Web

Semantic Web enables us to help machines 'understand' and easily share more information on the Web. In [Berners Lee 2001] Tim Berners Lee describes how meaningful content of web pages can be structured and used by software agents which can carry out sophisticated tasks for users. For this, new technologies and data modeling techniques are necessary.

Currently different data modeling techniques are already developed. The Resource Description Framework (RDF) is a metadata model which can be used to represent and exchange information. RDF is a directed graph structure that enables making statements about resources in the form of triples (subject, predicate, object). The subject denotes the resource which can be information on the web, real persons or other metadata. The predicate denotes the relation between the subject and the object. In principle, the relations can express anything about anything. For example, one way to represent that Vincent van Gogh painted the painting 'Sunflowers' is a triple with the subject 'Vincent van Gogh', the predicate 'hasPainted' and the object 'Sunflowers'. This mechanism for describing resources is convenient for annotating multimedia collections.

RDF can be used through several syntax formats, however, XML is commonly used. The subject or resources are denoted as a Uniform Resource Identifier (URI). This makes them identifiable and exchangeble, although it is possible to have anonymous subjects: blank nodes. The predicates are resources that represent a relationship. Objects are also resources, or literals (strings).

Ontologies build upon RDF. They are used to represent semantical relations between concepts within a domain and group concepts into classes. Ontologies include constructions to create semantical relations between concepts or groups of concepts. For example the relation equivalence. These semantical relations are based on Description Logics and allow reasoning and inferencing. That is, the ontological constructions have logical implications that support processing of the relations and resources. For example consider the previous example of Vincent van Gogh. Suppose we add the triples <'hasPainted', 'domain', 'Person'> and <'hasPainted', 'range', 'Painting'>. Because the concept 'Sunflowers' has the property 'hasPainted', it inferences that it is a painting. Another inference that can be made is that 'Vincent van Gogh' is a 'Person'.

Although Semantic Web provides us with nice technologies and useful tools, the use of visual graph based tools for structuring and annotating information could be difficult to the average user. The major drawbacks are the need for knowledge of the tools and technologies, as well as the correct interpretation of concepts in ontologies [Bateman 2006]. Therefore, there is a need for an easier alternative.

1.2 Tagging

An easy approach to add metadata is tagging. Our interpretation of a tag is a short textual user input, that describes an object, whatever that object may be. Tags are a simple form of annotation and can be used to categorize objects. An object can have multiple tags to describe it more precisely. A key advantage of tags is that they can be freely chosen. Moreover, a lot of websites like Flickr[1] and Delicious[2] already support tagging. Many people are already used to the concept of tagging.

One of our assumptions is that if the same tag is assigned to an object numerous times by multiple users, the chance is greater that the tag is a good description of the object. These tags can be used to categorize objects, which improves searching and navigation. Examples of collaborative tagging applications are the famous Flickr for photo sharing, and Delicious as a social bookmarking tool. Another example is the ESP game[3], which is a nice example of finding ways to motivate people to tag images and therefore improve the quality of image searching.

The power of tagging is its simplicity, it is easy to do. However, tagging lacks an important feature of the semantic Web: it is impossible to express relationships between resources and annotations. This simplicity results in less richer navigation and searching possibilities. For example, if a painting is tagged with 'Sunflowers', and someone is looking for paintings of Vincent van Gogh but does not know that one of his works is called Sunflowers, it is unlikely that he or she will find the painting with the tag 'Sunflowers'. Tags have no semantics or metadata to derive such relations as in the previous examples of van Gogh.

[1] See http://www.flickr.com/

[2] See http://del.icio.us/

[3] The ESP Game can be found at http://www.espgame.org/. It is a two-player game where points are earned if the players come up with the same tags for an image. The idea is that such generated tags could be used to improve image search. Also other variants of the game are available.

1.3 Collaborative Tagging

Collaborative tagging is the process of multiple users adding tags to content and using the user agreement on these tags. The main purpose of the collaborative effort is to achieve tags that accurately describe the content. The content can be multimedia files like pictures, audio, video, text, bookmarks or every identifiable object imaginable.

If the content is too large for a single authority to classify, a collaborative effort can be a good solution. However, if nobody has the 'librarian' role to review all user additions, the quality is an issue. Fortunately, the law of large numbers states that the average of many observations (the most occurring tags) will eventually be close to the population mean (a good tag set). A minimal number of tags is required before the tag set can be a good description of the content of a object.

The success of collaborative tagging depends on several factors. First of all, people usually only tag if they see benefits for doing this [Golder 2005]. For example, the personal benefit of bookmarking sites like Delicious is that your bookmarks are available everywhere. The overall benefit is that the very big collection of URLs on the web will get categorized and for instance 'popular' lists[4] can be generated. The extra overall benefit together with the easiness of use makes collaborative tagging popular.

Annotating content can have different forms. According to Scott A. Golder et al. [Golder 2005] different kinds of tags can be distinguished. Tags that identify what (or who) the content is about are far out the most used. But also tags that identify the kind of content, the owner, qualities or characteristics, and refinements are used. The last kind of tags they distinguish are only relevant for the tagger, they are for self reference and task organization. Other authors have made different categorizations, but are similar in the end. Due to the different kind of tags, not all tags will be equally useful for content describing and they may be ambiguous. For instance does the tag 'Vincent van Gogh' describe the Dutch painter himself or state that he is the painter of a painting. Additional context information or other tags can be disambiguating. Adding the tag 'painter' to the previous implies that the object describes the painter and not a painting. According to [Golder 2005] people tend to use general tags first which implies that the ordering of tags may contain semantics. If the object is the painter himself, users probably add the tag 'Vincent van Gogh' first. By describing a painting, tags like 'Sunflowers' or 'painting' are probable to be the first tag.

There are also linguistic and cognitive matters. Two objects can have different tags although they describe the same thing. These tags can be synonyms or plural forms. Also homographs (words with multiple meanings) can lead to confusion. Furthermore, the level of basic knowledge of people can introduce problems. For instance, tags like 'java' and 'php' could be too specific or incomprehensible for some users, where 'programming' is too general for others. This basic knowledge problem is a result of the non-hierarchical structure of a tag set. A hierarchical structure, like taxonomies, could eliminate the problem by relating the general and specific terms. Using taxonomies directly instead of tags would reduce the usability for the users. However, tool-support could use taxonomies to assist users with tagging.

[4] See for example http://del.icio.us/popular/

1.4 Ontology Mapping

Ontologies are designed to allow applications to agree on the terms that they use when communicating. There are many ontologies, all developed and maintained independently of each other. Therefore different ontologies may describe (roughly) the same concepts, using different structures, terms and depth. Also multiple applications need access to these different ontologies. Mapping these ontologies could provide a common layer for applications and exchange of information in a sound manner.

Ontology mapping is the task of relating the vocabulary of two ontologies in such a way that a translation from one ontology into the other preserves the intended semantic [Kalfoglou 2003]. The mapping is the recipe for the translation. Many different terms and approaches for ontology mapping exist. Ontology merging and integration generate a new ontology from two different ontologies. Ontology alignment is the task of creating links between ontologies to make them consistent with each other. Finally the term ontology translation is the actual translation which is defined by a mapping.

It is not likely that there will ever be a complete global agreement on terms and their meaning [Uschold 2004]. However, for some specific domains such agreement is possible to some extend. Vocabularies like Cyc[5], Dublin Core[6] and SKOS[7] are an attempt to create a common understanding. These vocabularies are not fixed, they can be extended. Ontology mappings to such shared ontologies increase the semantic interoperability. Applications that can handle these vocabularies can also handle ontologies that are mapped to these vocabularies.

Creating an accurate ontology mapping is still a process that is often performed by hand, possibly with the help of semi-automated mapping tools. Fully automatic mapping tools often use a combination of linguistic and learning techniques. However, the results are often not very precise.

1.5 Goal and project structure

This research examines the possibility to combine tagging and existing ontologies in such a way that the simplicity of tagging is preserved and the expressiveness of ontologies is obtained.

What we want to do is, given the input tags, find those concepts and corresponding labels that are most probably related to that input tag. These relationships between tags and concepts would allow us to exploit the semantics of ontologies.

The main goal of this project is:

Finding high-quality relations between tags and concepts in Semantic Web ontologies.

This research is conducted for two other projects. The first one is CHI2 (Cultuur Historische Informatie 2), which is a web view of a digital heritage collection. In order to improve keyword searches and assist users with tagging multi media objects, additional tags are found based on the semantics that can be derived from the user input. The second project is ViTa (Video Tagging

[5] See http://www.cyc.com/
[6] See http://dublincore.org/
[7] See http://www.w3.org/2004/02/skos/

project), which focuses on improving the retrieval of educational videos based on their content. Therefore metadata is added to the videos. Users can do that with tagging and they get assistance from suggestions based on relations that are found between tags and concepts. Both scenarios are fully described in Section 5.

Section 2 will explain how tag information can be extended. As a proof of concept we introduce a framework that is able to relate tags to concepts. The design of this framework is described in Section 3. It is followed by the implementation details in Section 4. The CHI2 and ViTa scenarios are described in Section 5. Next, several test cases and their evaluations are described in Section 6. And finally, Section 7 gives the conclusion and future work.

Section 2: Relate tags to concepts

Although collaborative tagging is simple for the user, the result is a flat tag set with very little semantics. As stated in Section 1.5, relating tags to existing structured data sources is an approach to preserve the simplicity of tagging and obtain the expressiveness of ontologies. This is achieved if tags can be correctly mapped to concepts. In that case the relations between the tag set and the ontology can be exploited to use all features of ontologies.

2.1 Scenarios

The use of tags in web-applications is increasing. Tags are used for several purposes: for categorization, as short descriptors, or for searching and navigation (e.g. tag clouds). The purpose of relating tags to concepts is discussed on the basis of four scenarios. In these scenarios the framework TagConceptualizer is used to relate tags to concepts.

2.1.1 Conceptualization

Tags are a very basic way to describe resources. Relating tags to concepts makes it possible for applications to obtain more semantical information of a resource. That is the purpose of conceptualization.

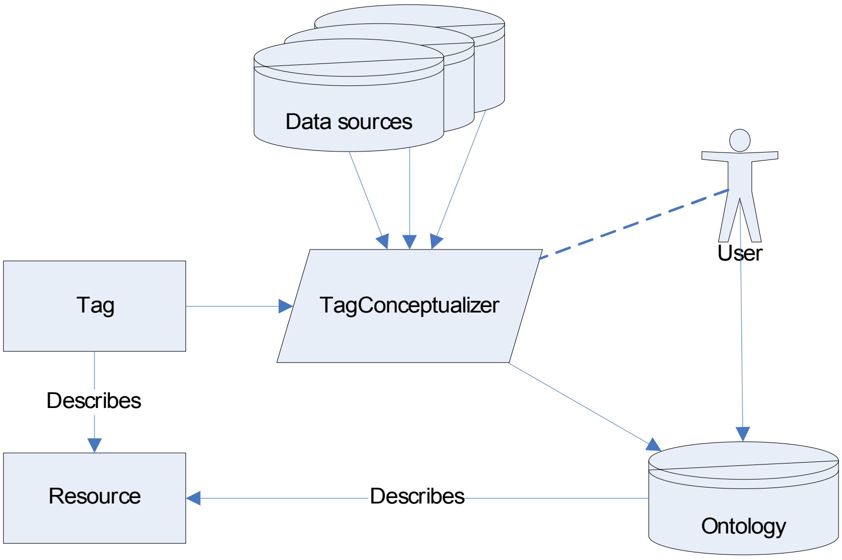

Figure 1: Scenario of Conceptualization

In each scenario ontologies are used as an information source. The input tags can be related to concepts within these ontologies. These concepts are used as alternative suggestions for the input tags. In order to find relevant suggestions, the ontologies must contain information of the same domain as the resource. However, they do not have to describe the individual resources. Relations between concepts in the ontology are used to find related and relevant concepts. Therefore, finding good concept suggestions depends on the quality of the source ontologies.

Figure 1 shows how tag descriptions are conceptualized. The tags that describe a resource are used as input for the TagConceptualizer. This tool relates the tags to concepts from ontologies. The concepts suggestions form an alternative description for the resource and contain more semantics than the input tags. Some of these concepts may be wrong or irrelevant because the input tags are homographs (words with multiple meanings), or not clear or complete enough. For example the input tag 'van Gogh' may relate to the concept of the Dutch painter Vincent van Gogh, but also to the concept of the film producer Theo van Gogh. Therefore they are presented to the user for approval. By storing the approved concept suggestions, an ontology that describes the specific resources is created.

2.1.2 Improve keyword searches

Keyword queries can be extended to increase the search results. Figure 2 shows this possible scenario. The user submits one or more keywords as the search query. These keywords are used as input tags for the TagConceptualizer. By default the output consists of concepts suggestions, but it is possible to output strings based on the labels of the concepts in the ontology. These tag suggestions, possibly together with the original keywords, form the new search query.

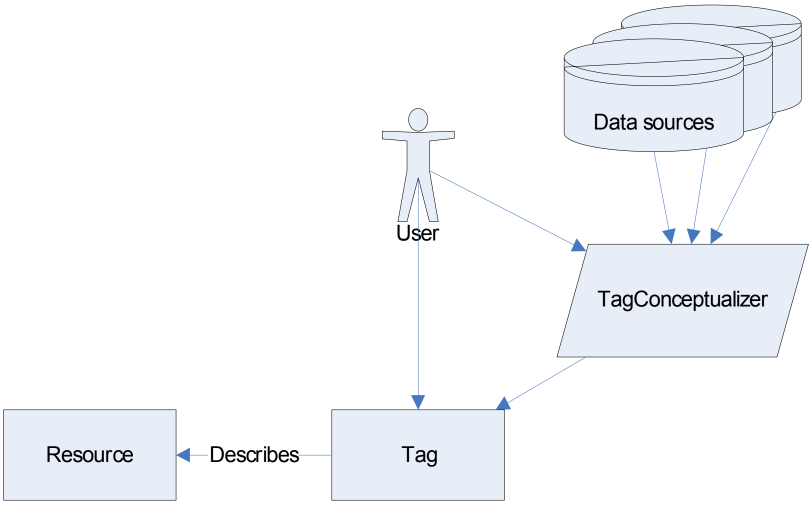

Figure 2: Scenario of Increasing keyword searches

For example, a user likes the painting 'Sunflowers' of Vincent van Gogh and wants to find more paintings of him. However the user does not know the painters name. He or she may use the keywords 'painting', 'sunflowers' and 'painter'. Normally, the search application will return all pictures with the tags 'painting', 'sunflowers' or 'painter' attached to it. If the keyword search is automatically extended with the tag 'Vincent van Gogh', the search engine can also find and return other paintings of Vincent van Gogh and not just any painter.

2.1.3 Tagging

Several web-applications or tagging systems encourage users to add tags to resources. The instant visibility is a motivation [Jakob Voss 2007]. However users create their own terms and probably do not tend to provide synonyms, different word forms and closely related terms, because that seems to be double work. With this scenario the additional tags do not have to be typed in by the tagger but are generated automatically by the TagConceptualizer that is able to suggest such additional tags.

The tags that users add to resources depend on what they find relevant. Therefore the tags heavily depend on the interest of the users and consistency among different taggers is difficult to achieve [Jakob Voss 2007]. The suggestions of TagConceptualizer influence the tagging behavior towards a consensus.

For categorization it may be necessary to have control of the vocabulary or tag set that describes the resources. It cannot be expected from users that they exactly know which tags are allowed. Influencing the tagging behavior avoids ambiguous or overlapping tags. For example, if the user adds the tag 'Vincent van Gogh' to a resource, but for categorization the tag 'van Gogh, Vincent' is used, a system could suggest that tag.

Taggers often create a minimal or incomplete tag set when describing a resource. This can have several reasons, like available time or knowledge, but mainly because users are unaware that software cannot make all implicit relations by themselves. Therefore a combination of generic and specific tags or different synonyms will not be part of a small tag set. Extending such tag sets can create a better description. Additional tags can be refinements of the original tags, like 'painting' is more specific than 'art work'. But extra tags can also be new information, for example the tag 'impressionism' adds information to the tag 'painting'.

The extended tag sets make it also easier to find the resources because they can be found by more tags. Therefore keyword searches are more likely to return a results that match the search query.

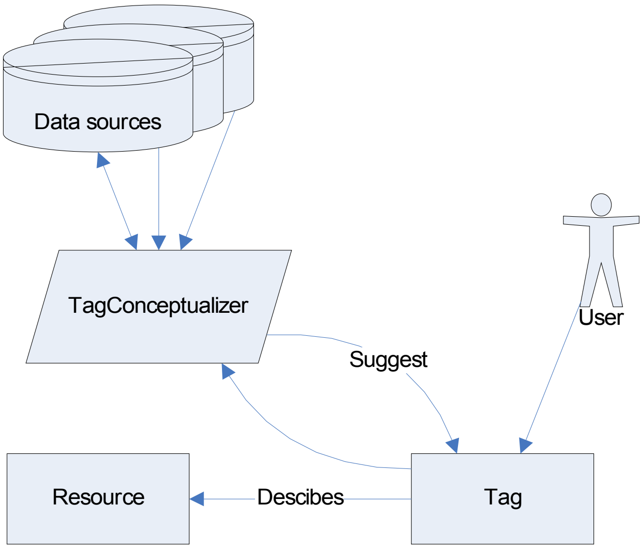

Figure 3: Scenario of Tagging

Figure 3 shows the tagging process. The resources can already have tags attached and the user add new tags. The complete tag set of a resource is used as input for the TagConceptualizer. The user can choose to add one or more of the returned tag suggestions to the resource.

2.1.4 Automatic Annotating

A lot of resources on the web are described by text documents. Such descriptions are understandable for users, but not for software. Text documents contain no structure that can be used by software to determine what the resource is about. Creating the metadata for the resources manually is a time consuming task. Fortunately, keyword extraction tools are available to extract keywords from the text documents. These keywords can be used as input tags for the TagConceptualizer. The returned concepts suggestions can be evaluated by a user and stored into an ontology that contains semantical metadata of the resources.

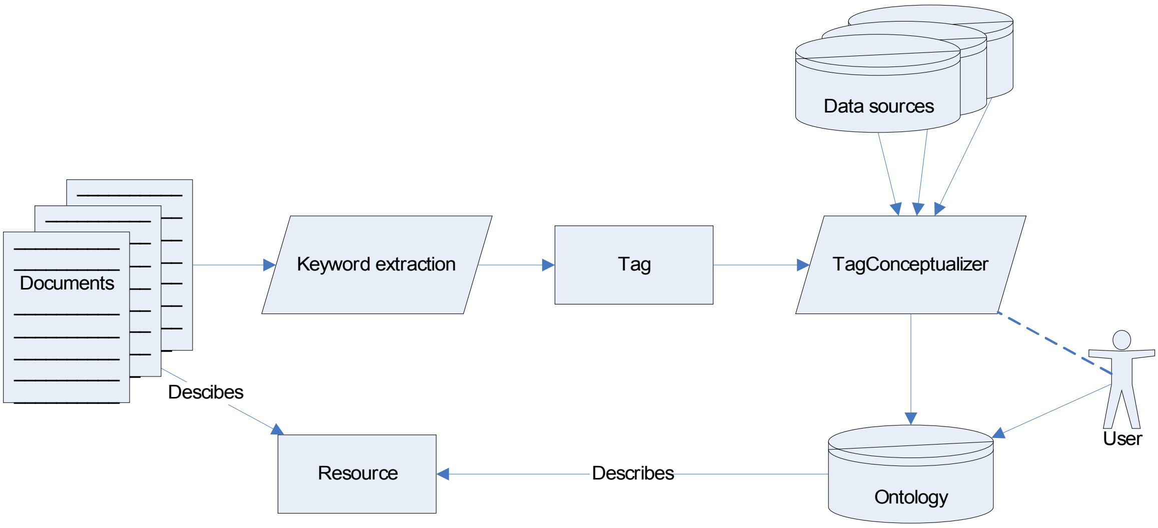

Figure 4: Scenario of Automatic Annotation

All the scenarios benefit from concept suggestions. The TagConceptualizer suggests additional concepts (or tags) based on a small set of tags. The following Sections describe several algorithms to extend tag information.

2.2 *String Matching*

2.2.1 From Tag to Concept

Using the possibilities of semantic web on a tag set requires that tags are first related to concepts. After that the concepts can be used to search and navigate through resources. However, tags and concepts are different things and their relation is not always straightforward. Concepts in RDF are represented by URIs and have no further semantics. For example, the tag 'Vincent van Gogh' and the concept 'http://repository.example.tld/0220321' are not alike on first sight. Fortunately a lot of concepts have related labels that can be short descriptors, titles, preferred labels, alternative labels, etc. This information can be extracted from concepts and compared to tags to find a relation.

The relation between concepts and their labels can be different for each ontology. Therefore it must be configured on beforehand how the labels with their associated concepts can be found. The labels are strings and therefore comparable to tags. In general, it cannot be assumed that they always exactly match, because of misspellings or different word forms. However, a string similarity metric can be used to find similar labels by calculating the string distance between the tags and labels. Different string similarity metrics exist. In this project, the Levenshtein metric, the Soundex metric and the JaroWinkler metric are used. They are described in more detail in Section 3.4.1. The normalized string distance is a number between zero and one and is used as the initial certainty of the concept. This certainty value describes the certainty that the concept found by this label is relevant.

2.2.2 Preliminary operation

Although string matching can relate tags to concepts, it is merely a first step in the conceptualization process. It does not use semantic relations between concepts to explore information. The main purpose of string matching is to find starting points in an ontology. Other algorithms use these starting points to find semantically related information.

Another purpose of string matching is to overcome spelling errors. People do not always use the correct spelling, or multiple spellings are correct like different word forms. For example the word 'postimpresionism' is syntactical close to 'post-impressionism' and therefore a concept on 'post-impressionism' is returned although the original tag was not correctly spelled.

2.3 Semantic Broadening

2.3.1 Explore interesting relations

For each setting specific semantic relations are interesting. These relations express what you want to find based on the information that is already present. Following semantic relations from the original concepts will return new concepts that are interesting and could be part of the suggestions. For example, if the intended purpose of using the TagConceptualizer is to get more specific information than the original information, relations like 'skos:narrower'[8] can be used to find such information. The suggestions for the concept 'painting' would then result in concepts on 'Still life', 'Oil painting', etc.

The previous example explores semantic relations in an ontology that represents the information domain. This algorithm can be executed multiple times with different ontologies and different relations. Therefore, also general purpose ontologies can be explored. For example, the Wordnet ontology can be used to find synonyms. Performing a semantic search to find synonyms for 'painting' would then for example result in 'illustration', 'visual arts', etc.

2.3.2 Calculating certainty

The TagConceptualizer determines for each concept a certainty. The certainty of new concepts is based on the certainty of the concepts they are related to. The initial certainty is calculated by the preceding algorithm, for example a string matching algorithm. The new certainty is the old certainty minus a predefined deterioration value.

[8] See http://www.w3.org/TR/swbp-skos-core-guide/

2.4 Context Disambiguation

Performing semantic broadening as described in Section 2.3 can result in a large number of concepts which all seem to fit to the input tags. These concepts can be based on terms that are syntactically close and therefore have similar certainties. However, the semantical difference could be bigger than is expressed with the current certainties. Therefore a way to refine the certainties of the concept suggestions is needed. There are several ways to converge the results and adjust the certainties.

- Merge multiple occurrences of suggestions and increase their certainty. If a concept is found more than once via different ways, it is more likely that this concept is relevant.
- Investigate the semantical closeness (e.g. number of relations from one concept to another) and increase the certainty of closely related concepts.
- Determine combinations of suggestions and increase the certainty of results that are part of reoccurring and overlapping combinations.

The first option is straightforward and available after each algorithm. If results are found multiple times, or already existed, they can be merged. The new certainty becomes: c2+c1*(1-c2) where c1 and c2 are the original certainties and c2 ≥ c1. This calculation ensures that the new certainty is between the highest original certainty and one. It seems a good choice, but other or better calculations can also work.

In the context disambiguating algorithm, the mutual overlap between set of tags that are related to the concepts, are examined (third option). The assumption is that tags of such combinations are more relevant if they occur more often. The initial certainty of the tags are based on the concepts that related to these tags. After that, the tag certainties are increased, based on the size and number of occurrences of the overlap. All tags with a certainty above a predefined threshold are returned as tag suggestions.

This algorithm is suitable for ontologies that describe collections with tags or labels. Therefore the result consists of tag suggestions and not concept suggestions. A variation on this algorithm is to examine the mutual overlap between combinations of concepts instead of combinations of labels/tags. In that case, the result consists of concepts suggestions. This variation is not implemented.

2.5 User Feedback

The quality of the suggestions depends on the used algorithms and data sources. If this data is static, the results are always the same. Although this is not necessarily bad, it means that a less relevant suggestion is always found for a certain input and there is no way to remove it permanently to improve the quality of future suggestions. On the other hand, suggestions that are often rated positively by users seem to be very good suggestions and therefore deserve a higher certainty. Semantical relationships in ontologies do not have a importance property, that means that each relations is equally important. However, some concepts occur more often and the chance that such concepts should be suggested is higher. The end-user can decide which suggestions are good or not. Storing and using these decisions improves the quality of future suggestions.

Each decision of the user to use a suggestion or explicitly decline it, is stored together with the input tags and final suggestion. This way, the certainties of found tags can be boosted if enough positive feedback is available for the combination of an input tag and a tag suggestion. The feedback consists of several counters and a stored certainty of the suggestion. The new certainty of the tag is the old certainty plus ((positiveCount-negativeCount)/totalCount)*(1-storedCertainty). This ensures that the new certainty is between the old certainty and one if there is enough positive feedback. If there is more negative feedback, the new certainty is between zero and the old certainty. The positiveCount and negativeCount are the counters for respectively the explicit user approvals and rejections of a suggestion. The totalCount is the number of stored feedback for the input tag and the storedCertainty is the stored certainty for the suggestion. If it finds a match and the new certainty is above a predefined threshold, it adds the stored suggestion to the current suggestions. Other suggestions are removed from the result.

2.6 Semantic closeness

The goal of the TagConceptualizer is to establish relations between tags and concepts in a way that they are semantically close. However, there is no exact or uniform way to calculate the semantic closeness of two objects. A possible approach is to determine the semantic closeness by domain experts.

All suggestions have a certainty. This certainty is an indication of the relative semantic closeness between an input tag and all its suggestions. A high certainty does not automatically imply a high semantic closeness. But a suggestion with a high certainty is most likely to have a higher semantic closeness than suggestions with a lower certainty for the same input tag. For example, an input tag results in three suggestions, with certainties of 0.7, 0.9 and 0.85. That means the input tag and the second suggestion are likely to have the highest semantic closeness, compared to the other two suggestions.

Section 3: Framework Design

Section 2 describes several algorithms to derive ontological information from tags. These algorithms are part of the framework TagConceptualizer. The purpose of this framework is to provide ontological information on tags to other applications that needs additional information on tags. This Section provides the detailed design of the TagConceptualizer.

3.1 Architecture

The TagConceptualizer is middleware. It extends tag information with concept or tag suggestions that are used by other applications. It is implemented as a Java library. The Java programming language is chosen because it is widely used, operating system independent and a lot of other libraries are available. Several libraries are used in this project, for storage and several algorithms.

The Sesame framework[9] is used for storage, inferencing and querying ontologies. This provides a way to use one method for accessing ontological information. Unfortunately, some queries have a weak performance. For example retrieving all labels from a repository. Therefore, the Apache Lucene project[10] is used to create indexes that enable fast lookup of tags or labels. These two approaches of storage are described in Section 3.3.

Normally, tags are represented by strings and concepts by URIs. But tags and URIs have also other properties that we want to utilize. Therefore the TagConceptualizer uses a internal representation for tags and concepts. These data types are explained in Section 4.1.

Different kind of algorithms are used to relate tags to concepts. The TagConceptualizer includes algorithms for string matching, semantic search and the usage of user feedback. Other algorithms can also be plugged-in. Descriptions of the algorithms can be found in Section 3.4.

The TagConceptualizer takes a tag set as input. This can be of the type TagTable or an array of strings. A TagTable object holds more information than an array of strings. The difference between them is explained in Section 4.1. Other formats can be converted into the TagTable type. The output depends on the configuration that is used. The two possible options are a set of tags or a set of concepts. These are of the type TagTable and ConceptTable. The configuration also contains which ontologies are used for getting information and storing user feedback. These ontologies can be stored in local repositories or remote repositories.

[9] See http://www.openrdf.org/doc/sesame2/users/

[10] See http://lucene.apache.org/

3.2 TagConceptualizer

Figure 5 gives a graphical view of the TagConceptualizer. The framework contains the algorithms as described in Section 2, with each a specific function. The exact behavior of the algorithms and the order of execution depends on a configuration file. First, an algorithm relates the input tags to concepts. After that other algorithms use these concepts to find other relevant concepts. It is possible to execute the algorithms after each other and use the output of one algorithm as input for the next. Finally, the concepts are returned as suggestions.

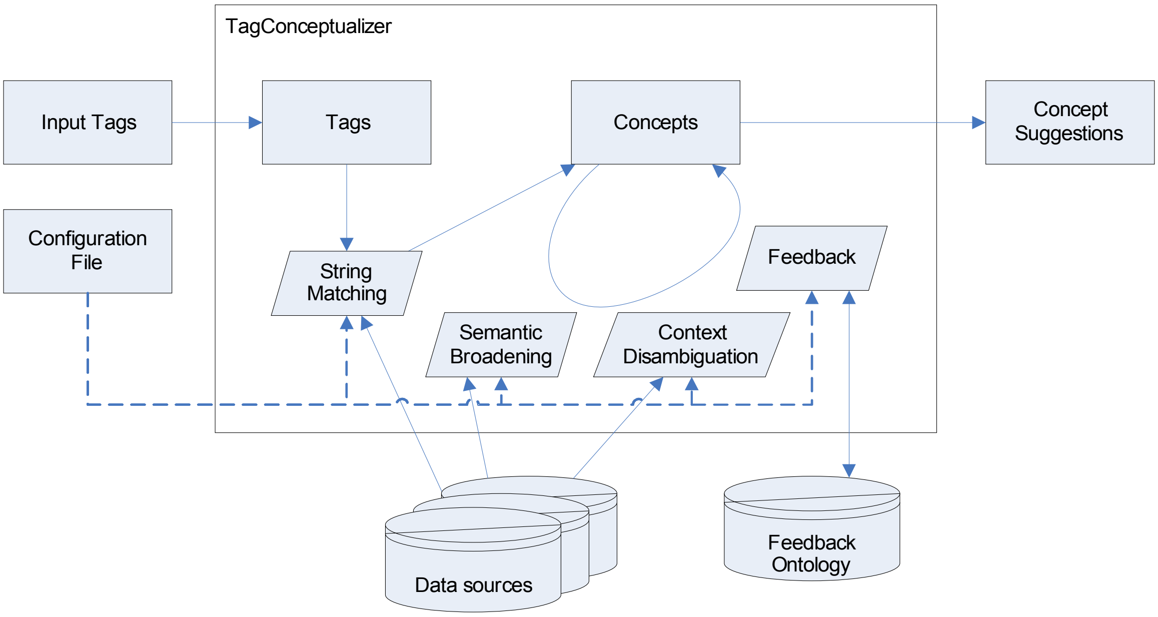

Figure 5: Graphical view of the TagConceptualizer

The aim of combining the effort of several algorithms, is to improve the quality of the suggestions. Each algorithm has its own purpose and focus. Achieving good results may require to explore several ontologies in different ways. See also the following example of a possible configuration of algorithms.

Table 1: Example of algorithm sequence

Algorithm	**Purpose and result**
String matching on a linguistic ontology	Overcome spelling errors, find concepts (words) for each tag
Semantic broadening on the same linguistic ontology	Find synonyms and other word forms
Convert concepts to tags	Result are tags with certainties attached; the new tags have correct spelling and contain synonyms
String matching on ontology with relevant information	Find concepts for each tag
Semantic broadening on ontology with relevant information	Find related concepts for each previously found concept
Semantic broadening on ontology with relevant information	Find more specific concepts for each previously found concept
Use User Feedback	Adjust certainties of concepts based on previously stored feedback
Prepare result: remove concepts with a low certainty and limit the maximum number of suggestions	Final concept suggestions

3.3 Storage

3.3.1 OpenRDF Sesame

As already stated in Section 1, ontologies and RDF data sources are a good way to annotate information and utilize logical properties. To use this semantical information, a repository is needed. Sesame is a suitable repository for storing, inferencing and querying such semantical information [Broekstra 2002]. According to [Lee 2004] Sesame has a good scalability and performance compared to other repositories and supports several query languages. It is a well known, frequently used and continuously developed semantical repository.

Sesame supports several ways of storing local repositories: RDBMS[11], in-memory or native (files). The native repository saves the ontology as files and has a short initialization time. In-memory repositories are loaded into the main memory. Sesame also support persistent in-memory repositories. These are also loaded into the main memory, but are saved as files after usage. Loading these files back into the main memory needs an initialization time. For the TagConceptualizer we used the persistent in-memory and native storage options with RDF Schema inferencing. These are directly available and do not need additional setup like RDBMS. The functionality of in-memory and native repositories do not differ from repositories based on RDBMS.

[11] Relational Database Management System

Several file formats are supported by Sesame and can be imported directly into a repository. The supported file formats are N3/Notation3, N-Triples, RDF/XML, TriG, TriX and Turtle. The TagConceptualizer uses Sesame to convert these files into local repositories.

The TagConceptualizer can also access remote Sesame repositories. This enables the use of external repositories that are created by others. It does not matters whether these remote repositories are implemented as in-memory, native or RDBMS.

Sesame supports the query languages SPARQL[12] and SeRQL[13]. Both query languages are similar, but with a different syntax and expressiveness. SeRQL was originally developed for Sesame and as a better alternative for the query languages RQL and RDQL. SPARQL is the upcoming W3C[14] standard. However, the TagConceptualizer uses SeRQL instead of SPARQL, because it has a more efficient Sesame implementation, is further developed and it has a user-friendly syntax.

3.3.2 Apache Lucene

Apache Lucene is a high-performance, full-featured text search engine library[15]. Lucene creates an index of the content it has to search. In our case that are pairs of concepts and their labels. The text search is like exact string matching, which will not suffice. But Lucene offers several additional searching capabilities. One of them is the usage of n-grams.

N-grams

N-grams are sub-sequences of n characters from a given string. They are used for several purposes, one of them is efficient string similarity matching. The assumption is that misspellings only affect a few of the constituent *n*-grams of a word [White 2005]. Strings are converted into a combinations of 3-grams and 4-grams and compared to the 3-grams and 4-grams of correctly spelled words. The proportion of the identical n-grams is a measure to recognize the intended word. Lucene treats this as a classic search engine problem with an inverted index on n-grams.

Searches in indexes are based on exact matches. Matching to the n-grams instead of the whole labels enables us to retrieve labels that are similar to, and not necessarily exactly the same as, the tags. This n-grams matching algorithm can be used on its own, or to speed up other similarity matching algorithms. If the accuracy of the n-gram similarity match is set to a low value, it returns a relatively large set of concept – label pairs.

The set of available string matching algorithms (see Section 3.4) is extended with a n-gram similarity matching algorithm. This can be used directly, or in combination with other similarity matching algorithms. The performance of the Lucene index is better than the performance of Sesame repositories, because it does not have to return all labels. Only the labels that have enough identical n-grams to the input tag are returned. This set of labels is a pre-selection for the string matching algorithms. As we will see in Section 6, this approach has a better performance than Sesame, because Sesame has to retrieve all labels from a repository.

[12] See http://www.w3.org/TR/rdf-sparql-query/

[13] Sesame RDF Query Language; see http://www.openrdf.org/doc/sesame2/users/ch09.html

[14] World Wide Web Consortium, see http://www.w3c.org/

[15] See http://lucene.apache.org/

3.4 Algorithms

3.4.1 String Matching

String Matching algorithms are used to compare words. Tag names can be compared to labels in a repository. These labels belong to concepts. If a tag name matches with a label, the associated concept is returned and can be further explored in the Semantic Matching algorithms.

Several different String Matching algorithms exist. The simplest one is to find an exact match. Or at least the tag is a sub string of the label. This approach uses a simple query per tag. The query has the following structure:

```
SELECT * FROM  {x} p {y} WHERE y LIKE “*Vinsent v. Goch*” IGNORE CASE USING
   NAMESPACE ...
```

The variable x represents the concept which can be returned as a concept suggestion. The predicate p is the relation between the concept x and the associated label y. This example returns all labels y that are equal to or contain the string 'Vinsent v. Goch'.

Unfortunately, it does not return all results in case of small spelling variations or misspelled words. In previous the example the tag 'Vinsent v. Goch' is used. A concept with the label 'Vincent van Gogh' is not found, although it is obvious that is should. That requires the use of similarity metrics.

Three string similarity metrics are available in the TagConceptualizer to compare labels in a repository. These are the Levenshtein Distance, JaroWinkler and Soundex.

Levenshtein Distance

This metric is an approximate string matching algorithm and was first published in 1966 [Levenshtein 1966]. It calculates the minimal edit distance of two strings. The operations of the algorithm are substitution, deletion or insertion of a character. The Levenshtein distance $d(s_1,s_2)$ between two strings is the minimum number of these edit operations required to make $s_1 = s_2$. This can be calculated using a (n+1 x m+1) matrix. For example, if s_1 = "Vinsent v. Goch" and s_2 = "Vincent van Gogh" the Levenshtein distance between s_1 and s_2 would be $d(s_1,s_2) = 4$ as the strings could be made equal by using two substitutions and one insertion. The corresponding matrix with the path to the minimal edit distance in the bottom right is presented in Table 2.

Table 2.

		V	i	n	s	e	n	t		v	.		G	o	c	h
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
V	1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
i	2	1	0	1	2	3	4	5	6	7	8	9	10	11	12	13
n	3	2	1	0	1	2	3	4	5	6	7	8	9	10	11	12
c	4	3	2	1	1	2	3	4	5	6	7	8	9	10	11	11
e	5	4	3	2	2	1	2	3	4	5	6	7	8	9	10	11
n	6	5	4	3	3	2	1	2	3	4	5	6	7	8	9	10
t	7	6	5	4	4	3	2	1	2	3	4	5	6	7	8	9
	8	7	6	5	5	4	3	2	1	2	3	4	5	6	7	8
v	9	8	7	6	6	5	4	3	2	1	2	3	4	5	6	7
a	10	9	8	7	7	6	5	4	3	2	2	3	4	5	6	7
n	11	10	9	8	8	7	6	5	4	3	3	3	4	5	6	7
	12	11	10	9	9	8	7	6	5	4	4	3	4	5	6	7
G	13	12	11	10	10	9	8	7	6	5	5	4	3	4	5	6
o	14	13	12	11	11	10	9	8	7	6	6	5	4	3	4	5
g	15	14	13	12	12	11	10	9	8	7	7	6	5	4	4	5
h	16	15	14	13	13	12	11	10	9	8	8	7	6	5	5	4

JaroWinkler

This metric [Winkler 1999] is a variation on the Jaro metric [Jaro 1989], which takes into account spelling deviations. Given two strings s$_1$ and s$_2$ the Jaro distance d_j is calculated as follows:

$$d_j=\frac{1}{3}\left(\frac{m}{|s_1|}+\frac{m}{|s_2|}\frac{m-t}{m}\right)$$

where:

- *m* is the number of characters of s$_1$ and s$_2$ that are not farther than

$$\left\lfloor\frac{max\left(|(s_1)|,|(s_2)|\right)}{2}\right\rfloor-1$$

- *t* is the number of transpositions, that are matching, but different characters divided by two.

The JaroWinkler distance uses a prefix scale *p* which gives more favorable ratings to strings that match from the beginning for a set prefix length *l*. Given two strings s$_1$ and s$_2$, their JaroWinkler distance d_w is:

$$d_w=d_j+(l\,p(1-d_j))$$

where

- d_j is the Jaro distance

- *l* is the length of the common prefix, with a maximum of four characters
- *p* is a constant scaling factor

The higher the JaroWinkler distance for two strings is, the more similar the strings are. This metric is designed and best suited for short strings such as person names.

Soundex

This metric is a coarse phonetic algorithm for indexing names by sound, as pronounced in English. It is based on the similarity of the pronunciation of two strings. It tries to encode the strings to the same representation so that they can be matched despite minor differences in spelling[16]. The algorithm keeps the first letter of the word and replaces other letters with numbers according to Table 3. Letters that are not listed in the table are ignored. Double numbers are removed and the string is transformed to 4 characters, add zeros if needed.

Table 3.

p, b, f, v	1
c, k, q, g, s, j, x	2
d, f	3
l	4
m, n	5
r	6

Applying the Soundex algorithm on the previous example shows that a match is found. Both 'Vincent van Gogh' and 'Vinsent v. Goch' get a Soundex code of V525.

Use of metrics

All these similarity metrics take two strings as input and return their similarity. The tags are compared to all the labels in the repository. If the similarity of a tag and a label is above some predefined threshold, the associated concept is added to the intermediate result. The similarity value is used as the certainty for that concept suggestion.

Retrieving all labels from a Sesame repository is a costly operation. Large result sets take a lot time to evaluate. If the repository contains a small dataset this is not an issue. However, performing such queries on large repositories drastically reduces the performance of the framework. This is a serious issue because some applications need concept suggestions in real-time.

As stated earlier in Section 3.3.2, using n-gram matching of Lucene to retrieve a pre-selection of labels improves the performance. A drawback of this solution is the possibility that some concepts are not found using one of the three similarity algorithms in combination with n-gram similarity algorithm. If a label is not returned based on n-gram matching, it cannot be discovered by another algorithm. Lowering the accuracy of the n-gram matching reduces this issue. However, we assume that the far better performance justifies the possibility of lost concepts.

[16] See http://www.archives.gov/genealogy/census/soundex.html

3.4.2 Semantic Broadening

The Semantic Broadening algorithm explores semantic relations between concepts. The input is a set of concepts that can be related to concepts that are part of the repository to search in.

Each semantic search is translated into a query. For example, if the types of the objects are interesting, a relation like 'rdf:type' can be followed. An input concept on the painter Vincent van Gogh would then result into a concepts suggestion on painter or person. The query has the following form:

```
SELECT * FROM  {x} rdf:type {y} WHERE x = <http://repository.example.tld/0220321>
```

The variable x represents the concept which can be returned as a concept suggestion. This query is executed for each input concept. That means it uses a concept from the input as a starting point, follows the relation of interest and returns a set of related concepts. The returned concepts of all queries are added to the intermediate result set. The certainty of a concept suggestion is based on the certainty of the input concept it is found by. In some cases it is assumable that the certainty of the concepts suggestions is less than the input concept. For example a relation like 'skos:related' returns related concepts, but they are not necessarily relevant. Therefore the new certainties can be decreased by a predefined deterioration value.

It may be necessary to explore several semantic relations to achieve good results. The semantic match algorithm can be executed several times with different parameters, queries and ontologies each time.

3.4.3 Context Disambiguation

The Context Disambiguation algorithm is a variation on the semantic broadening algorithm and also explores semantic relations between concepts. However, the purpose is not to find related concepts, but overlaps between sets of related tags. The query for each input concept returns a set of tags. If the sets of tags have a (partial) overlap with other sets of tags, then this overlap is a reoccurring combination of tags. This overlap is is used to calculate the new certainties.

The tag sets are compared and a list of tag combinations is determined. For example, if the input consists of three concepts, the queries return three tag sets: {tag1, tag2, tag3}, {tag1, tag4} and {tag2, tag3, tag5}. One combination of tags can be determined if the minimal size of a combination is set to two, namely: {tag2, tag3}. The minimal size of a combination can be set higher. It is assumed that these tags are more relevant and their certainties are increased with a predefined factor.

Also tags that occur more often in the result set are assumed to be more relevant. These tags are merged and the new certainty is between the highest certainty of the two tags and one.

The final result are tag suggestions. As already stated in Section 2.4 a variation would be to generate concept suggestions. In order to do so, overlap of sets of concepts instead of tags must be examined.

3.4.4 User Feedback

The User Feedback algorithm consists of two parts. The first part is retrieving feedback and storing it in a repository for later use. The second part uses the stored feedback to improve the tag suggestions.

Retrieving and Storing User Feedback

Retrieving and storing User Feedback is not used as an algorithm to search for tag suggestions. It is a separate process. But it is necessary for the part that does search for them.

Judgments of users are stored in the feedback repository. Users can accept, reject or ignore tag suggestions. Ignored tag suggestions are not stored in the repository, because it is uncertain whether the suggestion was good or bad. Only explicit judgments are stored into the repository.

Figure 6 shows the data model of the feedback ontology. If a user provides feedback, it consists of the original input tag, the tag suggestion of the TagConceptualizer, the previously calculated certainty, the user judgment and the user identification. Each input tag is stored only once, but it has a counter that represents the number of feedback for this input tag. Each combination of an input tag and its tag suggestion is also stored only once and two counters represent the number of positive and negative feedback. The certainty that was calculated by the TagConceptualizer for this tag suggestion and a related concept are also stored. Furthermore, each user judgment is stored together with the user URI and the score (positive or negative). The counters may seem superfluous as all judgments are stored, but the lack of support for aggregate functions in SeRQL makes them necessary.

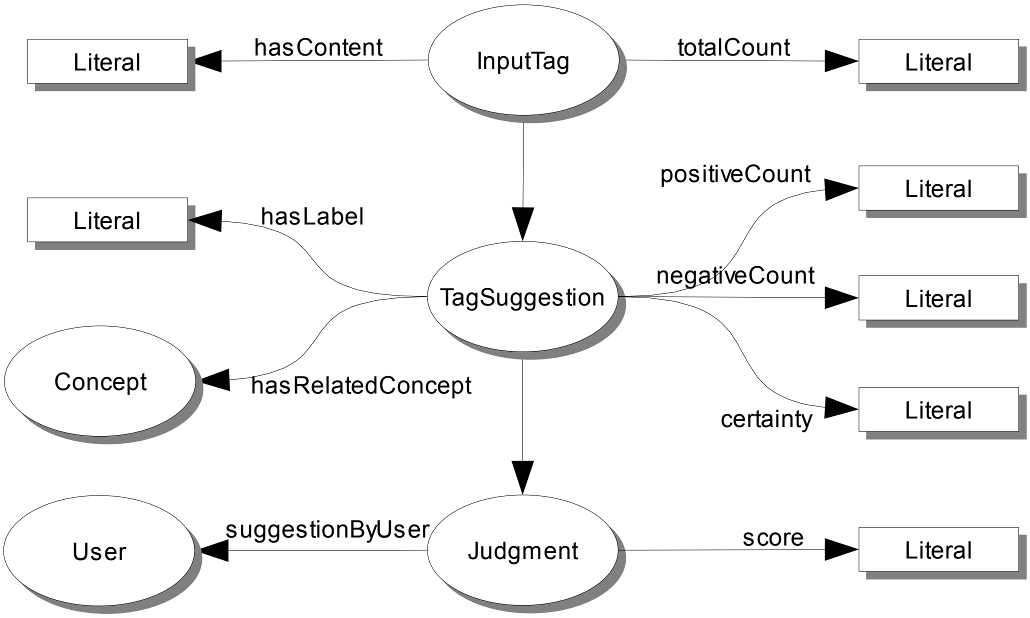

Figure 6: Data model User Feedback

Using feedback

The User Feedback algorithm that uses the stored user feedback to adapt the suggestions can be used as the other algorithms. The only difference is that this algorithm does not find new suggestions, but it adapts the certainties of the suggestions that are previously judged by other users. The calculation of the certainties is already described in Section 2.5.

3.5 Configuration

The exact behavior of the framework is dictated by the configuration. The class MCProperties is used to read and return these properties from a configuration file. The class TagExtendManager calls the algorithms based on the configuration and keeps track of the intermediate results. The output of one algorithm is the input for the next. Algorithms are invoked at runtime, there are no explicit calls. Therefore it is possible to plug-in new algorithms as a library with the same package name as the TagConceptualizer. It has only to be configured in the configuration file without the need to alter the source code of the TagConceptualizer.

The configuration file consist of several parts:

- Names of directories that contain (rdf) files, repositories and indexes that are used.
- Repository properties, for each available repository the name, type, and source files or server name and repository id.
- Index properties, for each available index the name, source repository and analyzer.
- Algorithm properties, for each algorithm the class name, method name, repository or index name. If a repository is used, the graphpattern, return variables, and namespace of the (SeQRL) query have to be supplied. Option properties are accuracy (for indexes), deterioration value, threshold, maxresults and how to handle the suggestions. These optional properties depend on the algorithm.
- Several general properties like the minimal threshold and the namespace of feedback ontology.

The details of the configuration options and an example configuration file can be found in Appendix B.

Section 4: Implementation

The previous Section describes the detailed design of the framework. This Section provides implementation details.

4.1 Data Types

The TagConceptualizer uses types to represent tags, concepts and lists of tags and concepts. Lists of strings and URIs do not hold additional information. However this additional information is useful to the algorithms. The types used in the TagConceptualizer hold additional properties, context information and relationships to other (lists of) tags or concepts. The class diagram of the data types are presented in Figure 7.

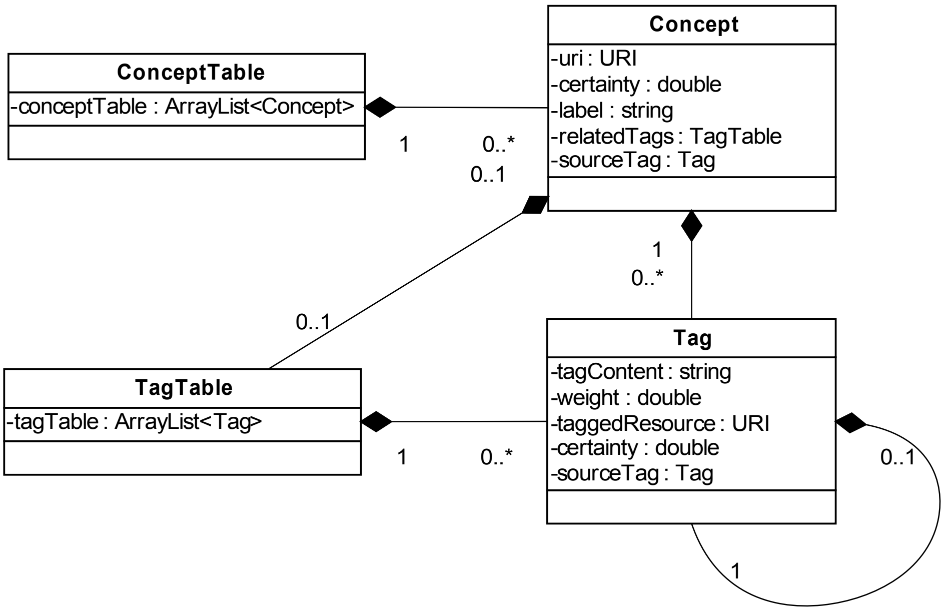

Figure 7: Class Diagram of Data Types

4.1.1 Tags

Tags (strings of text) are represented by the class Tag. Each tag of type Tag has the following properties.

Table 4: Properties of the class Tag

Property	Description	Type
tagContent	The actual string of the tag.	String
weight	The weight is and optional input parameter and is used to represent the importance of the tag. It influences the suggestions in such a way that suggestions based on tags with a lower weight get a lower certainty.	float
taggedResource	If the tag is associated with a concept, this property contains its URI.	URI
certainty	The certainty that this tag suggestion is relevant. For example, a tag with certainty 0.9 could be relevant, a tag with certainty 0.3 is supposed to be less (or not) relevant. Tags with a low certainty can be removed. Original input tags have a certainty of 1.	float
sourceTag	Each tag suggestion is based on one of the original input tags. This property relates a tag suggestions to an input tag.	Tag

The class Tag provides methods to create, merge, edit and compare tags. Two tags can be merged if they are equal. The new certainty becomes: c2+c1*(1-c2) where c1 and c2 are the original certainties and c2 > c1. Equal tags are tags with the same tagcontent regardless of the other properties. Tags are compared with their certainties.

Tag sets are represented by the class TagTable. This is a list of tags and provides methods to add, replace, remove (duplicate) tags and sort, add and merge tag sets. It is also possible to compare different instances of TagTable to determine their equality or overlap.

4.1.2 Concepts

Concepts (URIs) are represented by the class Concept which provides in several additional properties. These are described in Table 5.

Table 5: Properties of the class Concept

Property	Description	Type
uri	The actual URI of the concept.	URI
certainty	The certainty that this concept suggestion is relevant.	float
label	In several ontologies, concepts have labels. This property keeps the value of the preferred label. Note: this does not have to be a skos:prefLabel, which label is used depends on the configuration.	String
relatedTags	A list of associated tags.	TagTable
sourceTag	Each concept suggestion is based on one of the original input tags. This property relates a concept suggestions to an input tag.	Tag

The class Concept provides methods to create, merge, edit and compare concepts. Two concepts can be merged if they are equal. The new certainty becomes: c2+c1*(1-c2) where c1 and c2 are the original certainties and c2 > c1. Equal concepts are tags with the same uri. Concepts are compared with their certainties.

Concept sets are represented by the class ConceptTable. This is a list of concepts and provides methods to add, replace, remove (duplicate) concepts and sort, add and merge concept sets. It is also possible to convert a ConceptTable to a TagTable, but not the other way around.

4.2 Architecture

The class diagram of the TagConceptualizer is presented in Figure 8.

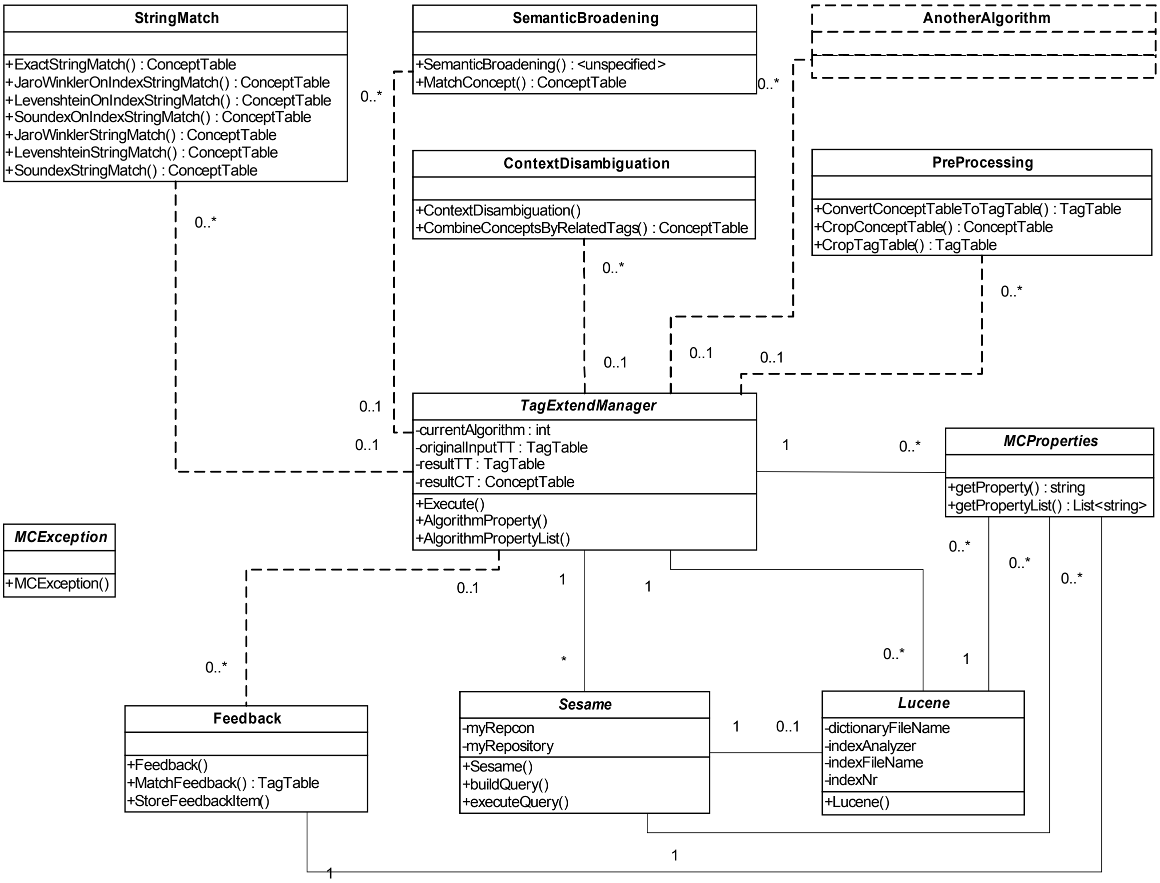
StringMatch
+ExactStringMatch() : ConceptTable
+JaroWinklerOnIndexStringMatch() : ConceptTable
+LevenshteinOnIndexStringMatch() : ConceptTable
+SoundexOnIndexStringMatch() : ConceptTable
+JaroWinklerStringMatch() : ConceptTable
+LevenshteinStringMatch() : ConceptTable
+SoundexStringMatch() : ConceptTable
SemanticBroadening
+SemanticBroadening() : <unspecified>
+MatchConcept() : ConceptTable
AnotherAlgorithm
ContextDisambiguation
+ContextDisambiguation()
+CombineConceptsByRelatedTags() : ConceptTable
PreProcessing
+ConvertConceptTableToTagTable() : TagTable
+CropConceptTable() : ConceptTable
+CropTagTable() : TagTable
TagExtendManager
-currentAlgorithm : int
-originalInputTT : TagTable
-resultTT : TagTable
-resultCT : ConceptTable
+Execute()
+AlgorithmProperty()
+AlgorithmPropertyList()
MCProperties
+getProperty() : string
+getPropertyList() : List<string>
MCException
+MCException()
Feedback
+Feedback()
+MatchFeedback() : TagTable
+StoreFeedbackItem()
Sesame
-myRepcon
-myRepository
+Sesame()
+buildQuery()
+executeQuery()
Lucene
-dictionaryFileName
-indexAnalyzer
-indexFileName
-indexNr
+Lucene()
0..*
0..*
0..*
0..*
0..*
0..1
0..1
0..1
0..1
0..1
0..1
1
0..*
0..*
0..*
0..*
0..1
1
1
*
0..*
1
0..*
1
0..1
1
1

Figure 8: Class Diagram of TagConceptualizer

The class MCProperties is used by other classes to retrieve properties of the configuration. It can return individual properties or a list of properties, for example all source files for a specific repository. The properties are read from the configuration file that is located in the classpath.

The classes Lucene and Sesame support the storage information. In this case Sesame stores semantical ontologies in repositories and Lucene creates indexes of tag names, their related concepts and n-grams of tag names. These classes use the class MCProperties to retrieve properties of the repositories and indexes. An index is always based on a repository.

The static class TagExtendManager contains the method Execute(tt). This method instantiates the models, invokes the algorithms and keeps track of the intermediate tag and concept suggestions. A TagTable and ConceptTable object are instantiated to store the intermediate results and are updated after the execution of each algorithm. The actual behavior depends on the configuration as defined in the configuration file. The configuration specifies which algorithms are used in which order and it specifies the parameters for each algorithm. For each algorithm the properties are retrieved. The method Execute instantiates and initializes the required repository or index and invokes the method which contains the appropriate algorithm. After all algorithms are executed, it returns the suggestions.

Several classes with algorithms exist. Each algorithm needs several parameters. The first parameter is an object of the type TagTable or a ConceptTable. This parameters contains the input tags or concepts that are to be extended. The second parameter is a model, which can be a instantiation of a Sesame object (repository) or a Lucene object (index). This model is used to find suggestions for the input tags or concepts. The method of the algorithm cannot directly retrieve the properties from MCProperties, but uses TagExtendManager.getAlgorithmProperties(s) to get the appropriate properties for the current execution of the algorithm.

Each class, except the data type classes for tags and concepts, can throw exceptions. Several exceptions can be of the type MCException. This type of exception is thrown if no standard Java exception is applicable. For example if the configuration of an algorithm uses an undefined repository, a MCException is thrown together with a message that this repository is not defined.

4.2.1 Execution

The framework is implemented in Java Standard Edition 5.0. It is implemented as a Java library that can be used by other applications. The method call TagExtendManager.Execute(tt) returns the suggestions. The parameter of this method must be one of the types TagTable, String[tagname][weight], String[tagname] or String. The first two types can also differentiate the weight of the input tags. The type of the suggestions can be a TagTable or ConceptTable. The following example uses an array of strings to represent the input tags and the result is a ConceptTable object which are the concept suggestions.

```
// Input tags
String[] inputTags = new String[2];
inputTags[0] = “tagcontent1”;
inputTags[1] = “tagcontent2”;

// Create objects for result
ConceptTable result = new TagConcept();

// Execute algorithms
result = (ConceptTable) TagExtendManager.Execute(s);

// Get concepts
for (int I = 0; I < result.size(); i++)
{
        URI u  = result.getConcept(i).getURI();
        double c = result.getConcept(i).getCertainty();
}
```

4.2.2 Dependencies

As already stated in Section 3.1, the TagConceptualizer depends on several other libraries.

- Repositories
  Sesame 2 is used to store and query repositories. Currently, it is still under development. The archive that is used is openrdf-sesame-2.0-beta5-onejar.jar. This archive provides support for local repositories. For remote repositories the following archives are also required: commons-codec-1.3.jar, commons-httpclient-3.0.jar and commons-logging-1.1.jar. These libraries of the Apache project enable the communicating to remote repositories.

- Indexes
  Apache Lucene is used to create indexes. A related project is the Lucene spellchecker. This library uses n-grams in its indexes. The corresponding archives are lucene-core-2.2.0.jar and lucene-spellchecker-2.2.0.jar.

- Similarity metrics
  Three similarity metrics are available in the framework. These are based on the corresponding metrics in the Simmetrics library. This library provides a variety of similarity metrics. The used archive is simmetrics_jar_v1_6_2_d07_02_07.jar. If stemming is used, the snowball archive is also needed: snowball-1.0.jar.

- Other dependencies
  The libraries mentioned above have also dependencies. Therefore the following archives must also be imported: slf4j-api-1.4.3.jar, slf4j-log4j12-1.4.3.jar, log4j-1.2.15.jar and xercesImpl.jar.

4.3 Setup

The framework is packaged as a single archive: tagconceptualizer.jar and can be imported into other Java projects as library. The package name is tagconceptualizer.tue.nl. Also the libraries that are needed for the framework must be imported.

Three directories are needed for the source files, for the local repositories and for the indexes. Ontologies and RDF files can be copied into source file directory. The syntax of these files can be RDF/XML, N3, N-Triples, Trix, Trig or Turtle. This list is based on the file types which are accepted by Sesame. The other two directories are used by the framework to store (native) repository files and indexes and therefore must have write permissions enabled.

Section 5: Scenarios

Tag-applications can benefit from the extension of tag sets with other tags or concepts. The framework, as described in Section 3, is middle-ware that provides this functionality. Section 2 already presented four general scenarios. This Section describes two actual scenarios that are implemented.

5.1 CHI2

5.1.1 Introduction to CHI2

CHI2 (Cultuur Historische Informatie 2) is a web view of the digital collection of RHCe (Regionaal Historisch Centrum Eindhoven)[17]. Visitors can browse through the collection and see (old) photographs, videos, etc. along with their descriptions. It uses semantic web technologies like ontologies to create relationships between the digital artifacts. The CHI ontology [Kamzol 2006] [Dorssers 2006] is developed for this purpose. It describes all kind of properties of the artifacts like authors, dimensions, titles, periods and terms (locations and keywords). The ontology also uses the concept Collection to groups different artifacts. Also several relations are used to relate different terms and thus artifacts to each other. These relations are:

Table 6: Relations of the CHI2 ontology

Relation	Description
wasVroeger	Indicates that a term is used in the past for the representation of the newer term. For example: "The Netherlands" <wasVroeger> "Republic of the Seven United Provinces".
bredereTerm	Indicates that a term has a broader meaning. This relation is comparable with <skos:broader>.
smallereTerm	Indicates that a term has a narrower meaning. This relation is comparable with <skos:narrower>.
omvat	This is a special kind of the <smallereTerm> relation. It indicates that a term is an instantiation of the concept of the other term. For example: "Painter" <omvat> "Vincent van Gogh".
gerelateerdAan	Indicates that two terms are related in another way than the relations described above. It is a symmetric relation and comparable to <skos:related>.

[17] RHCe is a regional heritage of the city Eindhoven in the Netherlands

5.1.2 The Problem

RHCe wanted to add their video collection to the digital collection and present it to their users. Unfortunately, these video and their fragments (parts of a video) are poorly described. At the moment, videos contain one sentence as a description, and the fragments only have a start and stop time. The video collection is large and keeps growing. Unfortunately, RHCe has only a limited number of domain experts to annotate these videos. Moreover, these experts do not know everything. For example, they can recognize a former mayor, but not all people on photographs. Their users could be helpful here. They can annotate video(fragments) and this meta data can be used to improve the access to and presentation of the videos. Because a lot of their users are older people, the annotation form should be easy. Tagging is easy and does not require knowledge of the terms used in the CHI ontology. Therefore it is a good candidate and enables a lot of people to offer help.

Another issue is retrieving (relevant) results when using keyword searches. The terms or tags which are used to annotate and archive objects at RHCe is limited. For example, it does not contain synonyms. Users use free texts for a keyword search, but that will currently only be successful if these exactly match the terms used for archiving. Therefore a mapping from the free texts to these archiving terms is needed.

5.1.3 The Solution

RHCe uses a more or less fixed thesaurus to describe their artifacts. This thesaurus can be a starting point to fill the CHI-ontology with information. The solution consists of two parts, the first for video tagging and the second for keyword searches,

The video tagging application allows users to tag the video and the individual fragments. The system can suggest additional tags and the user can select which of the suggested tags he wants to use. This results in bigger and broader tag sets, which makes the videos better to find. Moreover, tags that are related to concepts also improve the way videos can be found. For example, if the tag 'C. Church' is related to the concept 'Catherina Church', videos with this tag can also be found by a search for 'Churches' or 'Religious buildings' if these are related terms.

Currently the RHCe thesaurus is the ontology used to navigate through the collection. The navigation consists of several options. One can select times and locations or terms from a search cloud. Also a free text search is present. However, it cannot be assumed that users know the terms of this thesaurus. Therefore we have to make a mapping between their input search terms and the concepts in the ontology. There are several ways to solve this problem. For example presenting the user with the keywords they are allowed to use. This can be achieved by finding the correct spelling of a keyword with string matching.

Another approach, the one that is used in this project, is the following. The TagConceptualizer tries to relate the input of keyword searches with concepts in the CHI2 ontology. It explores several relations as described in Table 6. The found concepts are converted into terms. These terms come from the thesaurus. But it is not just a synonym finder, it also suggests related items. For example, a search for '18 Septemberplein' and 'bombing' could also return 'World War II', or maybe bombings nearby the 18 Septemberplein. The term 'World War II' can be found because a lot of objects are tagged with a combination of '18 Septemberplein', 'bombing' and 'World War II'. Therefore the context disambiguating algorithm finds the term 'World War II' and increases the certainty that is relevant because the combination occurs a lot.

The two solutions benefit from each other if feedback is used. When tagging videos, people use tags which are not necessarily included in the thesaurus. However, these user tags could be related to tags from the thesaurus. The assumptions is that if such user tags often result into the same tag suggestions, there exist a relation between the tags. Storing the user tags and their tag suggestions from the thesaurus gives insight into which tags are used by users. This is used to give popular tag suggestions a higher certainty and thus increasing the relevance of the tag suggestions in the future. It also works the other way around. Users will probably also use the same tag names when working with keyword searches as with tagging. These keywords can be related to the related tags which are part of the thesaurus and therefore find relevant results.

5.2 ViTa

5.2.1 Introduction

The ViTa project (Video Tagging project)[18] is started by the Telematica Insitituut and focuses on personalized retrieval of educational video material from Surfnet and Kennisnet. The Telematica Insitituut[19] is an international collaboration with companies, social institutions and knowledge organizations. It conducts research into ways of simplifying, and even more relevant, improving interaction. Surfnet[20] provides services like networks, group communication techniques and digital content to higher and academic education institutions. Kennisnet[21] aims at cooperation with schools, branch organizations and (municipal-)governments to provide tailor-made ict support within a broad spectrum of educational target groups in primary, secondary and adult education.

The videos from the ViTa project are presented through a web interface. Currently, the only information on videos are short textual descriptions. The project explores (scientific) ways to make this video collection available to students in such a way that the right videos are found with the limited amount of information available.

5.2.2 The Problem

Due to the limited information on videos, searching for relevant videos based on the content is difficult. The ViTa project explores several ways to overcome this problem. All approaches are based on tagging.

- Improve the user interface in a way that it is still intuitive and tags are presented with additional semantics. For instance, not just displaying or adding the word 'minister', but also be able to indicate that it is a person.
- Generating relevant tags based on video descriptions and related documents. Several automated text extraction methods can be used for this purpose.
- Relating tags to ontological information. Concepts of ontologies have more semantic meaning than plain words. These semantics can be explored and the results are converted to new relevant tags.

[18] See: http://www.telin.nl/index.cfm?ID=1351&context=1352&language=nl (Dutch)
[19] See: http://www.telin.nl/ (Dutch)
[20] See: http://www.surfnet.nl/ (Dutch)
[21] See: http://corporate.kennisnet.nl/international/

- Determining the correctness or quality of tags. Meta data is more useful if it consist of correct and high-quality tags. Irrelevant or wrong tags will results in wrong search results.

In this project the focus is on the third approach: relating tags to ontological information to find other related tags.

5.2.3 The Solution

The videos are tagged by users. To assist the users in creating high quality tags, the TagConceptualizer is used to provide tag suggestions based on the user tags as described in Section 2.1.3. Users of ViTa can tag the video´s. The user tags are related to concepts of the GTAA ontology and the TagConceptualizer suggests additional tags by exploiting the semantics of the concepts.

The GTAA ontology (Gemeenschappelijke Thesaurus Audiovisuele Archieven – Common Thesaurus for Audiovisual Archives) is a thesaurus on sound and vision which has approximately 160.000 terms in six facets: Subjects, Genres, Person Names, Names, Makers, and Locations. The GTAA ontology is provided by the Dutch institute for Sound and Vision[22] and is the official keeper of the Dutch audio-visual heritage [Brugman 2006].

Information from this Dutch ontology on sound and vision is applicable as annotation for the educational videos of the ViTa project. Therefore the GTAA ontology is used as the information source.

[22] See: http://www.beeldengeluid.nl/ (Dutch)

Section 6: Evaluation

6.1 Introduction

The main goal of this project as stated in Section 1 is to find high-quality relations between tags and concepts in Semantic Web ontologies. The framework as described in Section 3 relates tags to concepts in order to provide alternative tags or concepts. This Section describes several test cases and their evaluation.

6.1.1 Hardware

The test cases are performed on a Microsoft Windows XP workstation. It has a Intel® Pentium 4™ Processor with 3.00 GHz. The main memory (RAM) has a size of 1.00 GB. The Java compiler is executed with the option -Xmx640m to increase the heap size. This is necessary to load the repositories in the main memory.

6.1.2 Repositories

The GTAA ontology, as described in Section 5.2.3, is used in most tests. The size of the RDF representation of this ontology is 21.3MB. One test uses a small subset of the GTAA ontology which has a size of 2.23MB. Both ontologies are stored as Sesame memory repositories. This type of Sesame repository has the best performance, but requires enough main memory to load the data completely into it. Unfortunately, an initialization time to load such repositories is needed each time the TagConceptualizer is started. Accessing the repositories as remote (memory) repositories will eliminate the initialization time for each run because the sesame server keeps the memory repositories in memory. This approach is used for the test cases.

6.1.3 Indexes

All preferred and alternative labels of the concepts of both ontologies are stored in Lucene indexes. As stated in Section 43.3.2, an index is used to increase the performance of the string matching algorithms, but may result in fewer suggestions. For these test cases, the labels are retrieved from the repositories with the following query.

```
SELECT * FROM {x} p {y}
WHERE p = skos:prefLabel or p = skos:altLabel

USING NAMESPACE   gtaa=<http://www.beeldengeluid.nl/Thesaurus#>,
                  rdfs=<http://www.w3.org/2000/01/rdf-schema#>,
                  skos=<http://www.w3.org/2004/02/skos/core#>
```

In this query, variable x returns the concepts and variable y returns the related labels. All labels, together with the concept they belong to, are stored in a Lucene index. Also all all appropriate n-grams are stored together with the labels. This way labels are found based on n-gram matching and pairs of labels and concepts are returned.

6.1.4 Input tags

To determine and compare the quality of the suggestions of the TagConceptualizer, several tags are selected from a tag set and are used for all test cases. The tag set is obtained by a research of the ViTa project and contains 4500 tags. These tags are created by 200 students who tagged short educational movies. The following ten tags are randomly selected for the test cases: restaureer, platteland, emancipatie, amsterdam, rembrandt, rembrant, christendom, zang, koningshuis, koningin beatrix.

6.2 Questions

The purpose of the test cases is to determine the quality of the suggestions and the performance of the framework. This leads to the following questions.

- How does the quality of suggestions depends on the quality of the ontology and the algorithms?
- How big is the improvement of the performance if indexes are used?
- Does the use of indexes lead to loss of suggestions?
- How do the different string matching algorithms influence the result?

These questions are translated into test cases which are described in the following Sections.

6.3 Test Case 1

This test case is used to determine the quality of the suggestions made by the TagConceptualizer. The full GTAA ontology is used as semantical information source and an index improves the performance of the string matching algorithms.

6.3.1 Algorithm selection

A good configuration of algorithms is necessary to get high-quality suggestions. In this test case the configuration consists of three parts.

1. String Matching (Levenshtein) on an index. This is to find a starting point in the GTAA ontology. The intermediate result consists of concept suggestions.
2. Semantic Broadening. The concepts in the GTAA ontology are interconnected via semantic relationships. Related concepts are added to the intermediate result.
3. Convert Concepts to Tags. We chose to return the preferred labels of the concepts.

The exact implementation of this configuration can be found in the appendices. The semantic broadening algorithm uses the following query for each concept in the intermediate result.

```
SELECT * FROM {x} p {y} skos:prefLabel {z}
WHERE p = skos:related
OR      p = skos:narrower
OR      p = skos:broader
OR      p = gtaa:hasLinkedTerm
OR      p = gtaa:hasClassificationCode
USING NAMESPACE         gtaa=<http://www.beeldengeluid.nl/Thesaurus#>,
                        rdfs=<http://www.w3.org/2000/01/rdf-schema#>,
                        skos=<http://www.w3.org/2004/02/skos/core#>
```

The concept x is replaced by concepts of the intermediate result. This query is executed for each concept separately. The variable y returns the related concepts and variable z returns the preferred labels. The concepts and preferred labels are added to the intermediate result.

6.3.2 Results

The results of the test are presented in Table 7. The first column contains the input tags. The second column states the number of returned tag suggestions which are explicitly named in the third column. The certainty for each tag suggestion is presented in the fourth column. The last column gives the execution time for each iteration.

Table 7: Tag suggestions and performance for each input tag in test case 1

Input tag	Count	Tag suggestions	Certainty	Performance (seconds)
restaureer	9	restauraties huisvesting kunst en cultuur - algemeen filmrestauratie kunst monumenten monumentenzorg renovaties stadsvernieuwing	0,67 0,57 0,57 0,57 0,57 0,57 0,57 0,57 0,57	0,750
platteland	8	platteland ruimtelijke ordening / stad en platteland boerderijen dorpen ruimtelijke_ordening steden	1 0,9 0,9 0,9 0,9 0,9	0,765

		verstedelijking	0,9	
		weilanden	0,9	
emancipatie	13	emancipatie	1	0,797
		sociale vraagstukken	0,9	
		groepen in de samenleving	0,9	
		vrouwenemancipatie	0,9	
		arbeidersbeweging	0,9	
		bevolkingsgroepen	0,9	
		bevrijding	0,9	
		mannenbeweging	0,9	
		onderdrukking	0,9	
		vrijheid	0,9	
		vrouwenbeweging	0,9	
		Emancipade	0,73	
		Emancipatieraad	0,67	
amsterdam	9	Amsterdam	1	0,938
		Rotterdam	0,96	
		Nederland	0,91	
		Montana	0,79	
		VS	0,79	
		schepen	0,79	
		Westerdam	0,78	
		Gaasperdam	0,7	
		Amsterdamned	0,67	
rembrandt	2	Rembrandt	0,99	0,782
		Ere-Rembrandt	0,69	
rembrant	1	Rembrandt	0,95	0,766
christendom	34	christendom	1	0,891
		christenen	0,97	
		religies	0,9	
		katholicisme	0,9	
		oosterse_kerken	0,9	
		protestantisme	0,9	
		aartsbisschoppen	0,9	
		apocalyptiek	0,9	
		bijbelstudies	0,9	
		bisschoppen	0,9	
		christelijk_onderwijs	0,9	
		christelijke_feestdagen	0,9	
		christen-democratie	0,9	
		kerken	0,9	
		kerkgebouwen	0,9	
		kerkscheuringen	0,9	
		kruistochten	0,9	
		oecumene	0,9	
		priesters	0,9	
		sacramenten	0,9	
		theologie	0,9	
		zondagsrust	0,9	
		zonden	0,9	
		vorstendom	0,73	
		bevolkingsgroepen	0,63	
		heiligen	0,63	
		geschiedenis en staatsinrichting	0,63	
		staatsvormen	0,63	
		koningen	0,63	

		koninginnen	0,63	
		prinsen	0,63	
		prinsessen	0,63	
		troonopvolging	0,63	
		vorsten	0,63	
zang	25	zang	1	0,797
		muziek	0,99	
		liederen	0,99	
		muziek en podiumkunsten	0,9	
		jodelen	0,9	
		karaoke	0,9	
		zanghulde	0,9	
		gregoriaanse_muziek	0,9	
		koren	0,9	
		musicals	0,9	
		operas	0,9	
		operettes	0,9	
		oratoria	0,9	
		songfestivals	0,9	
		zangonderwijs	0,9	
		zand	0,75	
		Wang	0,75	
		producten	0,65	
		geologie / delfstoffen	0,65	
		grondsoorten	0,65	
		delfstoffen	0,65	
		stranden	0,65	
		zandopspuitingen	0,65	
		zandverstuivingen	0,65	
		zandwinningen	0,65	
koningshuis	5	koningshuizen	0,77	0,813
		Koningsvis	0,73	
		geschiedenis en staatsinrichting	0,67	
		hofhoudingen	0,67	
		vorsten	0,67	
koningin beatrix	0	-	-	0,609

The results show a few remarkable aspects with respect to the returned suggestions and the performance.

Analysis of found suggestions

The input tag 'restaureer' returned nine suggestions. We analyzed how these suggestions are found. The first step consists of matching the input tag 'restaureer' to all preferred labels returned from the index. This results in a concept and label. The next step in this test case is exploring semantic relationships. Several relations are found and the new concepts and preferred labels are added. The last step converts the concepts into tags, thus only keeping the preferred labels.

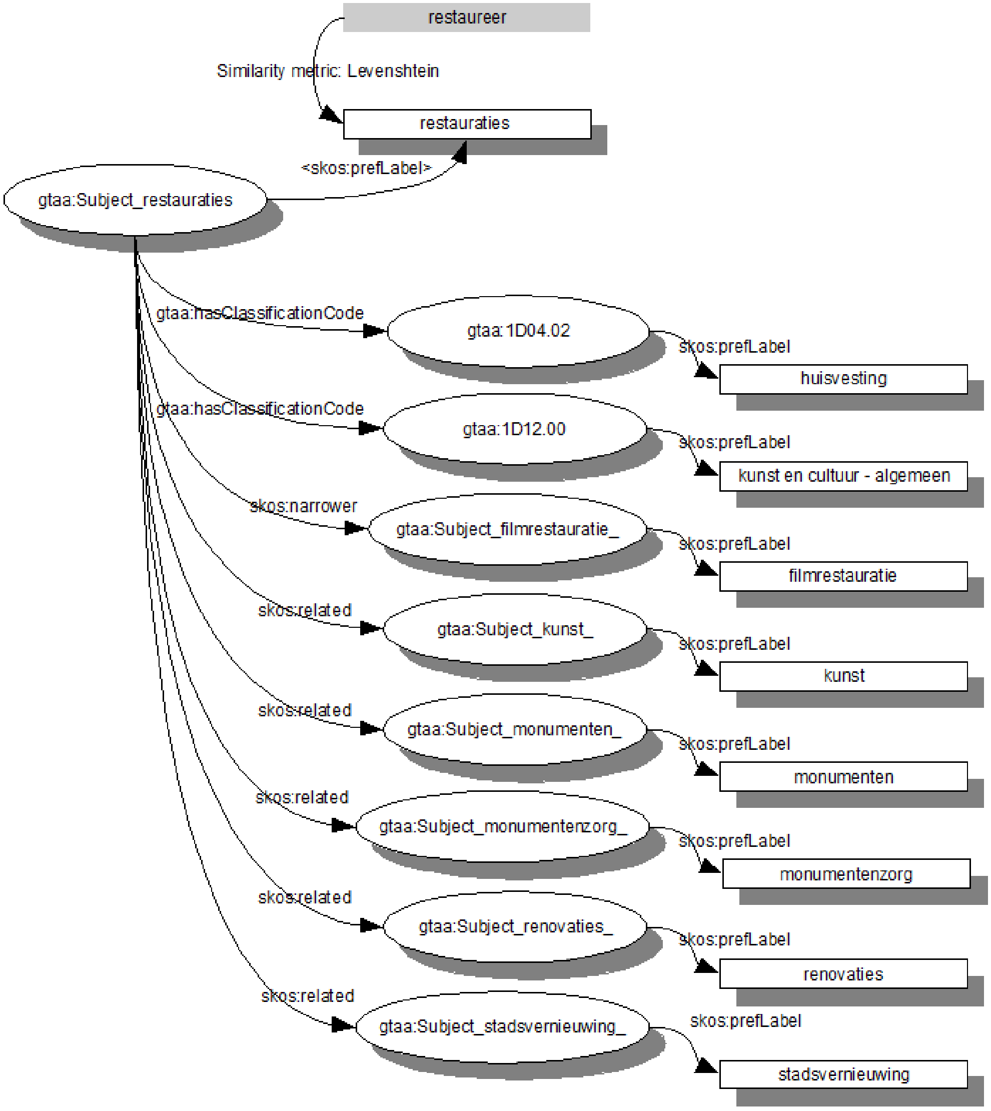

Figure 9: Analysis of found suggestions for the input tag 'restaureer'

Returned suggestions

Most searches return good tag suggestion for the input tags. The number of suggestions can be different for the different input tags. It depends on the number of related concepts that are found. The certainties of tag suggestions per input tag do often not differ much. However, it differs for the different tags. This is caused by the initial string matching algorithm. The syntactical closeness is used as a initial value for the certainty. Therefore, as already stated in Section 2.6, the certainty does not represent the semantic closeness, but indicates per input tag which suggestion is most likely to have the highest semantic closeness.

The input tag 'restaureer' results in several tag suggestions that can be divided into three categories. These are renovation of buildings, movies and art. The suggestions are all more or less valid and provide more specific descriptions. The second input tag 'platteland' has several good suggestions and two that are not straightforward: 'steden' and 'verstedelijking'. These are more opposites of each other. The suggestion 'bevrijding' for the input tag 'emancipatie' is an overstatement. The results for 'amsterdam' are spelled with a capital and are locations, except for 'schepen' and 'Amsterdamned'. The framework related them to names of cities and countries, but results like 'hoofdstad' or 'stad' are not returned. The input tag 'rembrandt' matches to the name 'Rembrandt' and the name of the prize 'Ere-Rembrandt'. It does however not relate to tags like 'schilder' or 'filmprijs'. Probably relations to the concepts of painters and prizes are not stored in the GTAA ontology. The input tag 'Rembrant' is a misspelling. But the tag 'Rembrandt' if found by the string matching algorithm. The suggestions for 'christendom' and 'zang' with a certainty of 0.9 or higher are as expected. The suggestions with a lower certainty seem not relevant. The string matching algorithm matched 'christendom' to 'vorstendom' and 'zang' to 'zand'. The irrelevant suggestions are related to these concepts. The tag 'koningshuis' also returns good suggestions. However the suggestion 'Koningsvis' is semantically not related to 'koningshuis'. The last input tag 'koningin beatrix' does not return suggestions because the string matching algorithms could not find a syntactically similar label in the ontology.

Several different measures for the evaluation search results exist. One of them is precision. That is the fraction of the documents retrieved which are relevant to the user.

$$precision = \frac{|\{\text{relevant suggestions}\} \cap \{\text{retrieved suggestions}\}|}{|\{\text{retrieved suggestions}\}|}$$

The precision of the suggestions for each input tag is calculated and presented in Table 8. The relevance of the suggestions have to be determined by an expert. In this case the researcher acted as the expert.

Table 8.Precision per input tag in test case 1.

Input tag	Relevant and retrieved suggestions	Retrieved suggestions	Precision
restaureer	6	9	0,67
platteland	5	8	0,63
emancipatie	8	13	0,62
amsterdam	4	9	0,44
rembrandt	2	2	1
rembrant	1	1	1
christendom	23	34	0,68
zang	15	25	0,6
koningshuis	4	5	0,8
koningin beatrix	0	0	-
Average			**0,72**

6.3.3 Conclusion

The purpose of this test case is to determine the quality of the suggestions and the performance of the framework. The results show that the designed framework in general returns relevant tags. A few exceptions were found. For example the unexpected tag suggestion 'Koningsvis' for the tag 'koningshuis'. This suggestion is found by string matching, but it does not seem to be a good semantical suggestion.

Although the framework presents good tag suggestions, sometimes expected suggestions are not returned. For example the input tag 'amsterdam' results in the locations 'Amsterdam' and 'Nederland'. The tag suggestion 'hoofdstad', which is a relation between the two suggested tags, is not returned. This is due to the fact that the GTAA ontology does state that the concept 'Nederland' is broader then the concept 'Amsterdam', but does not specify the relation between the two concepts.

The GTAA ontology includes concepts of locations that are streets, parks or districts. For example the 'Spaarndammerbuurt', 'Tuindor-Oostzaan' and 'Westerpark' are districts in Amsterdam (NL) and included in the GTAA ontology. These districts are concepts of the type Location and have a <skos:scopeNote> with the literal 'Amsterdam' or 'wijk_in_Amsterdam'. The relation <skos:scopeNote> was not included in the query and therefore the districts of Amsterdam were not found. However, the <skos:scopeNote> usually contains a textual explanation of the concepts and is not used to describe semantic relationships.

The average performance of the TagConceptualizer is 0,791 seconds. That is sufficient for (web)applications to work with.

6.4 Test Case 2

This test case is used to determine the influence of the ontology on the suggestions. Therefore this test case has the same configuration as the first test case, but uses another ontology. The small version of the GTAA ontology is used. This ontology contains less concepts and relationships. The assumption is that less tag suggestions are returned compared to the results of the first test case.

6.4.1 Results

The results of the test are presented in Table 9. The last column gives the average execution time for each iteration. The average is based on five executions to get reliable results and to remove noise.

Table 9: Tag suggestions and performance for each input tag in test case 2

Input tag	Count	Tag suggestions	Certainty	Performance (seconds)
restaureer	9	restauraties (volks)huisvesting (hierbij woningnood, renovatie) Kunst en cultuur - algemeen (hierbij musea, tentoonstellingen) filmrestauratie kunst monumenten monumentenzorg renovaties stadsvernieuwing	0,67 0,57 0,57 0,57 0,57 0,57 0,57 0,57 0,57	0,688
platteland	8	platteland ruimtelijke ordening, planologie, platteland, stedenbouw boerderijen dorpen ruimtelijke ordening steden verstedelijking weilanden	1 0,9 0,9 0,9 0,9 0,9 0,9 0,9	0,672
emancipatie	11	emancipatie sociale vraagstukken (hierbij armoede, agressie, werkloosheid) groepen in de samenleving (hierbij arbeiders, ouderen, migranten), emancipatie vrouwenemancipatie arbeidersbeweging bevolkingsgroepen bevrijding mannenbeweging onderdrukking vrijheid vrouwenbeweging	1 0,9 0,9 0,9 0,9 0,9 0,9 0,9 0,9 0,9 0,9	0,688
amsterdam	0	-	-	0,516
rembrandt	0	-	-	0,500
rembrant	0	-	-	0,484
christendom	34	christendom christenen religies katholicisme oosterse kerken protestantisme aartsbisschoppen apocalyptiek	1 0,97 0,9 0,9 0,9 0,9 0,9 0,9	0,781

		bijbelstudies	0,9	
		bisschoppen	0,9	
		christelijk onderwijs	0,9	
		christelijke feestdagen	0,9	
		christen-democratie	0,9	
		kerken	0,9	
		kerkgebouwen	0,9	
		kerkscheuringen	0,9	
		kruistochten	0,9	
		oecumene	0,9	
		priesters	0,9	
		sacramenten	0,9	
		theologie	0,9	
		zondagsrust	0,9	
		zonden	0,9	
		vorstendom	0,73	
		bevolkingsgroepen	0,63	
		heiligen	0,63	
		geschiedenis en staatsinrichting (hierbij staatsvormen, monarchie)	0,63	
		staatsvormen	0,63	
		koningen	0,63	
		koninginnen	0,63	
		prinsen	0,63	
		prinsessen	0,63	
		troonopvolging	0,63	
		vorsten	0,63	
zang	24	zang	1	0,657
		muziek	0,99	
		liederen	0,99	
		muziek en podiumkunsten (hierbij	0,9	
		theater, dans)	0,9	
		jodelen	0,9	
		karaoke	0,9	
		zanghulde	0,9	
		gregoriaanse muziek	0,9	
		koren	0,9	
		musicals	0,9	
		opera?s	0,9	
		operettes	0,9	
		oratoria	0,9	
		songfestivals	0,9	
		zangonderwijs	0,75	
		zand	0,65	
		specifieke producten	0,65	
		geologie, gesteenten, delfstoffen	0,65	
		grondsoorten	0,65	
		delfstoffen	0,65	
		stranden	0,65	
		zandopspuitingen	0,65	
		zandverstuivingen	0,65	
		zandwinningen		
koningshuis	4	koningshuizen	0,77	0,672
		geschiedenis en staatsinrichting (hierbij	0,67	
		staatsvormen, monarchie)	0,67	
		hofhoudingen	0,67	
		vorsten		

koningin beatrix	0	-	-	0,532

The tag suggestions of this test case are very similar to the results of test case 1. It is obvious that some concepts are not defined in the small GTAA ontology. No suggestion for the input tags 'amsterdam', 'rembrandt' and 'rembrant' are found any more. Further investigation of the ontology also pointed out that the relationship gtaa:hasLinkedTerm does not exist in the small GTAA ontology. The other input tags return more or less the same tag suggestions.

The precision of the tag suggestions for each input tag is calculated and presented in Table 10.

Table 10.Precision per input tag in test case 2.

Input tag	**Relevant and retrieved suggestions**	**Retrieved suggestions**	**Precision**
restaureer	6	9	0,67
platteland	5	8	0,63
emancipatie	7	11	0,64
amsterdam	0	0	-
rembrandt	0	0	-
rembrant	0	0	-
christendom	23	34	0,68
zang	15	24	0,63
koningshuis	4	4	1
koningin beatrix	0	0	-
Average			**0,71**

6.4.2 Conclusion

This test case compared the results for two different ontologies using the same configuration for the TagConceptualizer. The smaller GTAA ontology results in less tag suggestions for some input tags because is has less concepts defined and misses some relationships that are present in the full GTAA ontology.

If fewer relationships are defined, it means that the ontology contains less semantics. Only defining concepts, but not defining the relationships that connect them, makes it impossible to relate concepts to each other.

The average precision of the tag suggestions is almost the same when using the small GTAA ontology. However, not all input tags return results. The same average precision does not mean that the overall results are better. In fact they are worse, or at least smaller.

6.5 Test case 3

In the first two test cases a Lucene index is used to increase the performance. As stated in Section 4, this can influence the tag suggestions. This test is the same as test case 1, but without the use of a Lucene index. The Levenshtein string matching algorithm is now performed directly on the repository. The purpose of this test case is to investigate if the increased performance justifies possible loss of tag suggestions.

6.5.1 Results

The results of the test are almost the same as the results of test case 1. All tag suggestions and their certainties of test case1 are also found in test case 3. The additional tag suggestions are presented in Table 11.

Table 11: Tag suggestions and performance for each input tag that appears in test case 3 and not in test case 1.

Input tag	Total Count	Add. Count	Tag suggestions that are not in the results of test case 1	Certainty	Performance (seconds)
restaureer	9	0	-	-	7,344
platteland	8	0	-	-	7,187
emancipatie	13	0	-	-	7,766
amsterdam	13	4	Westerham VU Amsterdam FC Amsterdam HM Amsterdam	0,67 0,67 0,67 0,67	8,15
rembrandt	2	0	-	-	7,32
rembrant	1	0	-	-	6,828
christendom	34	0	-	-	7,625
zang	25	0	-	-	5,969
koningshuis	6	1	Ons koningshuis	0,73	7,453
koningin beatrix	2	2	Koningin Beatrix Koningin Beatrix-woud	0,88 0,67	8,609

All tag suggestions that are found in test case 1 are also found in this test case with the same certainties. Several additional tags suggestions are found. The input tag 'amsterdam' has four more results. Also, the input tag 'koningshuis' returns more tag suggestions. The additional tag suggestion is 'Ons koningshuis'. The last input tag 'koningin beatrix' returns two results. The tag suggestion ' Koningin Beatrix' is exactly the same as the input tag but uses capitals. This should normally be found using a n-gram matching algorithm as in test case 1. Probably the analyzer used for the Lucene index does not handle spaces very well. Another analyzer or removing the spaces could solve this issue.

The average execution time for each search is 7,425 seconds. That is far worse than the average execution time of test case 1 which is 0,791 seconds.

Table 12 presents the precision of the tag suggestions. The average precision is a little bit higher compared to the former test cases. Also, for each input tag one or more tag suggestions are found in this test case.

Table 12.Precision per input tag in test case 3.

Input tag	Relevant and retrieved suggestions	Retrieved suggestions	Precision
restaureer	6	9	0,67
platteland	5	8	0,63
emancipatie	8	13	0,62
amsterdam	7	13	0,54
rembrandt	2	2	1
rembrant	1	1	1
christendom	23	34	0,68
zang	15	25	0,6
koningshuis	5	6	0,83
koningin beatrix	2	2	1
Average			**0,76**

6.5.2 Conclusion

The string matching algorithm Levenshtein does find some additional tag suggestions if it is applied directly on the Sesame repository instead of the Lucene index. However the performance is far worse without using a Lucene index. If the intended use of the framework is to return suggestions in real-time, it is more useful to work with the Lucene index.

On the other hand, if the framework is not used for real-time suggestions, it is recommended to perform string matching directly on the repository. This can result in more or better suggestions because there is no pre-selection that can eliminate potential matches.

6.6 Test Case 4

The previous test cases indicated the importance of the string matching algorithms. If they do not return concept suggestions, the semantical broadening has nothing to start with. That means that in such cases the TagConceptualizer does not return suggestions.

This test case investigates the differences between several strings matching algorithms. The following (combinations of) string matching algorithms are available in the TagConceptualizer.

- Exact String match
- Levenshtein metric
- JaroWinkler metric
- Soundex metric
- N-gram matching (on index)
- N-gram matching & Levenshtein metric
- N-gram matching & JaroWinkler metric
- N-gram matching & Soundex metric

For each algorithm the number of concept suggestions and the performance are measured. The results are presented in Table 13 and Figure 10. The numbers in the table represent the number of concept suggestion for the input tag in the column and using the string matching algorithm in that row.

Table 13.Number of concept suggestion for each input tag and string matching algorithm in test case 4.

	restaureer	platteland	emancipatie	amsterdam	rembrandt	rembrant	christendom	zang	koningshuis	koningin beatrix
Exact string match	0	8	4	160	16	0	2	31	1	4
Levenshtein	1	1	3	9	2	1	3	3	3	2
JaroWinkler	2	4	3	5	1	0	4	7	4	5
Soundex	1506	682	1266	3369	1196	1196	3080	1042	2272	1753
N-gram matching	23	24	24	16	23	22	20	25	23	7
N-gram Levenshtein	1	1	3	5	2	1	3	3	2	0
N-gram JaroWinkler	2	2	2	2	1	0	2	2	4	1
N-gram Soundex	2	4	4	5	5	1	13	9	9	6

The Soundex metrics finds by far the most matches if applied directly on the Sesame repository, but much less in combinations with the Lucene index. The threshold of the Soundex algorithm was increased to 0.85 to limit the search results, but increasing it more would give zero results if used with the Lucene index. The Soundex metric used in this test case was intended for the Englisch language, but applied on Dutch words. This could explain the weird behaviour.

The Levenshtein en JaroWinkler algorithms produce more or less the same number of suggestions. The latter a little less in combination with the Lucene index. The JaroWinkler metric returned some relevant matches which were not returned by the Levenshtein algorithm. For instance for the input tag 'koningin beatrix' it returned also 'koningkwesties' and 'koninginnen'.

The exact string match algorithm has varying results. The input tags 'restaureer' and 'rembrant' are not found in any label in the ontology. The input tag 'amsterdam' on the other hand is found often. 'Amsterdams' and 'Amsterdamse' occur often as a prefix, for example in 'Amsterdamse Effectenbeurs'. This algorithm did also return several other relevant tags that were not found by any other metric. The input tag 'Rembrandt' returned suggestions like 'Rembrandt van Rijn', 'Rijn, Rembrandt van' and 'Rembrandtplein'. Probably these suggestions differ to much in length to be found by Levenshtein or n-gram match.

The performance of the three similarity metrics applied directly on the Sesame repository is slow. The soundex metric does perform worst because the slightly slower performance of this metric is magnified by the large number of executions (one for each label). If the three similarity metrics are applied on the Lucene index, the performance is much better. The difference is that these metrics use the result of the n-gram algorithm, which consist of at most 25 labels in this test case, instead of all the labels in the GTAA ontology which are approximately 160.000 labels [Brugman 2006]. The performance of the exact string match is a little slower than the use of n-grams with a similarity metric, but still much faster compared to the similarity metrics without the Lucene index.

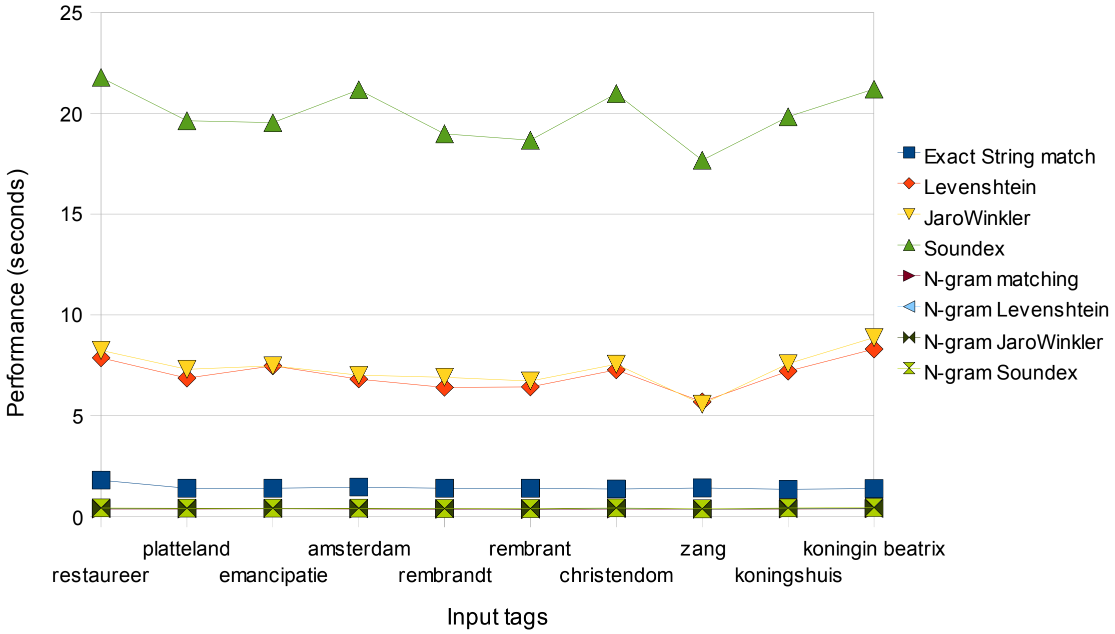

Figure 10: Performance of string matching algorithms in test case 4.

6.6.1 Conclusion

The Levenshtein and JaroWinkler algorithms produce relevant results. Unfortunately also some irrelevant results, but that cannot be completely avoided. In combination with an Lucene index, their performance is also acceptable. The exact string match algorithm could produce additional suggestions. In this test case it did find several additional relevant suggestions.

It is important to choose a good threshold for the similarity metrics. A low threshold results in too much (irrelevant) results. Only a couple of concept suggestions are enough. The semantic broadening algorithm can be used to explore the ontology further and find semantically related concepts. However, if the threshold is too high, no suggestions are returned at all.

Section 7: Conclusion and Future Work

7.1 Conclusion

The main goal of this project is to find high-quality relations between tags and concepts in Semantic Web ontologies. For this purpose the TagConceptualizer was designed. This framework uses existing technologies like similarity metrics, tagging, semantic web ontologies, etc. and combines them. It is able to provide semantically related suggestions on tags in real-time by utilizing the Lucene framework for indexing labels.

The framework is tested with the GTAA ontology as information source. The results show that the designed framework produces good tag suggestions. This implies that high-quality relations between tags and concepts are made.

Although the overall result consists of relevant suggestions, sometimes an unexpected suggestion is returned. This noise can have several reasons. The framework uses string matching algorithms that return results based on syntactical closeness. If the source ontology has a broad scope, several matches from different domains are returned.

The result of the test cases also indicated that sometimes an expected tag suggestion is not returned. The framework can only find relations if they are already present in the ontology that is used. Therefore the quality of the ontologies are important in order to find relevant relations between tags and concepts.

7.2 Future Work

7.2.1 Pre-editing labels

The initial relation between tag and concepts is currently made by string matching. Doing this directly on a Sesame repository decreases the performance, and using an Lucene index can reduce the number of found relations. If both the input tags and the labels in the index can be adjusted to a sort of normal form before comparison, it would relax the string matching. For example, only using the stemmed versions of words, or removing all spaces to overcome the issue discussed in Section 6.5.1.

Another approach is to create expected variations on all labels and store them also in the index. That would make the use of string matching superfluous, but could require a lot of disk space. Expected variations can be typical misspellings or words that have a similar pronunciation.

7.2.2 Ontology learning

The User Feedback algorithm enables users to provide feedback on suggestions. These suggestions are stored into a separate (feedback) ontology. Suggestions that receive positive feedback could also be related to concepts in the ontology that is used as semantical information source. This would enrich the ontology and makes and separate algorithm to explore user feedback superfluous. Moreover, the feedback is available for other applications that use the same ontologies.

7.2.3 Advanced User Feedback handling

The User Feedback algorithm uses several counters and a certainty of the tag – concept relationship to adapt the suggestions. Several learning algorithms like Bayesian or neural networks exist which can be applied on user feedback. Also differentiating on users or user groups can reveal other patterns. Using such techniques makes the use of user feedback more dependable.

7.2.4 Server version

The TagConceptualizer is implemented as a Java library that is executed separately for each suggestion. That requires a lot of initialization time. Creating a sever version would require just one initialization. The same connections to the repositories can be used for all suggestions. This would increase the performance of the TagConceptualizer. Moreover, it improves the accessibility as it can be implemented as a web service.

Bibliography

[Bateman 2006] Scott Bateman, Christopher Brooks and Gord McCalla *Collaborative Tagging Approaches for Ontological Metadata in Adaptive E-Learning Systems*, 2006

[Berners Lee 2001] Tim Berners Lee *The Semantic Web*, 2001

[Broekstra 2002] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen *Sesame: A Generic Architecture for Storing and Querying RDF*, 2002

[Brugman 2006] Brugman, H., Malaisé, V., Gazendam, L. *A Web Based General Thesaurus Browser to Support Indexing of Television and Radio Programs*, 2006

[Dorssers 2006] G.H.J. Dorssers *Ontsluiting van Cultuurhistorische Informatie de chi-navigator: datamodel & implementatie*, 2006

[Golder 2005] Scott A. Golder, Bernardo A. Huberman *The Structure of Collaborative Tagging Systems*, 2005

[Jakob Voss 2007] Jakob Voss *Tagging, Folksonomy & Co - Renaissance of Manual Indexing?*, 2007

[Jaro 1989] M.A. Jaro *Advances in record linking methodology as applied to the 1985 census of Tampa Florida*, 1989

[Kalfoglou 2003] Yannis Kalfoglou and Marco Schorlemmer *Ontology mapping: the state of the art*, 2003

[Kamzol 2006] F.T.M. Kamzol *Ontsluiting van Cultuurhistorische Informatie de chi-navigator: datamodel en gebruikerstest*, 2006

[Lee 2004] Ryan Lee *Scalability Report on Triple Store Applications*, 2004 http://simile.mit.edu/reports/stores/stores.pdf

[Levenshtein 1966] V. Levenshtein *Binary codes capable of correcting deletions, insertions, and reversals*, 1966

[Uschold 2004] Michael Uschold and Michael Gruninger *Ontologies and Semantics for Seamless connectivity*, 2004

[White 2005] Tom White *Did you mean: Lucene?*, 2005 http://today.java.net/pub/a/today/2005/08/09/didyoumean.html

[Winkler 1999] W.E. Winkler *The state of record linkage and current research problems*, 1999

Appendices

Appendix A: Configuration manual

The TagConceptualizer can be configured with the file tagconceptualizer.props. The configuration consists of the following parts: File locations, repository configurations, index configurations, algorithm configurations and some default properties.

File locations

The following locations must be defined

- FILE_BASE
  Ontologies and RDF files can be copied into source file directory. These files are used to build repositories.
- REPOSITORY_BASE
  The repositories that are created by the TagConceptualizer are stored into this directory.
- INDEX_BASE
  Indexes that are created by the TagConceptualizer are stored into this directory.

Repositories

Multiple repositories can be defined in the configuration file. The repositories are numbered and identified by their (unique) name. The prefix for a repository property is: Repository.n. where n denotes the number of the repository. The obligated properties for a repository are:

```
Repository.0.Name        = ...
Repository.0.Type        = ...
```

The type can be one of the following: remote, native, or memory. In case of a remote repository, a server and repository id must be defined.

```
Repository.0.SesameServer      = http://...
Repository.0.RepositoryID      = ...
```

In case of a native or memory repository, the source files must be defined. If only one source file is specified, the file number can be omitted. The files are relative to the FILE_BASE.

```
Repository.0.FileName.0.       = ...
```

Indexes

Like repositories, indexes are numbered and identified by their unique name. Other properties are the name of the repository that has to be indexed and the name of the analyzer used to index the repository. Possible analyzers are: StandardAnalyzer, SimpleAnalyzer and KeywordAnalyzer. These are based on the available analyzers of Lucene.

```
Index.0.Name                = ...
Index.0.RepositoryName      = ...
Index.0.Analyzer            = ...
```

The query that is used to retrieve concepts and labels from the repository is defined by the first algorithm that uses this index. This query is needed to create an index and therefore only used when the index is not present.

Algorithms

The algorithms are also numbered, but have no name. They are identified by their class and method names. Each algorithm uses one repository or one index as an information source. If applicable, a deterioration value can be defined to lower the certainty of the algorithm suggestions. A threshold value is used to filter the suggestions with a low certainty from the algorithm result. There are several ways to combine the result of the algorithm to the intermediate result of the TagConceptualizer.

- Merge results, duplicates are merged into one and get a higher certainty.
- Add results, duplicates are allowed.
- Add and replace results, in case of a duplicate, the suggestion with the lowest certainty is removed.
- Replace results, old results are removed and only the results of the algorithm are retained.

In case of an index, an accuracy can be defined. This accuracy is a number between zero and one and influences the retrieval of the labels based on n-grams.

```
Algorithm.0.ClassName           = nl.tue.tagconceptualizer. ...
Algorithm.0.MethodName          = ...
Algorithm.0.Index/Repository    = ...
Algorithm.0.Threshold           = ...
Algorithm.0.Accuracy            = ...
Algorithm.0.HandleResult        = ...
Algorithm.0.Query. ...          = ...
```

For several algorithms it is necessary to define a query. The definition of a query in the configuration file consists of several parts. The graph pattern, where clause and namespace are defined as separate properties. The graph pattern represents the variable names and relations between them. The ReturnVar and ReturnConcept properties can be used to retrieve these variables. If a query is applied for each input tag or concepts, a ReplaceVar can be defined to point out the place where the input tag or concept can be fit in.

```
Algorithm.0.Query.GraphPattern      = ...
Algorithm.0.Query.Where             = ...
Algorithm.0.Query.ReturnConcept     = ...
Algorithm.0.Query.ReturnVar         = ...
Algorithm.0.Query.Namespace         = rdfs=<http://...#>, ...
```

Default properties

A default minimal threshold can be defined to filter out intermediate suggestions with a low certainty. Also the namespace of the feedback ontology must be defined if feedback information is used.

```
Default.Threshold            = ...
Default.FeedbackNamespace    = ...
```

Several boolean properties are available for debugging. These are ShowIntermediateResults, ShowFinalResults, ShowTimeTable, ShowTotalTime and PrintQuery. If they are not defined, their values are false.

Appendix B: Example of configuration file

Example of configuration file tagconceptualizer.props

```
#Property file of TagConceptualizer

# File and repository locations (must be relative to classpath)
# REPOSITORY_BASE is the folder where the Sesame repository files will be stored. This
    folder may be initially empty, but the folder itself has to exist
# FILE_BASE is the folder where the ontology-files, as mentioned bellow, are located (given
    that not all repositories are remote).
# INDEX_BASE is the folder where the indexes of repositories are located
REPOSITORY_BASE   = ../Repository/
FILE_BASE         = ../Files/
INDEX_BASE        = ../Index/

# Repositories to be used
# If a repository is unavailable, the (owl)file is used to build it
# Name is the name of the repository, so that we can refer to it in the Algorithm selection.
    Compulsorily.
# FileName is the name of the RDF-based-file. If this file exist in the Repository_Base that
    will be used, otherwise the source file will be used to (once) build the repository. The file
    must exist in the FILE_BASE directory.
# Type of store. Possible values are "memory", "native" or "remote". Native might be a bit
    faster (except the first run which is rather slow), but native doesn't include inferences,
    were memory does.

# Repository 0 is the Feedback ontology
Repository.0.Name = Feedbackremote
Repository.0.SesameServer = http://localhost:8080/openrdf-sesame/
Repository.0.RepositoryID = Feedback
Repository.0.Type = remote

# Repository 1 is the full GTAA ontology (21MB)
```

```
Repository.1.Name = GTAA
Repository.1.FileName.0 = GTAA/GTAA2006locationAxisRevised.owl
Repository.1.FileName.1 = GTAA/GTAA2006makersAxisV2.owl
Repository.1.FileName.2 = GTAA/GTAA2006namenAxis-edited.owl
Repository.1.FileName.3 = GTAA/GTAAjune2006genres.owl
Repository.1.FileName.4 = GTAA/GTAApersonAxisV6.owl
Repository.1.FileName.5 = GTAA/SubjectAxis2006v4.owl
Repository.1.Type = memory

# Repository 2 is the full GTAA ontology (21MB, remote)
Repository.2.Name = GTAAremote
Repository.2.SesameServer = http://localhost:8080/openrdf-sesame/
Repository.2.RepositoryID = GTAA
Repository.2.Type = remote

# Repository 3 is the new RHCe ontology
Repository.3.Name = RHCeremote
Repository.3.SesameServer = http://localhost:8080/openrdf-sesame/
Repository.3.RepositoryID = RHCe
Repository.3.Type = remote

# Repository 4 is the small GTAA ontology
Repository.4.Name = GTAAsmall
Repository.4.SesameServer = http://localhost:8080/openrdf-sesame/
Repository.4.RepositoryID = Test
Repository.4.Type = remote


# Indexes to be used
# Lucene is used to index parts of the repositories. Running algorithms on indexes should
   have better performance

Index.0.Name = GTAAIndex
Index.0.RepositoryName = GTAAremote
Index.0.Analyzer = KeywordAnalyzer

Index.1.Name = GTAAIndexSmall
Index.1.RepositoryName = GTAAsmall
Index.1.Analyzer = KeywordAnalyzer

Index.2.Name = RHCeIndex
Index.2.RepositoryName = RHCeremote
Index.2.Analyzer = KeywordAnalyzer
#Query.GraphPattern = {x} tha:TermWaarde {y}, {x} rdf:type {tha:Trefwoord}
```

```
# Default settings
Default.Threshold = 0.1
Default.FeedbackNamespace = http://www.fb.mc
Default.ShowIntermediateResults = false
Default.ShowFinalResults = true
Default.ShowTimeTable = false
Default.ShowTotalTime = true
Default.PrintQuery = false


#Algorithm selection. One can select which type of algorithm to use, which specific
   algorithm, the treshold for the certainties, and which custom properties to follow for
   finding relevant labels.

# Algorithm 0: String matching (on index)
Algorithm.0.ClassName = nl.tue.matchingcomponent.StringMatch
Algorithm.0.MethodName = LevenshteinOnIndexStringMatch
Algorithm.0.Index = GTAAIndex
Algorithm.0.Query.GraphPattern = {x} p {y}
Algorithm.0.Query.Where =  p = skos:prefLabel or p = skos:altLabel
Algorithm.0.Query.ReturnConcept = x
Algorithm.0.Query.ReturnVar = y
Algorithm.0.Query.Namespace = gtaa=<http://www.beeldengeluid.nl/Thesaurus#>,
   rdfs=<http://www.w3.org/2000/01/rdf-schema#>, skos=<http://www.w3.org/2004/02/skos/
   core#>
Algorithm.0.Threshold = 0.65
Algorithm.0.Accuracy = 0.2
Algorithm.0.HandleResult = addreplace

# Algorithm 1: Semantic match: find semantical related concepts
Algorithm.1.ClassName = nl.tue.matchingcomponent.SemanticMatch
Algorithm.1.MethodName = MatchConcept
Algorithm.1.Repository = GTAAsmall
Algorithm.1.Deterioration = 0.1
Algorithm.1.Query.GraphPattern = {x} p {y} skos:prefLabel {z}
Algorithm.1.Query.Where =  p = skos:related or p = skos:narrower or p = skos:broader or p
   = gtaa:hasLinkedTerm or p = gtaa:hasClassificationCode
Algorithm.1.Query.ReplaceVar = x
Algorithm.1.Query.ReturnVar = z
Algorithm.1.Query.ReturnConcept = y
Algorithm.1.Query.Namespace = gtaa=<http://www.beeldengeluid.nl/Thesaurus#>,
   rdfs=<http://www.w3.org/2000/01/rdf-schema#>, skos=<http://www.w3.org/2004/02/skos/
   core#>
Algorithm.1.HandleResult = merge

# Algorithm 2: Convert ConceptTable to TagTable
```

```
Algorithm.2.ClassName = nl.tue.matchingcomponent.StandardActions
Algorithm.2.MethodName = ConvertConceptTableToTagTable
Algorithm.2.HandleResult = replace
```