



D2.2 Ontology learning implementation

Miha Grcar (Jozef Stefan Institute),
Dunja Mladenic (JSI), Marko Grobelnik (JSI),
Blaz Fortuna (JSI), Janez Brank (JSI)

Abstract

EU-IST Specific targeted research project (STREP) IST-2004-026460 TAO
Deliverable D2.2 (WP 2)

This report presents the theoretical background of the ontology-learning software implementation. The software was named LATINO which stands for “Link-analysis and text-mining toolbox”. At the current stage of development, LATINO serves mainly as a data pre-processing facility which generates OntoGen input files. Therefore this report also presents OntoGen, a system for semi-automatic data-driven ontology construction. OntoGen is an important part of the ontology-learning process in TAO as it complements LATINO with providing interaction with the user through its graphical user interface.

Keyword list: text mining, link analysis, graph, network, feature vector, software artefacts, ontology construction

WP2 Learning Web service ontologies

Nature: Report

Contractual date of delivery: 28 February 2007

Dissemination: PU

Actual date of delivery: 04 April 2007

TAO Consortium

This document is part of a research project partially funded by the IST Programme of the Commission of the European Communities as project number IST-2004-026460.

University of Sheffield

Department of Computer Science
Regent Court, 211 Portobello St.
Sheffield S1 4DP
UK
Tel: +44 114 222 1930
Fax: +44 114 222 1810
Contact person: Kalina Bontcheva
E-mail: K.Bontcheva@dcs.shef.ac.uk

Mondeca

3, cité Nollez
75018 Paris
France
Tel: +33 (0) 1 44 92 35 03
Fax: +33 (0) 1 44 92 02 59
Contact person: Jean Delahousse
E-mail: jean.delahousse@mondeca.com

University of Southampton

Southampton SO17 1BJ
UK
Tel: +44 23 8059 8343
Fax: +44 23 8059 2865
Contact person: Terry Payne
E-mail: trp@ecs.soton.ac.uk

Sirma Group Corp., Ontotext Lab

Office Express IT Centre, 5th Floor
135 Tsarigradsko Shosse Blvd.
Sofia 1784
Bulgaria
Tel: +359 2 9768 303
Fax: +359 2 9768 311
Contact person: Atanas Kiryakov
E-mail: naso@sirma.bg

Atos Origin Sociedad Anonima Espanola

Dept Research and Innovation
Atos Origin Spain, C/Albarracin, 25, 28037
Madrid
Spain
Tel: +34 91 214 8835
Fax: +34 91 754 3252
Contact person: Jaime García Sáez
E-mail: jaime.2.garcia@atosorigin.com

Dassault Aviation SA

DGT/DPR
78, quai Marcel Dassault
92552 Saint-Cloud
Cedex 300
France
Tel: +33 1 47 11 53 00
Fax: +33 1 47 11 53 65
Contact person: Farid Cerbah
E-mail: Farid.Cerbah@dassault-aviation.com

Jozef Stefan Institute

Department of Knowledge Technologies
Jamova 39
1000 Ljubljana
Slovenia
Tel: +386 1 477 3778
Fax: +386 1 477 3131
Contact person: Marko Grobelnik
E-mail: Marko.Grobelnik@ijs.si

Executive Summary

In this report we discuss the theoretical background, design, and initial implementation of the ontology-learning software architecture. The developed software is a more-or-less general data-mining framework that joins text mining and link analysis for the purpose of (semi-automated) ontology construction. The ontologies are constructed from the knowledge extracted from the data that accompany typical legacy applications. We introduce the term “application mining” which denotes the process of extracting this knowledge. In this deliverable we interpret application mining as being a combination of content and structure mining. In this context we also define software mining as being the process of extracting knowledge from software artefacts.

The implemented software was named LATINO which stands for “Link-analysis and text-mining toolbox”. An interface to LATINO is implemented as a Web service, which provides interoperability and platform independence. The reference manual for the LATINO Web-service interface is part of this deliverable (i.e. Appendix A). At its current stage of development, LATINO provides functions for working with the intermediate data layer, functions for basic graph/network operations, functions for generating feature vectors, and functions for generating OntoGen input files.

OntoGen (which stands for “Ontology Genesis”) is a system for semi-automatic data-driven ontology construction. It was initially developed in FP6 IP project SEKT¹ and is to be adapted for the purposes of TAO. It is an important part of the ontology-learning process in TAO as it complements LATINO with providing interaction with the user through its graphical user interface. OntoGen is presented in the second half of this report (i.e. in Section 5).

At the current stage, LATINO serves mainly as a data pre-processing facility which generates OntoGen input files. The long-term goal is to expose several data-mining algorithms that are used in OntoGen directly through LATINO so that developers can use them in their own way. These algorithms include hierarchical and “flat” clustering, active learning, learning, and classification.

The evaluation of the ontology construction methods has not yet been considered in this document. Basically, we can evaluate ontology learning methods either by comparing the resulting ontologies with a golden-standard ontology (if such ontology exists) or, on the other hand, by employing them in practice. In the second scenario, we measure the efficiency of the users that are using these ontologies (directly or indirectly) in order to achieve certain goals. The aspects on the quality of the methods presented herein will be covered in the next deliverable. The results of the evaluation will yield a methodology for using LATINO, suitable for the domain experts. At that time we will also train first users and evaluate the system from the perspectives of user friendliness and user efficiency.

¹ EU project Semantically Enabled Knowledge Technologies, EU IST IP 2003-506826, <http://www.sekt-project.com>.

Contents

TAO Consortium	3
Executive Summary	4
Contents	5
1 Introduction.....	6
2 The revised intermediate data representation	6
3 Mining content and structure of software artefacts	7
3.1 Textual content.....	9
3.2 Determining the structure	10
3.2.1 Comment reference graph.....	11
3.2.2 Name similarity graph.....	13
3.2.3 Type reference graph	15
3.2.4 Inheritance and interface implementation graph.....	17
4 Transforming content and structure into feature vectors.....	19
4.1 Converting content into feature vectors	19
4.2 Converting structure into feature vectors.....	19
4.3 Joining different representations into a single feature vector	22
5 OntoGen.....	24
5.1 OntoGen's main window	24
5.2 Creating and saving ontologies.....	24
5.3 Basic concept management.....	25
5.4 Concept suggestion	26
5.4.1 Unsupervised approach.....	28
5.4.2 Supervised approach	28
5.5 Ontology visualization	30
5.6 Concept documents management	30
5.7 Concept visualization.....	31
5.8 Relation management.....	32
5.9 Adding instances to an existing ontology	32
6 Conclusion	34
Bibliography and references	35
Appendix A: LATINO reference manual	37
A.1 Intermediate data layer management	37
A.1.1 Working with instances.....	37
A.1.2 Working with relations	38
A.1.3 Working with arcs.....	40
A.2 Algorithms	41
A.3 Dynamic objects.....	43
A.3.1 Sparse matrix	44
A.3.2 Sparse vector.....	44
A.4 Feature vectors	46
A.5 Input and output	48
Appendix B: Installing and running OntoGen.....	49
Appendix C: State transition probability computation.....	50

1 Introduction

Many data-mining (i.e. knowledge-discovery) techniques are these days in use for ontology learning – text mining, Web mining, graph mining, network analysis, link analysis, relation data mining, stream mining, and so on (see D2.1 [7]). Text mining seems to be a popular approach to ontology learning mainly for two reasons – there are many textual sources available out there (one of the biggest data sources in this context is the Web) and furthermore, text-mining techniques are shown to produce relatively good results. In the current state-of-the-art bundle there is a lack of “application-mining” techniques. With “application mining” we refer more to a methodology than to a concrete set of specialized algorithms. The term merely denotes the process of extracting knowledge (i.e. useful information) out of data sources that typically accompany a legacy application.

In this report we interpret the term “application mining” as being a combination of methods for structure mining and for content mining. To be more specific, we approach the application-mining task with the techniques used for text mining and link analysis (see Section 2). In Section 3, we explain many of the techniques in the light of mining the source code and the accompanying documentation where the assumption is that every legacy application can be transitioned into a regular (i.e. non-semantic) Web service. We introduce the term “software mining” to refer to this process. The TAO GATE case study serves as a perfect example in this perspective. We discuss how each instance (i.e. a piece of data) can be represented as a feature vector that combines the information about how the instance is interlinked with other instances, and the information about its (textual) content (see Section 4). The so-obtained feature vectors serve as the basis for the construction of the domain ontology with OntoGen. Section 5 thus presents OntoGen, a system for semi-automatic data-driven ontology construction.

2 The revised intermediate data representation

In D2.1 we proposed an intermediate data representation (see D2.1 [7], Section 4). It turned out that the suggested data structure is too rigid to support the more general cases where the documented source code is not available. Therefore, we decided to present the intermediate layer on a more abstract level.

The assumption we made when defining the intermediate layer in D2.1 was that every legacy application can be transitioned into a regular (i.e. non-semantic) Web service. This was taken for granted at least for the functional part of the application even if a graphical user interface had to be built on top of that². It was also assumed that a regular Web service comes with its source code and the accompanying documentation, and it is therefore convenient to interpret application mining as “software mining” (i.e. the process of extracting knowledge out of source code and the accompanying documentation). The TAO GATE case study (WP 6) is a reference scenario of such transition – GATE is open source, well documented, and has recently been wrapped into a set of regular Web services.

² Transitioning legacy applications to regular Web services is outside the scope of TAO as it is an engineering process rather than a research question.

The assumption that we are dealing with documented source code in the process of transitioning is not always true. The TAO Aviation case study (WP 7) is a good counterexample. In a real-life industrial environment (such as the Dassault corporation), the data sources are usually subjected to some legal constraints and may therefore not be available to the domain expert who is performing the transitioning process. In such cases the domain expert must work with the data that is available under the given circumstances. Therefore, rather than expecting the documented source code to be available, we interpret application mining as being a combination of **content and structure mining**. Such interpretation is more general and therefore covers a wider variety of transitioning scenarios.

With “content mining” we denote the process of extracting knowledge from text, images, audio, video, and other types of multimedia contents. In the scope of TAO we will be mainly focusing on extracting knowledge from textual contents hence we will be talking about **text mining**. “Structure mining”, on the other hand, is the process of using the methods from graph theory to analyze intra-relations within a document and interrelations between documents. In TAO we will be focusing on the latter so we will be mainly talking about **link-analysis techniques**. In short, we will be dealing with interlinked textual documents. A document is also said to be *an instance* (from the text-mining perspective³) or *a vertex* (from the link-analysis perspective). The links between vertices can be either *edges* (if they are undirected) or *arcs* (if they are directed). Each link can have a semantic meaning (see Sections 3.2.1–3.2.4 for some examples) and a weight denoting the strength of the relation between the two vertices.

3 Mining content and structure of software artefacts

Given a concrete TAO scenario, the first question we need to answer is – *what are the text-mining instances in this particular case and how do we describe them?* In other words – *how do we create a set of textual documents out of the data sources at hand?* It is impossible to answer this question in general – it depends on the available data sources. However, we can say something about this in the light of software artefacts.

In this section we present an illustrative example of data preprocessing from documented source code using the GATE case study data. In the context of the GATE case study the content is provided by the reference manual (textual descriptions of Java classes and methods), source code comments, programmer’s guide, annotator’s guide, user’s guide, forum, and so on. The structure is provided implicitly from these same data sources since a Java class or method is often referenced from the context of another Java class or method (e.g. a Java class name is mentioned in the comment of another Java class). Additional structure can be harvested from the source code (e.g. a Java class contains a member method that returns an instance of another Java class), and code snippets and usage logs (e.g. one Java class is often instantiated immediately after another). In this example we limit ourselves to the source code which also represents the reference manual (the so called *JavaDoc*) since the reference manual is generated automatically out of the source code comments by a documentation tool.

³ To avoid confusing the term “instance” in the text-mining sense with the term “instance” from the ontological perspective, we will talk about “text-mining instances” to emphasize the text-mining context.

A Web-service domain ontology provides two views on the corresponding software library: the view on the data structures and the view on the functionality [14]. In GATE, these two views are represented with Java classes and their member methods – these are evident from the GATE source code. In this example we limit ourselves to Java classes (i.e. we deal with the part of the domain ontology that covers the data structures of the system). This means that we will use the GATE Java classes as text-mining instances (and also as graph vertices when dealing with the structure).

First let us take a look at a typical GATE Java class (from here on we will refer to a GATE Java class simply as “a class”). It contains the following bits of information relevant for the understanding of this example⁴ (see also Figure 1):

- **Class comment.** It should describe the purpose of the class. It is used by the documentation tool to generate the reference manual (i.e. JavaDoc).
It is mainly a source of textual data but also provides structure – two classes are interlinked if the name of one class is mentioned in the comment of the other class.
- **Class name.** Each class is given a name that uniquely identifies the class. Usually a composed word that captures the meaning of the class.
It is mainly a source of textual data but also provides structure – two classes are interlinked if they share a common substring in their names.
- **Class modifiers.** These are flags that specify whether the class is public or private (a private class can not be used by the user but can be used by the software library to which it belongs), abstract (an abstract class can not be instantiated, it can only be inherited) or non-abstract.
We may want to exclude private and/or abstract classes from the process.
- **Field names and types.** Each class contains a set of member fields. Each field has a name (which is unique within the scope of the class) and a type. The type of a field corresponds to a Java class.
Field names provide textual data. Field types mainly provide structure – two classes are interlinked if one class contains a field that instantiates the other class.
- **Field comments.** A field can also be commented. The comment should explain the purpose of the field.
Field comments are a source of textual data. They can also provide structure in the same sense as class comments do.
- **Method names and return types.** Each class contains a set of member methods. Each method has a name, a set of parameters, and a return type. The return type of a method corresponds to a Java class. Each parameter has a name and a type which corresponds to a Java class.
Methods can be treated similarly to fields with respect to taking their names and return types into account. Parameter types can be taken into account similarly to return types but there is a semantic difference between the two pieces of information. Parameter types denote classes that are “used/consumed” for processing while return types denote classes that are “produced” in the process.
- **Method comments.** A method can also be commented. The comment should explain the functionality of the method.
Method comments are a source of textual data. They can also provide structure in the same sense as class and field comments do.

⁴ We expect the reader to be familiar with the basic computer programming terminology.

- **Field and method modifiers.** These are flags that specify whether a field (or a method) is public or private. Other modifiers are irrelevant for our purpose. *We may want to exclude private fields and/or methods from the process.*
- **Information about inheritance and interface implementation.** Each class inherits (fields and methods) from a base class. If the base class is not given explicitly, the class inherits from the Object class. Furthermore, a class can implement one or more interfaces. An interface is merely a set of methods that need to be implemented in the derived class. *The information about inheritance and interface implementation is a source of structural information.*

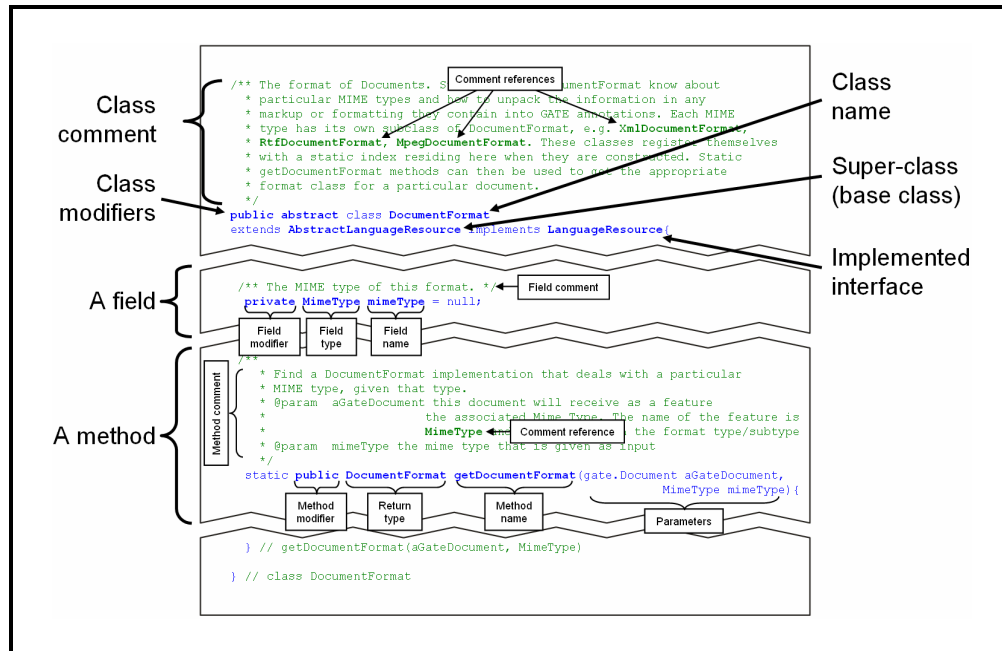


Figure 1: Relevant parts of a typical Java class.

3.1 Textual content

As already explained, in our example we use the GATE Java classes as text-mining instances. This means that we must assign a textual document to each GATE Java class (from here on we will refer to a GATE Java class simply as “a class”). Suppose we focus on a particular arbitrary class – there are several ways to form the corresponding document. Let us once more list those bits of information (from Figure 1) that can be used to compile such document:

- class comment,
- class name,
- field names,
- field comments,
- method names, and
- method comments.

It is important to include only those bits of text that are not misleading for the text-mining algorithms. At this point the details of these text-mining algorithms are pretty irrelevant provided that we can somehow evaluate their results – the domain ontology that we build in the end. The idea is to develop several (reasonable) rules (or “masks”) of what to include and what to leave out, and evaluate each of them in the given setting. The rule that would perform best would then become a kind of standard for the cases that have the same setting. The setting in the GATE case study is that we deal with well-documented Java source code. This is certainly not an isolated case – there are tons of open-source Java libraries available on the Web (see e.g. [15]). Since the evaluation may be costly in the sense of time and/or human effort, we can limit the number of combinations by resorting ourselves to common sense and to the work that other researchers have done when dealing with similar problems. Helm and Maarek, for instance, discuss a similar problem in the field of software repositories [9].

Another thing to consider is how to include composed names of classes, fields, and methods into a document. We can insert each of these as:

- a composed word (i.e. in its original form, e.g. “XmlDocumentFormat”),
- separate words (i.e. by inserting spaces, e.g. “Xml Document Format”), or
- combination of both (e.g. “XmlDocumentFormat Xml Document Format”).

The text-mining algorithms perceive two documents that have many words in common more similar than those that only share a few or no words. Breaking composed names into separate words therefore results in a greater similarity between documents that do not share full names but do share some parts of these names. To explain this on an example, let us consider two pairs of documents:

- “XmlDocumentFormat” and “RtfDocumentFormat”, and
- “Xml Document Format” and “Rtf Document Format”.

The first two documents have no words in common while the second two documents have two words in common (i.e. “Document” and “Format”). By counting common words we can easily conclude that the second two documents are more similar to each other than the first two even though both pairs were derived from the same two classes.

To conclude, it is important to include only those bits of text and to treat composed words in such a way that documents (i.e. instances) that should in the end belong to the same ontology concept, are perceived more similar than documents that should belong to separate concepts.

3.2 Determining the structure

The Java classes (from here on we will refer to a GATE Java class simply as “a class”) that we use as text-mining instances in the GATE case study are interlinked in many ways. In this section we discuss how this structure which is often implicit can be determined from the source code. Let us once more list those bits of information (from Figure 1) that can be used to determine links between classes:

- class, field, and method comments,
- class name,
- field types,
- method return types, and
- information about inheritance and interface implementation.

As already mentioned, when dealing with the structure, we represent each class (i.e. each text-mining instance) by a vertex in a graph. We can create several graphs – one for each type of links as follows. Links extracted from the comments are represented with the *comment reference graph*, links determined from the class names are represented with the *name similarity graph*, field types and method return types with the *type reference graph*, and the hierarchy of objects, typical for object-oriented architectures, with the *inheritance and interface implementation graph*. This section describes each of the graphs in details (Sections 3.2.1 through 3.2.4).

This approach allows us to decide post-hoc whether to include or exclude each of these graphs from the ontology-learning process. The intuition behind this is the same as with the bits of textual information discussed earlier – we need to include only those graphs that are not misleading for the ontology-learning algorithms. Strongly interlinked vertices (i.e. documents) are perceived more similar to each other by the ontology-learning algorithms than weakly interlinked vertices or vertices that are not interlinked at all. The strength of the link between two vertices is denoted by the corresponding arc weight as already mentioned earlier.

3.2.1 *Comment reference graph*

Every comment found in a class can reference another class by mentioning its name (for whatever the reason may be). In Figure 1 we can see four such references, namely the class *DocumentFormat* references classes *XmlDocumentFormat*, *RtfDocumentFormat*, *MpegDocumentFormat*, and *MimeType* (denoted with “Comment reference” in the figure). The comment reference graph for the GATE case study is shown in Figure 2. A section of the graph is enlarged in Figure 3. The vertices represent GATE Java classes found in the “gate” subfolder of the GATE source code repository (we limited ourselves to a subfolder merely to reduce the number of classes (i.e. vertices) for the purpose of the illustrative visualization). Vertices that share no arcs with other vertices (i.e. vertices of degree 0) were removed for the sake of the visualization. An arc that connects two vertices is directed from the source vertex towards the target vertex (these two vertices represent the source and the target class, respectively). The weight of an arc (at least 1) denotes the number of times the name of the target class is mentioned in the comments of the source class. The higher the weight, the stronger is the association between the two classes. In both figures, the thickness of an arc represents its weight; in Figure 3 you can also see the actual arc weight values.

3.2.2 Name similarity graph

A class usually represents a data structure and a set of methods related to it. Not every class is a data structure – it can merely be a set of (static) methods. The name of a class is usually a noun denoting either the data structure that the class represents (e.g. *Boolean*, *ArrayList*) or a “category” of the methods contained in the class (e.g. *System*, *Math*). If the name is composed (e.g. *ArrayList*) it is reasonable to assume that each of the words bears a piece of information about the class (e.g. an *ArrayList* is some kind of *List* with the properties of an *Array*). Therefore it is also reasonable to say that two classes that have more words in common are more similar to each other than two classes that have fewer words in common. According to this intuition we can construct the name similarity graph. This graph contains edges (i.e. undirected links) rather than arcs. Two vertices are linked when the two classes share at least one word. The strength of the link (i.e. the edge weight) can be computed as:

$$edge_weight = \frac{overlap}{name_len_1 + name_len_2 - overlap}, \quad (1)$$

where *overlap* is the number of words the two classes have in common, and *name_len_1* and *name_len_2* are the length of the name of the first class (in words) and the length of the name of the second class (in words), respectively. This is the so called *Jaccard similarity* formula and is often used to measure the similarity of two sets of items [17]. The name similarity graph for the GATE case study is presented in Figures 4 and 5. The vertices represent GATE Java classes found in the “gate” subfolder of the GATE source code repository. Equation (1) was used to weight the edges. Edges with weights lower than 0.6 and vertices of degree 0 were removed to simplify the visualization. In Figure 4 we have removed class names and weight values to clearly show the structure. The enlarged section presented in Figure 5 also shows class names and edge weight values.

String kernels. A more sophisticated technique for comparing composed words is by using the so called string kernels [5]. The main idea of string kernels is to compare composed words by the subsequences of words they contain – each possible subsequence defines one dimension of a feature vector which describes the composed word. For string kernels these subsequences do not need to appear contiguous in the composed word, but they receive different weighting according to the degree of contiguity. For example: the subsequence “AbstractResource” is present both in the composed word “AbstractResourcePersistence” and “AbstractProcessingResource” but with different weighting. The weighting depends on the length of the subsequence and the decay factor λ . In the previous example, the substring “AbstractResource” would receive weight λ^2 as part of “AbstractResourcePersistence” (as it “stretches” across *two* continuous words in the composed word) and λ^3 as part of “AbstractProcessingResource” (as it “stretches” across *three* continuous words in the composed word).

Edit distance. Another well-known method for comparing composed words is edit distance (also termed *Levenshtein distance*) [18]. Edit distance between two composed words is given by the minimum number of operations needed to transform one composed word into the other, where an operation is an insertion, deletion, or substitution of a single word.

3.2.3 Type reference graph

Field types and method return types are a valuable source of structural information. A field type or a method return type can correspond to a class in the scope of the study (i.e. a class that is also found in the source code repository under consideration) – hence an arc can be drawn from the class to which the field or the method belongs towards the class represented by the type. Figure 6 shows the type reference graph for the GATE case study. The vertices represent the classes found in the “gate” subfolder of the GATE source code repository. The enlarged section in Figure 7 also shows arc weights which are simply the number of times the target class is type-referenced from the source class.

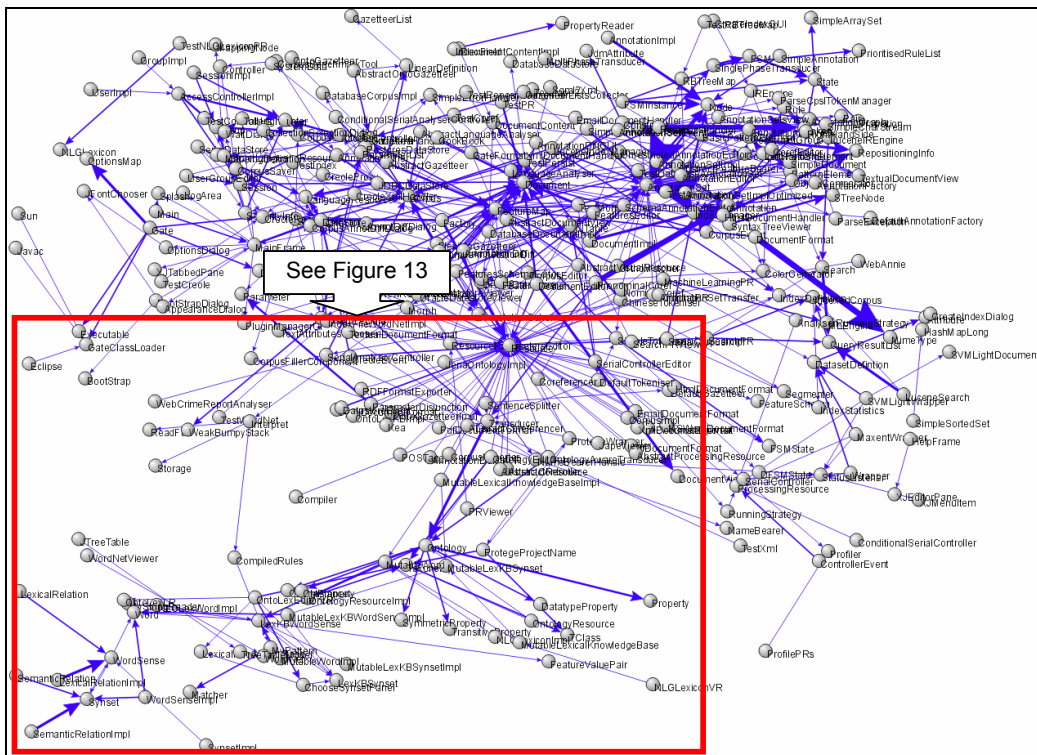


Figure 6: GATE type reference graph.

functions that use these data structures to perform certain operations (let us call these classes “processing classes”).

From the type reference graph we can tell data structures from processing classes by looking at the corresponding in- and out-degree values⁵. Classes that are represented with vertices that have high in-degree values are clearly being used (either as parameters, return values, or fields) by other classes and are thus the data structures. On the other hand, classes with high out-degree values are using several other classes (i.e. data structures) to perform certain operations and are thus the processing classes. According to the in- and out-degree values, we can distinguish between the following four types of classes (also shown in Figure 8):

- Low in-degree, low out-degree: We cannot say much about these classes except that they are poorly integrated (i.e. they do not interact much with other classes in the repository). Note that in our example we only considered one subfolder of the GATE source code repository. Classes that are deemed poorly integrated may in fact interact with classes outside the scope of the analysis.
- High in-degree, low out-degree: We can look at these classes as being the core data structures and data types of the system. They are being instantiated by other classes yet do themselves not instantiate other classes.
- High in-degree, high out-degree: These are the high-level data structures. They are being instantiated by other classes but they also depend on many low-level data structures.
- Low in-degree, high out-degree: These classes are the processing classes. They themselves are not instantiated by other classes but they use other classes to perform certain operations.

3.2.4 *Inheritance and interface implementation graph*

Last but not least, structure can also be determined from the information about inheritance and interface implementation. This is the most obvious structural information in an object-oriented source code and is often used to arrange classes into the browsing taxonomy. In this graph, an arc that connects two vertices is directed from the vertex that represents a base class (or an interface) towards the vertex that represents a class that inherits from the base class (or implements the interface). The weight of an arc is always 1. Figure 9 shows the inheritance graph for the GATE case study (we did not include the interface implementation information to simplify the graph for the purpose of the visualization). As before, the vertices represent the classes found in the “gate” subfolder of the GATE source code repository.

⁵ Note that the input/output domain size, and the input/output proximity prestige would carry the same semantics as the in-/out-degree but we limit ourselves to the simplest of the three measures here to explain the intuition.

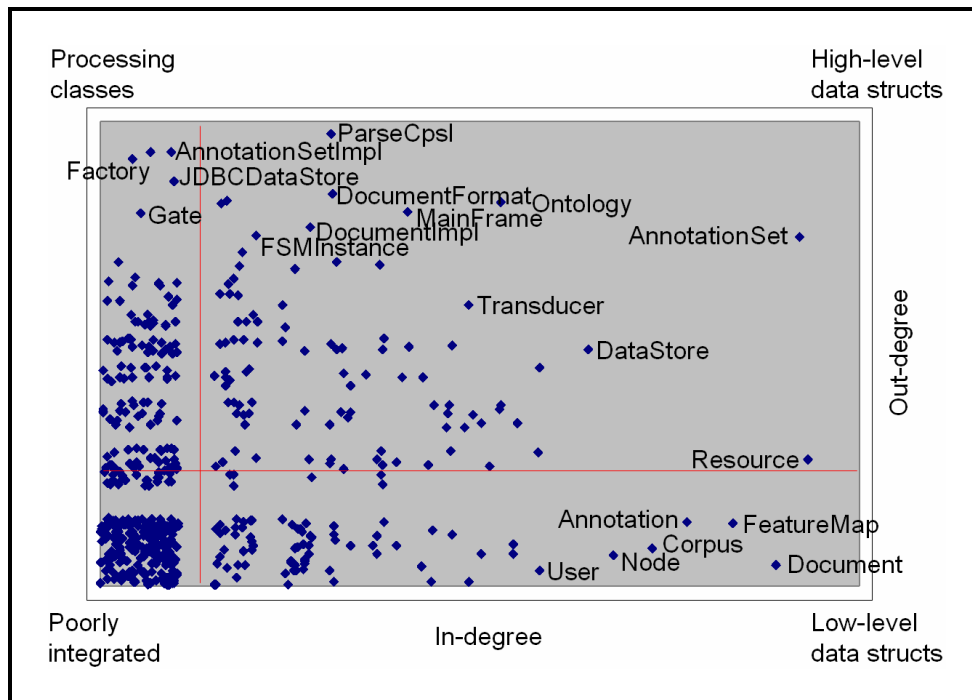


Figure 8: In- and out-degree scatter plot of GATE classes. For the purpose of this visualization we added some random noise to the data to prevent classes with the same in- and out-degrees to overlap. The chart is plotted on a log-log scale. The horizontal and the vertical red lines represent the average in-degree and the average out-degree value, respectively.

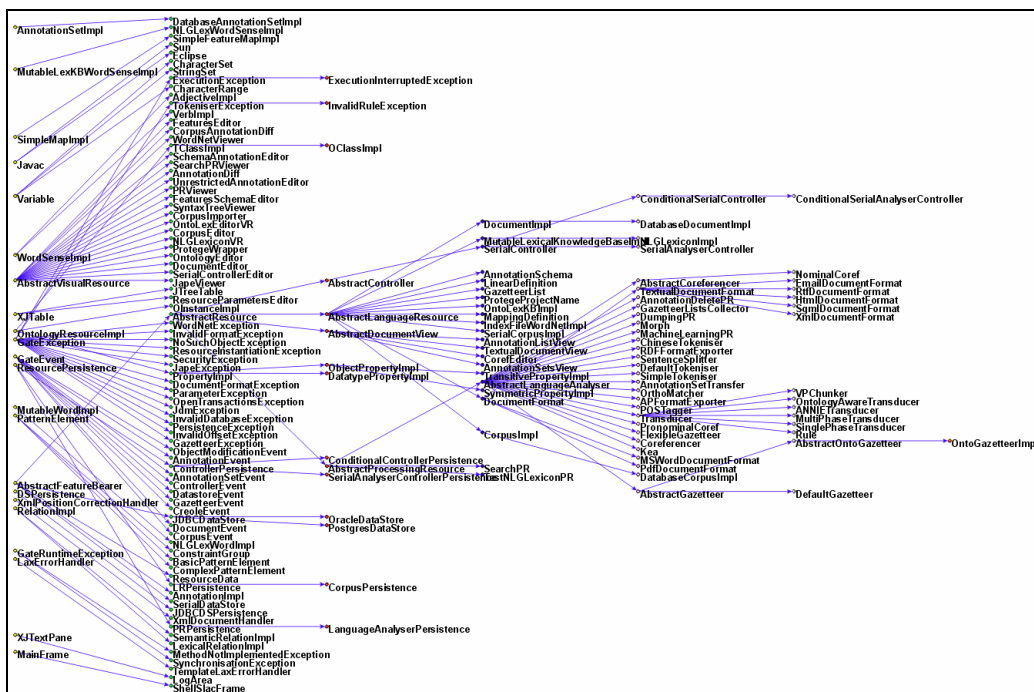


Figure 9: GATE inheritance graph (which is in fact a hierarchy).

4 Transforming content and structure into feature vectors

Once we have “enriched” the text-mining instances with textual documents and discovered the structure (in other words, once we have formed the intermediate data layer depicted in Figure 8 in D2.1), we are no longer bounded to a particular scenario. The methodology from this point onward is therefore general.

Many data-mining algorithms work with feature vectors. This is true also for the algorithms employed by OntoGen (see Section 5) and for the algorithms that we will implement as part of the TAO ontology-learning infrastructure. Therefore we need to convert the content (i.e. documents assigned to text-mining instances; see Section 4.1) and the structure (i.e. several graphs of interlinked vertices; see Section 4.2) into feature vectors. Potentially we also want to include other explicit features (e.g. in- and out-degree, input and output proximity prestige, and so on; see Section 3.2.3 for the description of these features).

4.1 Converting content into feature vectors

To convert textual documents into feature vectors we resort to a well-known text-mining approach. We first apply stemming⁶ to all the words in the document collection (i.e. we normalize words by stripping them of their suffixes, e.g. *stripping* → *strip*, *suffixes* → *suffix*). We then search for *n*-grams, i.e. sequences of consecutive words of length *n* that occur in the document collection more than a certain amount of times [11]. Discovered *n*-grams are perceived just as all the other (single) words. After that, we convert documents into their bag-of-words representations. To weight words (and *n*-grams), we use the TF-IDF weighting scheme (see D2.1, Section 1.3.2 for more details).

4.2 Converting structure into feature vectors

Let us repeat that the structure is represented in the form of several graphs in which vertices correspond to text-mining instances. If we consider a particular graph (e.g. the name similarity graph in the GATE case study), the task is to describe each vertex in the graph with a feature vector.

For this purpose we adopt the technique presented in [10]. First, we convert arcs (i.e. directed links) into edges (i.e. undirected links)⁷. The edges adopt weights from the corresponding arcs. If two vertices are directly connected with more than one arc, the resulting edge weight is computed by summing, maximizing, minimizing, or averaging the arc weights (we propose summing the weights as the default option). Then we represent a graph on *N* vertices as a $N \times N$ sparse matrix. The matrix is constructed so that the *X*th row gives information about vertex *X* and has nonzero components for the columns representing vertices from the neighborhood of vertex *X*.

⁶ We use the Porter stemmer for English [19].

⁷ This is not a required step but it seems reasonable – a vertex is related to another vertex if they are interconnected regardless of the direction. In other words, if vertex *A* references vertex *B* then vertex *B* is referenced by vertex *A*.

The neighborhood of a vertex is defined by its (restricted) domain. The *domain of a vertex* is the set of vertices that are path-connected to the vertex. More generally, a *restricted domain of a vertex* is a set of vertices that are path-connected to the vertex at a maximum distance of d_{max} steps. The X th row thus has a nonzero value in the X th column (because vertex X has zero distance to itself) as well as nonzero values in all the other columns that represent vertices from the (restricted) domain of vertex X . A value in the matrix represents the importance of the vertex represented by the column for the description of the vertex represented by the row. In [10] the authors propose to compute the values as $1/2^d$, where d is the path distance between the two vertices represented by the row and column. Figure 10 (borrowed from [10]) illustrates the graph transformation on a simple example.

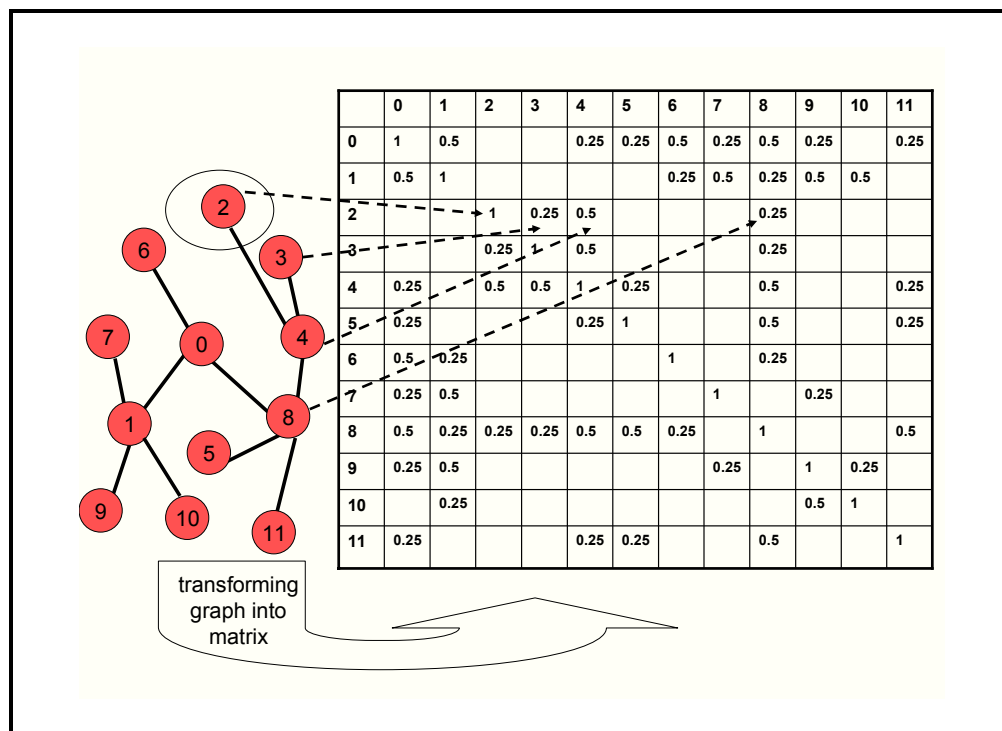


Figure 10: Illustration of the transformation process. The rows of the matrix represent the instances (vertices) and the columns represent the neighborhood with weights relative to the distance from the vertex in the corresponding row. Here we have set the maximal distance to $d_{max} = 2$. Notice that the diagonal elements all have weight 1 (showing that each vertex is in its own neighborhood). The dashed lines point out the neighboring vertices and the corresponding weights for the vertex labeled as 2. This vertex has four non-zero elements in its sparse vector representation: 1, 0.25, 0.5, 0.25. These elements correspond to the four neighboring vertices (labeled in the graph as 2, 3, 4, and 8).

The graph in Figure 10 is very simple and does not include cases in which two vertices are connected with more than one path. This also means that the graph does not contain any cycles. Furthermore, all the edges have the same weight (i.e. 1). Of course, this may not be the case. Therefore we need to decide how to take multiple paths and cycles into account and how to include edge weights into the matrix.

In order to resolve the issue of multiple paths and cycles we simply measure the distance according to the *shortest* path (also called the *geodesic distance* between two vertices). Consequently, if there is a cycle along the path between two vertices, we always bypass it to ensure taking the shortest path (alternatively we could traverse the cycle an arbitrary number of times).

Last but not least, we need to include edge weights into the matrix. The easiest way is to use the weights merely for thresholding. This means that we set a threshold and remove all the edges that have weights below this threshold. After that we construct the matrix which now contains the information about the weights (at least to a certain extent).

The ScentTrails algorithm. The simple approach described above is based on more sophisticated approaches such as ScentTrails [13]. The idea is to metaphorically “waft” scent of a specific vertex in the direction of its out-links (links with higher weights conduct more scent than links with lower weights). The scent is then iteratively spread throughout the graph. After that we can observe how much of the scent reached each of the other vertices. The amount of scent that reached a target vertex denotes the importance of the target vertex for the description of the source vertex. To compute the spreading of the scent, we first create the adjacency matrix \mathbf{A} . The matrix element t_{ij} denotes the weight of the arc (or the edge) from vertex i to vertex j , if such arc (or edge) exists, or is set to zero if such arc (or edge) does not exist. Next, we normalize the rows of matrix \mathbf{A} so that they sum to 1. In each iteration we then compute a *scent conduit matrix* $\mathbf{S}^{(t)}$. The scent conduit matrix element $s_{ij}^{(t)}$ denotes the amount of scent that reached vertex i from vertex j within t iterations⁸. The scent conduit matrix computation is done as follows:

$$\begin{aligned} \mathbf{S}^{(0)} &= \mathbf{I} , \\ \mathbf{S}^{(t)} &= \mathbf{I} + (1 - \beta) \cdot \text{zdiag}(\mathbf{A}^T \cdot \mathbf{S}^{(t-1)}) , \end{aligned} \quad (2)$$

where $\text{zdiag}(\dots)$ sets the diagonal entries of the matrix to zero (to prevent the scent being directed back to its origin), and β is the fraction of the scent intensity that is lost during the propagation through each link.

The ScentTrails algorithm shows some similarities with the probabilistic framework – the matrix rows are normalized (and can be perceived as probability distributions) and also Equation (2) shows similarities with equations used in the true probabilistic framework. Note however that the resulting matrix elements $s_{ij}^{(t)}$ do *not* denote the probability that the scent from vertex j reached vertex i within t iterations – in fact, the scent conduit matrix elements can be greater than 1. They indicate the aggregate width of the scent conduit between each pair of vertices.

The (true) probabilistic framework. The discussion on scent propagation has led us to the probabilistic framework. Starting in a particular vertex and moving along the arcs we need to determine the probability of ending up in a particular target vertex within m steps. At each step we can select one of the available outgoing arcs with the probability proportional to the corresponding arc weight (assuming that the weight denotes the strength of the association between the two vertices). In this context we

⁸ Note that the scent conduit matrix is transposed with respect to the adjacency matrix (therefore the columns represent source vertices and the rows represent target vertices).

can look at our graph as being a finite state machine with states $1, 2, \dots, n$. In this light we can define a random variable S_t which tells us the state of the machine at time t .

Let us now create the adjacency matrix \mathbf{A} . The matrix element a_{ij} denotes the weight of the arc (or the edge) from vertex i to vertex j , if such arc (or edge) exists, or is set to zero if such arc (or edge) does not exist. We then normalize the rows of matrix \mathbf{A} so that they sum to 1 (i.e. we transform the outgoing-arc weights of every vertex into a probability distribution). To put this more formally, a_{ij} now becomes:

$$a_{ij} = P(S_{t+1} = j \mid S_t = i) .$$

Let us denote the probability that a target state (denoted by c) is reached at least once in at most m steps with $q(c, m)$. $q(c, m)$ is computed as follows:

$$q(c, m) = q(c, m-1) + \sum_{j=1}^n r(c, j, m-1)a_{jc} ,$$

where $q(c, 0) = P(S_0 = c)$, and $r(c, j, m)$ is computed as follows:

$$r(c, j, m) = \begin{cases} 0 & c = j \\ b_j^{(0)} & c \neq j, m = 0 \\ \sum_{i=1}^n r(c, i, m-1)a_{ij} & c \neq j, m > 0 \end{cases} .$$

We formally derive and further explain these equations in Appendix C.

It is also possible to incorporate a “decay” parameter into the equations by extending the state machine with one additional state. The decay parameter represents the probability that the machine will stop at the current point in time. In Appendix C we explain how the decay parameter can be taken into account.

4.3 Joining different representations into a single feature vector

The next issue to solve is how to create a feature vector for a vertex that is present in several graphs at the same time (remember that the structure can be represented with more than one graph in the intermediate data layer) and how to then also “append” the corresponding content feature vector. In general, this can be done in two different ways:

- *Horizontally*. This means that feature vectors of the same vertex from different graphs are first multiplied by factors α_i ($i = 1, \dots, M$) and then concatenated into a feature vector with $M \times N$ components (M being the number of graphs and N the number of vertices). The content feature vector is multiplied by α_{M+1} and simply appended to the resulting structure feature vector.
- *Vertically*. This means that feature vectors of the same vertex from different graphs are first multiplied by factors α_i ($i = 1, \dots, M$) and then summed together (component-wise) resulting in a feature vector with N components (N being the

number of vertices). Note that the content feature vector cannot be summed together with the resulting structure feature vector since the features contained therein carry a different semantic meaning (not to mention that the two vectors are not of the same length). Therefore also in this case, the content feature vector is multiplied by α_{M+1} and appended to the resulting structure feature vector.

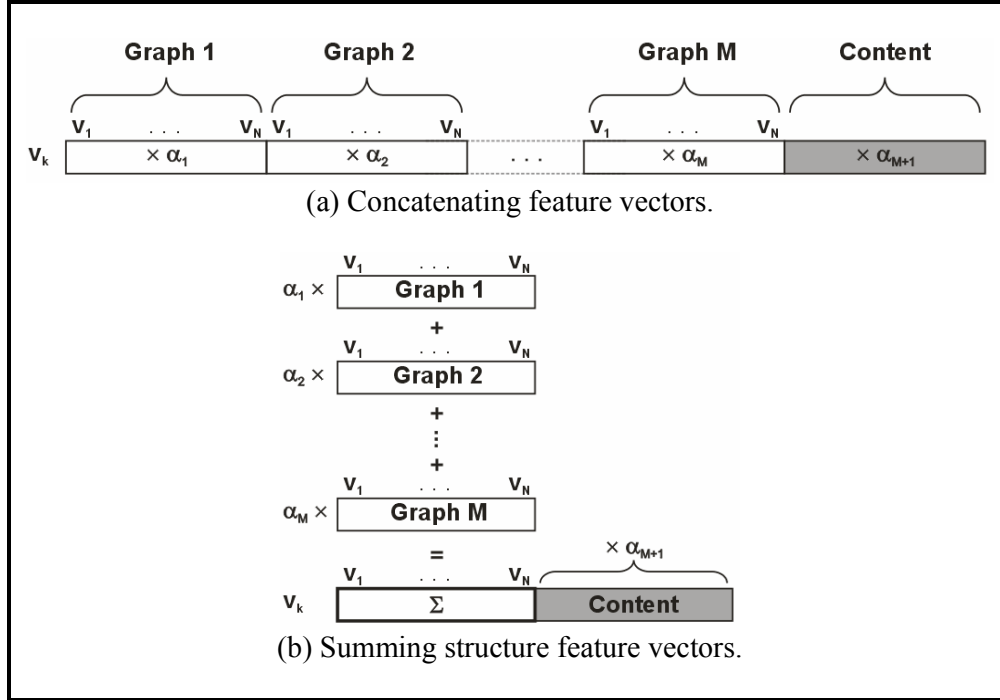


Figure 11: Two different ways of joining several different representations of the same instance.

Figure 11 illustrates these two approaches. A factor α_i ($i = 1, \dots, M$) denotes the importance of information provided by graph i , relative to the other graphs. Factor α_{M+1} , on the other hand, denotes the importance of information provided by the content relative to the information provided by the structure. The easiest way to set the factors is to either include the graph or the content (i.e. $\alpha_i = 1$), or to exclude it (i.e. $\alpha_i = 0$). In general these factors can be quite arbitrary. Pieces of information with lower factors contribute less to the outcomes of similarity measures used in clustering than those with higher factors. Furthermore, many classifiers are sensitive to this kind of weighting. For example, it has been shown in [2] that the SVM⁹ regression model is sensitive to how this kind of factors are set.

The computed feature vectors can be fed into OntoGen or used by a wide variety of data-mining algorithms (most obviously by those employed in OntoGen). The first version of our ontology-learning software focuses on preparing input data for OntoGen. OntoGen is an interactive system that facilitates the construction of the domain ontology out of a set of feature vectors.

⁹ SVMs (Support Vector Machines) are a set of supervised learning methods used for classification and regression [16].

5 OntoGen

OntoGen¹⁰ [4] is a SEKT¹¹ technology developed at JSI's Department of Knowledge Technologies. It is a system for *data-driven semi-automatic* ontology construction. Phrases “semi-automatic” and “data-driven” stand for:

- **Semi-automatic** – The system is an interactive tool that aids the user during the ontology construction process. The system suggests concepts, relations and their names, automatically assigns instances to concepts and provides a good overview of the ontology to the user through concept browsing and visualization. At the same time the user can fully adjust all the properties of the ontology by manually adding or deleting concepts, relations and reassigning instances.
- **Data-driven** – Most of the aid provided by the system (concept, relation suggestion, etc.) is based on some underlying data provided by the user at the beginning of the ontology construction. The data reflects the domain for which the user is building the ontology.

This chapter is two-fold: it first presents the OntoGen system from the end-user's perspective and then gives a tutorial on how to construct the domain ontology out of the GATE case study data step by step.

5.1 OntoGen's main window

The main window of OntoGen (Figure 12) is divided into three main areas. The largest part of the windows is dedicated to *ontology visualization* and *document management* part (the right side of the window). On the upper left side is the *concept tree* showing all the concepts from ontology and on the bottom left side is the area where the user can check details and manage properties of the selected concept and get suggestions for its sub-concepts.

5.2 Creating and saving ontologies

OntoGen supports several input formats for text instances (*Folder and Named Line-Documents*) and provides support for proprietary Text Garden format *Bag-Of-Words* as shown in Figure 13. Option *Folder* loads a set of HTML or text files stored in a given file folder and option *Named Line-Documents* loads documents from a given file where each line is interpreted as a document with the first word in the line being its title.

Ontologies created in OntoGen can be saved as Proton Topic, RDF Schema or OWL ontology.

¹⁰ OntoGen is freely downloadable from <http://ontogen.ijs.si>.

¹¹ EU project Semantically Enabled Knowledge Technologies, EU IST IP 2003-506826, <http://www.sekt-project.com>.

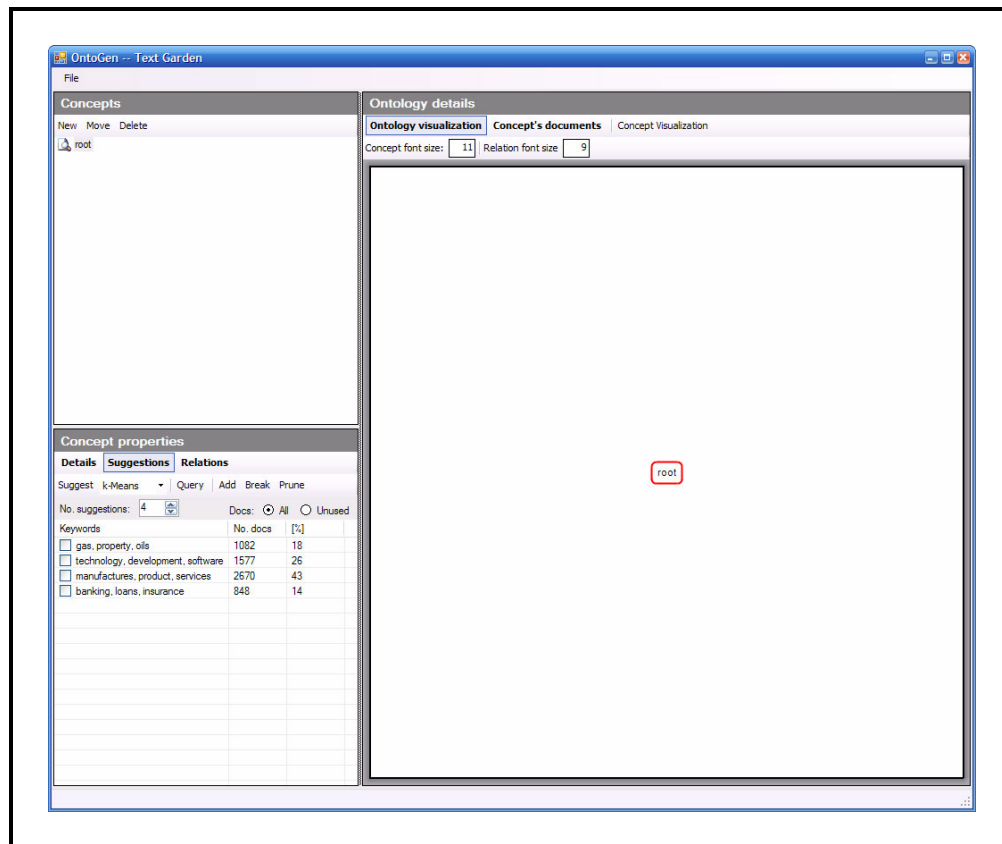


Figure 12: The main OntoGen window.

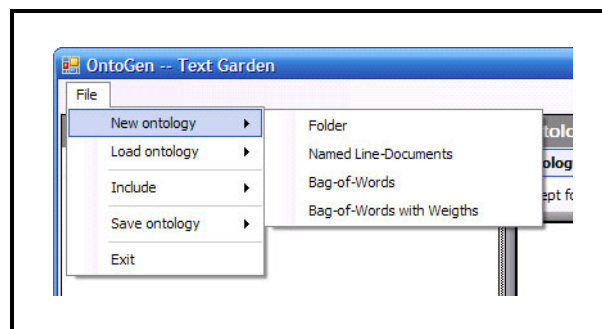


Figure 13: Input formats supported by OntoGen.

5.3 Basic concept management

The upper left side of the OntoGen window offers a place where the user can get a quick overview of all the concepts and how they are positioned in the concept hierarchy (left side of Figure 14). The selections here and in the ontology visualization are synchronized.

Here the user can create a new empty concept (instances must be manually added to the concept by the user), reposition an existing concept to a different position in the hierarchy or delete an existing concept from the ontology. Each of these tasks is

achieved by first selecting the concept in the concept tree and then clicking the button that corresponds to the desired action: *New*, *Move* or *Delete*. When moving the concept, the program pops up a dialog window asking the user to select the destination concept (right side of Figure 14).

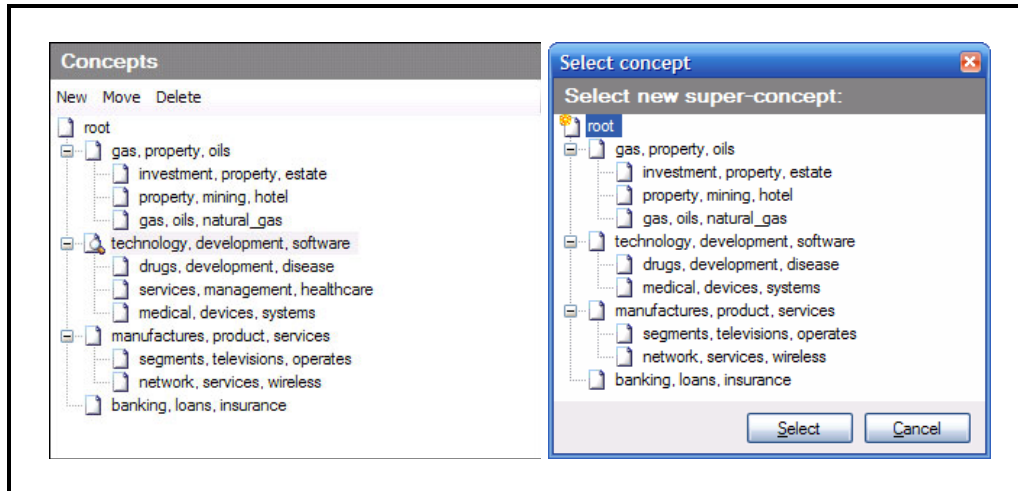


Figure 14: Concept hierarchy and concept selection.

In the bottom left part of the main window (Figure 15) the user can edit the name of the selected concept and check the concept's main keywords. There are two keyword extraction methods implemented in the system and both of them are presented. The first one, shown under *Keywords*, extracts the keywords from the concept's centroid vector and the second one, shown under *SVM Keyword*, extracts the keywords from the concept's SVM¹² linear model. SVM keywords are expensive to extract and are calculated only on user's request (with button *Calc*).

Information about the number of instances in the concept (All documents), number of instances not in any of the sub-concepts (Unused documents) and average inner-cluster similarity measure (Avg. similarity) are also presented here.

5.4 Concept suggestion

One of the main parts of the system is concept learning. In OntoGen this functionality is accessible under *Concept properties – Suggestions* tab that can be seen in Figure 16.

¹² SVMs (Support Vector Machines) are a set of supervised learning methods used for classification and regression [16].

The screenshot shows the 'Concept properties' window with three tabs: 'Details', 'Suggestions', and 'Relations'. The 'Details' tab is active. It contains the following information:

- Id:** 2
- Name:** technology, development. (with a 'Change' button)
- Keywords:** technology, development, software, services, solutions, management, systems, product, medical, drugs
- SVM Keywords:** software, development, technology, management, medical, solutions, information, pharmaceutical, research, services
- Buttons:** 'Calc' (below SVM Keywords)
- All documents:** 1577
- Unused documents:** 587
- Avg. similarity:** 0.238 (with a progress bar and a 'Calc' button)

Figure 15: Concept properties.

There are two different approaches implemented for concept learning: *unsupervised* and *supervised*. In the unsupervised approach the system provides suggestions for possible sub-concepts of the selected concept. In the supervised approach the user has an initial idea of what a sub-concept should be about and enters it into the system as a query.

There is a fundamental difference between the unsupervised and supervised methods. The main advantage of unsupervised methods is that it requires very little input from the user. The unsupervised methods provide well balanced suggestions for sub-concepts based on the instances and are also good for exploring the data.

The supervised method on the other hand requires more input. The user has to first figure out what should the sub-concept be, he has to describe the sub-concept through a query and go through the sequence of questions to clarify the query. This is intended for the cases where the user has a clear idea of the sub-concept he wants to add to the ontology but the unsupervised methods do not discover it.

The screenshot shows the 'Concept properties' window with the 'Suggestions' tab active. It displays a table of suggestions with the following data:

Keywords	No. docs	[%]
<input type="checkbox"/> drugs, development, disease	418	27
<input type="checkbox"/> services, management, healthcare	308	20
<input type="checkbox"/> software, solutions, technology	587	37
<input type="checkbox"/> medical, devices, systems	264	17

Additional interface elements visible in the screenshot include: 'Suggest k-Means', 'Query', 'Add', 'Break', 'Prune', 'No. suggestions: 4', and 'Docs: All (selected), Unused'.

Figure 16: Sub-concept suggestions.

5.4.1 *Unsupervised approach*

There are two clustering methods currently implemented in OntoGen: *k*-means and LSI [3]. There is also a suggestion method called *category* which just groups the instances according to the labels included in the input data. The user can select which method the system should use for generating suggestions.

The user can supervise the parameters for the methods (number of clusters) and on which documents should the clustering be performed (*All* for all documents in the concept and *Unused* for documents that are not already in any of the concept's sub-concepts).

After selecting the method and parameters the user can get sub-concept suggestions by clicking *Suggest* button. The clustering algorithm prepares suggestions and the program displays them in the list (see lower half of Figure 16). Each suggested sub-concept is described by the extracted main keywords (using the centroid method) and the size. A longer list of keywords is displayed when the user moves with mouse over the suggestion.

There are three follow-up tasks that the user can do with the suggestions. He can check the suggestions which he/she likes and add them to the ontology by clicking *Add* button. The suggestions are added as sub-concepts of the selected concept. He can replace the selected concept with suggested concepts by clicking *Break* button. This action removes the selected concept from the ontology and replaces it with the checked suggested concepts. All relations of the selected concept are redirected to/from the new concepts. Sometimes the system identifies a sub-concept for which the user thinks that should not be part of the concept. The user can decide to prune the suggested sub-concept (*Prune* button) from the selected concept which effectively removes suggested sub-concept's instances from the selected concept.

5.4.2 *Supervised approach*

A relatively new feature in OntoGen is a supervised method for adding concepts. It is based on SVM active learning method [12]. The querying and active learning is only applied to the instances from the selected concept (according to the parameters *All* or *Unused*).

The user can start this method by clicking *Query* button in Figure 16. The system then launches a dialog that takes the query from the user (left in Figure 17). After the user enters a query the active learning system starts asking questions and labelling the instances (right in Figure 17). On each step the system asks if a particular instance should belong to the concept and the user can select *Yes* or *No*.

Questions are selected so that the most information about the desired concept is retrieved from the user. After some initial labelled sample is collected from the user the system displays some additional information about the concept (Figure 18). It displays the current size (number of documents positively classified into the concept) and most important keywords for the concept (using SVM keyword extraction). The user can continue answering the questions or finish by clicking on the *Finish* button.

The more questions that the user answers the more reliable the assignment of instances to the final concept is.

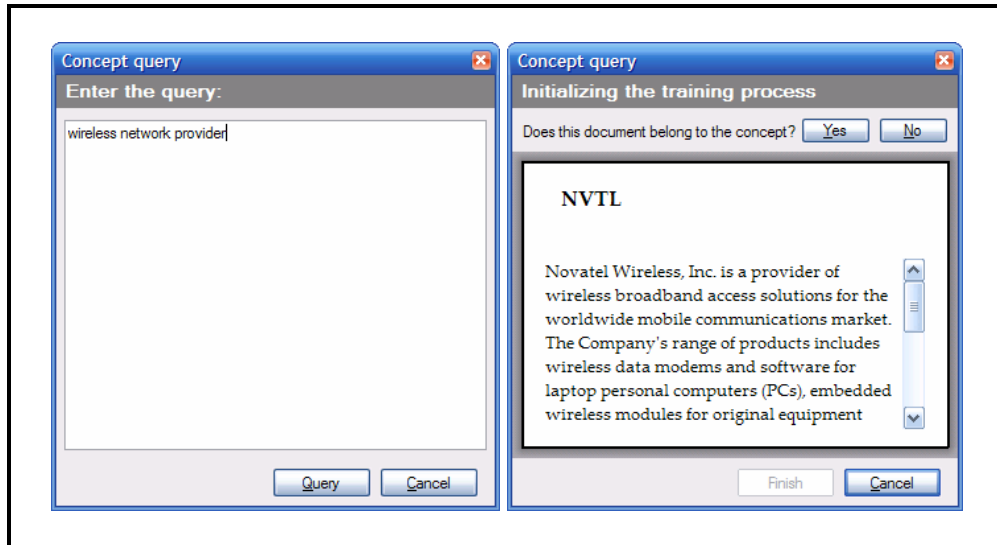


Figure 17: Active learning: initial query and question answering.

After the concept is constructed it is added to the ontology as a sub-concept of the selected concept.

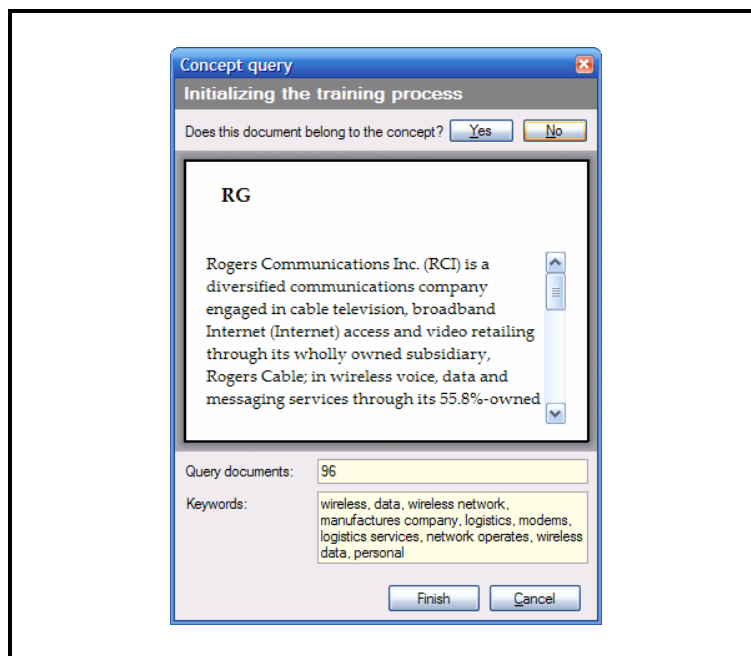


Figure 18: Information about the concept after several questions have been answered by the user in the active learning process.

5.5 Ontology visualization

While the user edits the ontology it is visualized in real time on the right side of the main OntoGen window (Figure 12). The root concept is displayed in the centre of the visualization pane; sub-concepts of the root concept are displayed in the circle around the root concept, and so on. The user can select a concept by clicking on it in the visualization. The currently selected concept is drawn in red colour, other concepts are drawn in blue colour and the relations are drawn in green colour (see Figure 19 for example of visualization).

5.6 Concept documents management

After a new concept is added to the ontology, either with unsupervised or supervised methods, the system automatically assigns instances to it. Since this assignment is often not perfect the user can manually reassign instances to and from the concept.

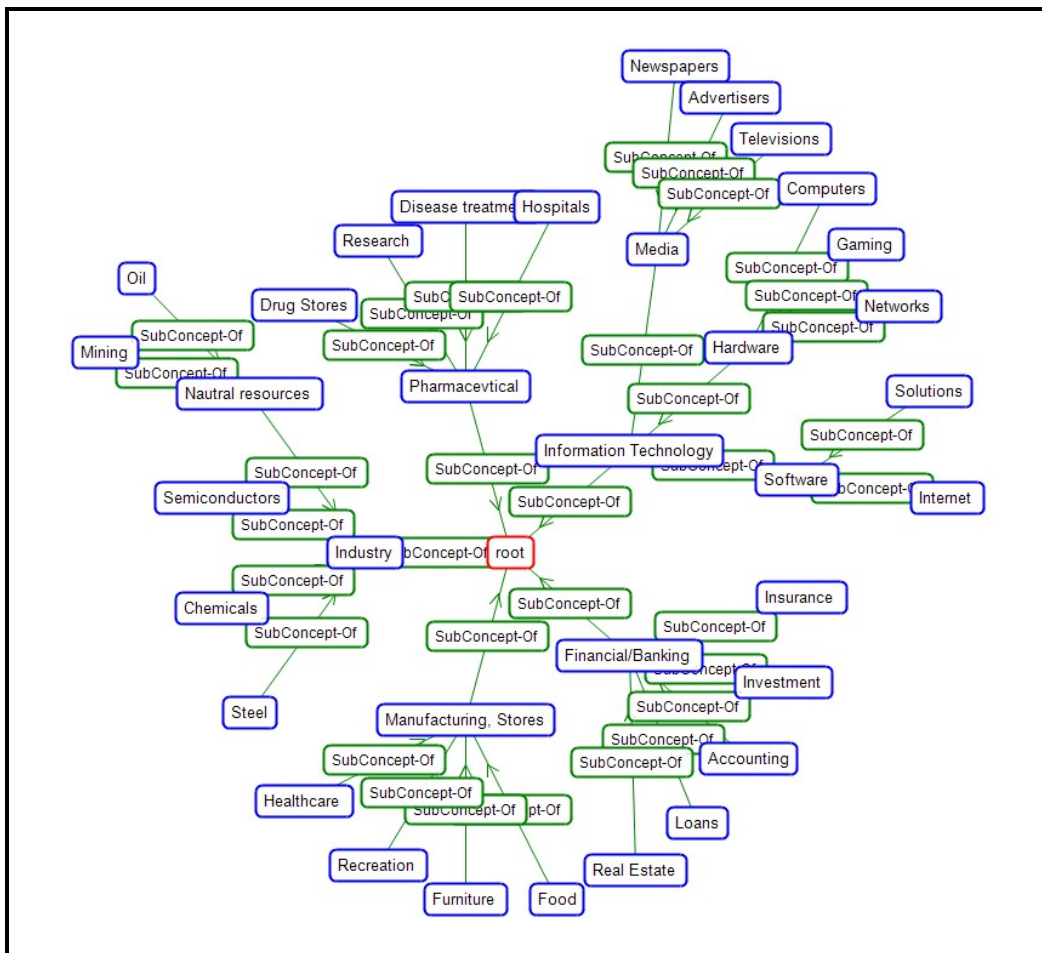


Figure 19: Ontology visualization in OntoGen.

This functionality is provided through the interface shown in Figure 20. On the left side is a list of all the instances. The instances from the concept are checked. Assignment of instances can be changed by checking or un-checking them. The user

must click *Apply* button at the end to confirm the changes. The content of an instance can be seen by selecting it from the list – the list is displayed on the right side.

The system calculates similarity between each instance in the list and the centroid vector of the concept (*cosine similarity* is used). The instances can be sorted according to similarity. At the bottom of the screen is a similarity graph with instances sorted according to similarity on the *x* axis and similarity on the *y* axis. Instances from the concept are shown in red colour. Similarity can be very practical when searching for outliers inside the concepts or for the instances that are not in the concepts but should be, considering their content. User can quickly add or remove instances from the concepts by simply selecting them on the graph with the mouse.

OntoGen also has functionality for detecting specific inconsistencies inside ontology. When an instance does not belong to a concept but it is in one of its sub-concepts, then it has red background in the list on all instances (for example see Figure 20). The user can, based on this information, assign the instance to the concept or remove it from its sub-concepts by right clicking on the instance and selecting *Remove from sub-concepts* from the menu.

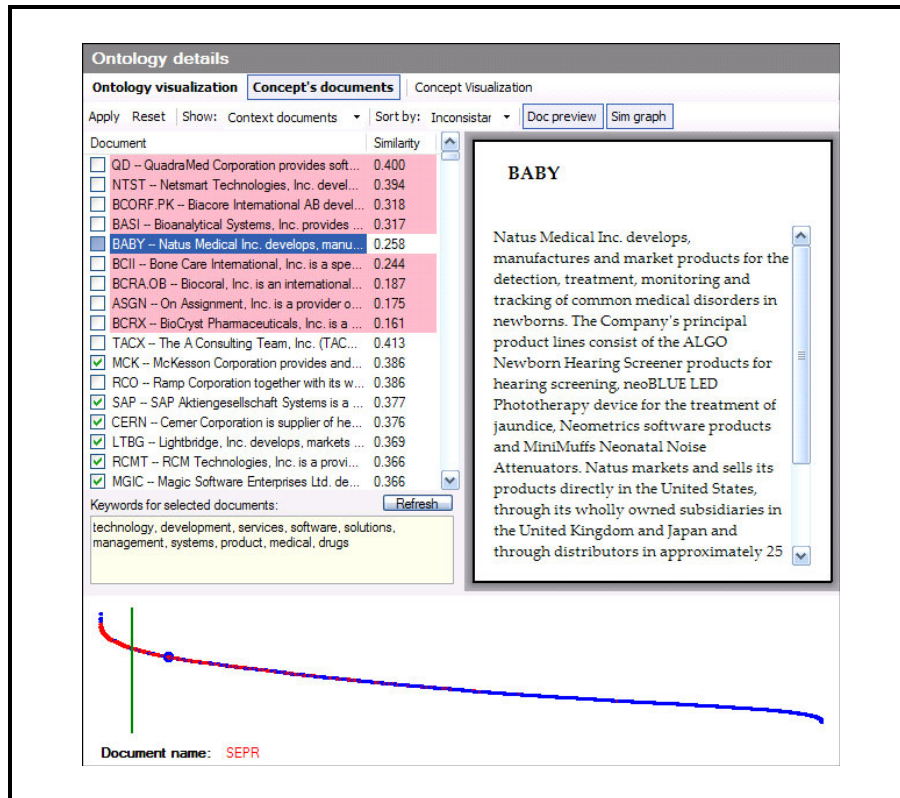


Figure 20: Concept documents management interface.

5.7 Concept visualization

Instances of selected concept can be visualized by clicking *Concept Visualization* button. Document Atlas tool is used for visualization [6].

5.8 Relation management

besides the *sub-concept of* relation, the user can also manage other types of relations. This is done through relation management screen shown in Figure 21. Here the user can see a list of all relations of the selected concept. He can add new relation, delete existing ones or add new types of relations.

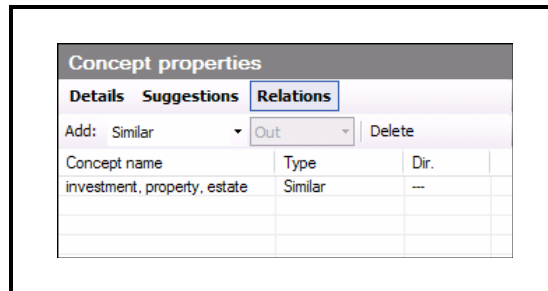


Figure 21: Relation management interface.

5.9 Adding instances to an existing ontology

The latest version of OntoGen also enables the user to add new instances to an existing ontology. The procedure is started by selecting a source of new instances in the menu *File* → *Include*. The system can load new instances from *Folder* or *Named Line-Documents*. Then the system trains SVM classifiers on the instances already arranged into ontology and uses them to classify new instances.

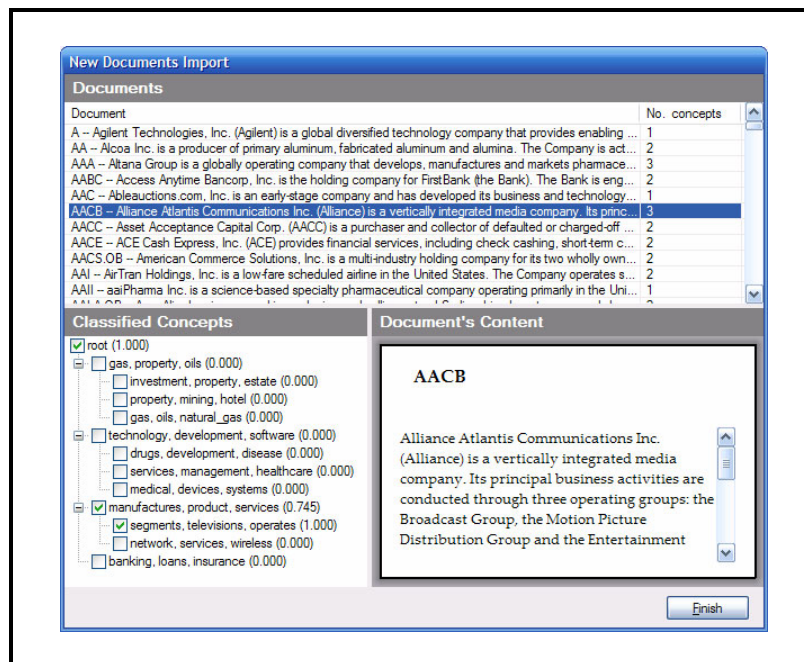


Figure 22: Newly imported instances and the classification results.

D2.2 / Ontology learning implementation

In the next step OntoGen presents to the user a list of all the newly imported instances and their classification results (Figure 22). User can check and correct classifications for each of the instances by first selecting the instance from the list and then checking the appropriate concepts in the concept tree. Preview of the selected instance is also displayed to aid the user. The instances are automatically added to the ontology after the user clicks *Finish*.

6 Conclusion

We have presented the theoretical background, design, and implementation of the ontology-learning software LATINO (Link-analysis and text-mining toolbox) which provides a Web-service interface to its functions. The reference manual for the LATINO Web-service interface is given in Appendix A. Currently, LATINO mostly provides functions for generating OntoGen input files. The long-term goal is to expose several data-mining algorithms that are used in OntoGen directly through LATINO so that developers can use them in their own way. These algorithms include – amongst others – hierarchical and “flat” clustering, active learning, learning, and classification.

The GATE adapter presented herein (see Section 3) was limited to the source code and to using Java classes for text-mining instances. In the course of TAO, we plan to extend the adapter to also include GATE methods (in order to support the part of the domain ontology that represents functionality of the (legacy) system) and potentially to take other types of data sources into account.

Currently, OntoGen does not support relation induction. We plan to work on this in the second project year. We will concentrate on providing relation-induction facilities through LATINO and – if deemed reasonable – through OntoGen as well.

The Dassault case study was not discussed with relation to LATINO and OntoGen in this report. We plan to thoroughly investigate the Dassault data sources and to put the case into perspective on bilateral bases in the second project year.

Last but not least, nothing has been said about evaluating LATINO (or the GATE adapter for that matter) in this report. We will consider this aspect in the second project year as the evaluation is important for – but not limited to – publishing our work at conferences and in journals. Basically, we can evaluate ontology learning methods either by comparing the resulting ontologies with a golden-standard ontology (if such ontology exists) or, on the other hand, by employing them in practice. The results of the evaluation will yield a methodology for using LATINO, suitable for the domain experts. This will enable us to train first users and evaluate the system from the perspectives of user friendliness and user efficiency.

Bibliography and references

- [1] Batagelj, V., Mrvar, A., de Nooy, W. (2004). Exploratory Network Analysis with Pajek. Cambridge University Press.
- [2] Brank, J., Leskovec, J. (2003). The Download Estimation task on KDD Cup 2003. In ACM SIGKDD Explorations Newsletter, Volume 5, Issue 2, Pages: 160–162, ACM Press, New York, NY, USA.
- [3] Deerwester, S., Dumais, S., Furnas, G., Landuer, T., Harshman, R., (2001). Indexing by Latent Semantic Analysis.
- [4] Fortuna, B, Grobelnik M., Mladenic D. (2006). Semi-automatic Data-driven Ontology Construction System. In Proceedings of the 9th International multi-conference Information Society IS-2006, Ljubljana, Slovenia.
- [5] Fortuna, B., Grobelnik, M., Mladenic, D. (2005). Classification of documents into hierarchy using string kernels. In 29th Annual Conference of the German Classification Society, March 9–11, 2005, Magdeburg.
- [6] Fortuna, B., Mladenic, D., Grobelnik, M. (2005). Visualization of Text Document Corpus. In Informatica 29, pages 497-502.
- [7] Grcar, M., Mladenic, D., Grobelnik, M., Bontcheva, K. (2006). D2.1: Data source analysis and method selection. Project report (IST-2004-026460 TAO).
- [8] Grobelnik, M., Mladenic, D. (2007). Text Mining Recipes. Springer-Verlag, Berlin; Heidelberg; New York (to appear), accompanying software available at <http://www.textmining.net> .
- [9] Helm, R., Maarek, Y. (1991). Integrating Information Retrieval and Domain Specific Approaches for Browsing and Retrieval in Object-Oriented Class Libraries. In Proceedings of Object-oriented Programming Systems, Languages, and Applications, pages 47–61, New York, USA. ACM Press.
- [10] Mladenic, D, Grobelnik, M. (2004). Visualizing very large graphs using clustering neighborhoods. In Local pattern detection, Dagstuhl Castle, Germany, April 12–16, 2004.
- [11] Mladenic, D., Grobelnik, M. (1998). Word sequences as features in text learning. In Proceedings of the 17th Electrotechnical and Computer Science Conference (ERK-98), Ljubljana, Slovenia.
- [12] Novak, B., (2004). Use of unlabeled data in supervised machine learning. Proceedings of the 7th International multi-conference Information Society IS-2004, Ljubljana: Institut "Jožef Stefan", 2004.
- [13] Olston, C., Chi, H. E. (2003). ScentTrails: Integrating browsing and searching on the Web. In ACM Transactions on Computer-Human Interaction (TOCHI), Volume 10, Issue 3, Pages: 177–197, ACM Press, New York, NY, USA.

[14] Sabou, M. (2004). From Software APIs to Web Service Ontologies: A Semi-Automatic Extraction Method. In Proceedings of the 3rd International Semantic Web Conference (ISWC'04).

[15] Java-source.net: A directory of open-source software in Java. At <http://java-source.net/> .

[16] Vapnik, V. (1998). Statistical Learning Theory. Wiley, New York.

[17] Wikipedia: Jaccard index. At http://en.wikipedia.org/wiki/Jaccard_index .

[18] Wikipedia: Levenshtein distance. At http://en.wikipedia.org/wiki/Levenshtein_distance .

[19] Wikipedia: Porter stemmer. At <http://de.wikipedia.org/wiki/Porter-Stemmer-Algorithmus> .

Appendix A: LATINO (Link-analysis and text-mining toolbox) reference manual

LATINO (Link-analysis and text-mining toolbox) is a reference implementation of the ontology-learning techniques presented in this report. In this appendix we provide the reference manual of the LATINO Web-service interface.

LATINO is partly based on Text Garden [8] and partly developed from scratch in .NET. It is expected to change and “grow” in the course of TAO thus this reference manual will be constantly updated. This implementation provides functions for working with the intermediate data layer, functions for basic graph/network operations, functions for generating feature vectors, and functions for generating OntoGen (and Pajek [1]) input files. In this appendix we describe each of the available functions in terms of its purpose, parameters, and return value.

A.1 Intermediate data layer management

A.1.1 Working with instances

int AddInstance (string name, string document)
Adds a new instance into the intermediate data layer.
Parameters: name: The name of the new instance. document: The textual document associated with the new instance.
Return value: non-negative integer: The id of the newly added instance. -1: Invalid parameter values (name is null).
Status: Implemented.

bool IsInstance (string name)
Checks if the instance with the given name already exists in the intermediate data layer.
Parameters: name: The name of the instance.
Return value: true: The instance exists. false: The instance does not exist.
Status: Implemented.

int GetInstanceId (string name)
Returns the id of the instance specified by the name.
Parameters: name: The name of the instance.
Return value: non-negative integer: The instance id. -1: The instance does not exist.
Status: Implemented.

int GetInstanceCount()
Returns the number of instances currently in the intermediate data layer. Together with GetInstanceName(...) this function can be used to traverse the list of instances. Note that the index of an instance in the list corresponds to the id of that instance.
Return value: non-negative integer: The number of instances currently in the intermediate data layer
Status: Implemented.

bool SetDocument(int instanceId, string document)
Assigns a textual document to the specified instance.
Parameters: instanceId: The id of the target instance. This id can be obtained by calling GetInstanceId(...). document: The textual document to append to the instance.
Return value: true: Success. false: The instance does not exist.
Status: Implemented.

string GetDocument(int instanceId)
Returns the textual document belonging to the specified instance.
Parameters: instanceId: The id of the instance. This id can be obtained by calling GetInstanceId(...).
Return value: non-null: The document assigned to the specified instance. null: The instance does not exist.
Status: Implemented.

string GetInstanceName(int instanceId)
Returns the name of the specified instance.
Parameters: instanceId: The id of the instance. This id can be obtained by calling GetInstanceId(...).
Return value: non-null: The name of the instance. null: The instance does not exist.
Status: Implemented.

A.1.2 Working with relations

In Sections 3.2.1–3.2.4 we presented several graphs of structural information. These graphs were all extracted from the same data source. In LATINO, such set of graphs is represented with one single graph that has semantic meanings assigned to its arcs (or edges). These semantic meanings are called “relations”.

int RegisterRelation (string name)
Registers a new relation with the specified name.
Parameters: name: The name of the relation.
Return value: positive integer: The relation was added – the returned value is the relation -1: Invalid parameter value (relationName is null).
Status: Implemented.

bool IsRelation (string name)
Checks if the specified relation already exists.
Parameters: name: The name of the relation.
Return value: true: The relation exists. false: The relation does not exist.
Status: Implemented.

int GetRelationId (string name)
Returns the identifier of the specified relation.
Parameters: name: The name of the relation.
Return value: positive integer: The relation identifier. -1: The relation does not exist.
Status: Implemented.

int GetRelationCount ()
Returns the number of currently registered relations.
Return value: non-negative integer: The number of registered relations.
Status: Implemented.

string GetRelationName (int id)
Returns the name of the specified relation.
Parameters: id: The identifier of the relation.
Return value: non-null: The name of the relation. null: The relation does not exist.
Status: Implemented.

A.1.3 Working with arcs

<p>bool AddArc(int relationId, int instanceId1, int instanceId2, double weight) Adds a new arc to which the specified relation is assigned, over the specified instances, and with the specified weight.</p>
<p>Parameters:</p> <ul style="list-style-type: none"> relationId: The relation assigned to the arc. instanceId1: The identifier of the source instance. instanceId2: The identifier of the target instance. weight: The weight assigned to the arc.
<p>Return value:</p> <ul style="list-style-type: none"> true: The arc was added successfully. false: Either the relation or (one of) the two instances does/do not exist.
<p>Status: Implemented.</p>

<p>bool IsArc(int relationId, int instanceId1, int instanceId2) Checks if the specified arc exists.</p>
<p>Parameters:</p> <ul style="list-style-type: none"> relationId: The relation assigned to the arc. instanceId1: The identifier of the source instance. instanceId2: The identifier of the target instance.
<p>Return value:</p> <ul style="list-style-type: none"> true: The arc exists. false: The arc does not exist.
<p>Status: Implemented.</p>

<p>bool SetArcWeight(int relationId, int instanceId1, int instanceId2, double weight) Assigns a new weight to the specified arc.</p>
<p>Parameters:</p> <ul style="list-style-type: none"> relationId: The relation assigned to the arc. instanceId1: The identifier of the source instance. instanceId2: The identifier of the target instance. weight: The weight to be assigned to the specified arc.
<p>Return value:</p> <ul style="list-style-type: none"> true: The weight was updated successfully. false: The specified arc does not exist.
<p>Status: Implemented.</p>

<p>double GetArcWeight(int relationId, int instanceId1, int instanceId2) Returns the weight associated with the specified arc.</p>
<p>Parameters:</p> <ul style="list-style-type: none"> relationId: The relation assigned to the arc. instanceId1: The identifier of the source instance. instanceId2: The identifier of the target instance.
<p>Return value:</p> <ul style="list-style-type: none"> valid double: The weight associated with the specified arc. NaN: The arc does not exist.
<p>Status: Implemented.</p>

A.2 Algorithms

int InvertArcs(int[] relationIds) Changes the direction of arcs that represent the specified relations.
Parameters: relationIds: The list of relations.
Return value: 0: Success.
Status: Implemented.

int AddInvertedArcs(int[] relationIds, int addMethod) Extends the graph so that the arcs that correspond to the specified relations become bidirectional (this corresponds to edges in LATINO). If an arc is already bidirectional (i.e. consists of two arcs pointing in opposite directions) then double arcs will appear when invoking AddInvertedArcs(...). Each double arc will be converted into a single arc using one of the methods for computing the new arc weight – summing the two weights, computing the average of the two weights, taking only the lower weight into account, or taking only the higher weight into account.
Parameters: relationIds: The list of relations. addMethod: The method for dealing with multiple arcs. 0: Sum the two weights together. 1: Compute the average of the two weights. 2: Take only the lower weight into account. 3: Take only the higher weight into account.
Return value: 0: Success.
Status: Implemented.

int GetOutDegree(int[] relationIds) Returns the id of a sparse vector that contains out-degrees of all the vertices in the graph. Only the arcs that correspond to the specified relations are taken into account. If a value is missing in the resulting sparse vector, the out-degree of the corresponding vertex is 0.
Parameters: non-negative integer: The identifier of the returned sparse vector.
Return value: 0: Success.
Status: Implemented.

int GetInDegree(int[] relationIds) Returns the id of a sparse vector that contains in-degrees of all the vertices in the graph. Only the arcs that correspond to the specified relations are taken into account. If a value is missing in the resulting sparse vector, the in-degree of the corresponding vertex is 0.
Parameters: relationIds: The list of relations.
Return value: non-negative integer: The identifier of the returned sparse vector.
Status: Implemented.

int GetNeighborhoodMatrix(int[] relationIds, int maxDistance) Returns the id of a sparse matrix that contains factors $1/2^d$ for the entire graph (see Section 4.2).
Parameters: relationIds: The list of relations. maxDistance: The maximum distance from the source vertex, d_{max} (see Section 4.2).
Return value: non-negative integer: The identifier of the returned sparse matrix.
Status: Implemented.

int GetNeighborhood(int instanceId, int[] relationIds, int maxDistance) Returns the id of a sparse vector that contains factors $1/2^d$ for the specified vertex/instance.
Parameters: instanceId: The instance. relationIds: The list of relations. maxDistance: The maximum distance from the specified vertex, d_{max} .
Return value: non-negative integer: The identifier of the returned sparse vector.
Status: Implemented.

int GetScentConduitMatrix(int[] relationIds, double scentDecay) Returns the id of the scent conduit matrix for the entire graph (see the ScentTrails algorithm in Section 4.2).
Parameters: relationIds: The list of relations. scentDecay: The scent decay parameter, β (see the ScentTrails algorithm in Section 4.2).
Return value: non-negative integer: The identifier of the returned sparse matrix.
Status: Not implemented yet.

int GetScentConduit (int instanceId, int[] relationIds, double scentDecay) Returns the id of the scent conduit vector for the specified vertex/instance.
Parameters: instanceId: The instance. relationIds: The list of relations. scentDecay: The scent decay parameter, β .
Return value: non-negative integer: The identifier of the returned sparse vector.
Status: Not implemented yet.

int GetStateProbabilityMatrix (int[] relationIds, int numStateTransitions, double stopProbability) Returns the id of the state-transition probability matrix for the entire graph (see the probabilistic framework in Section 4.2/Appendix C).
Parameters: relationIds: The list of relations. numStateTransitions: Maximum number of state transitions. stopProbability: The probability that the machine stops in its current state, β (see the probabilistic framework in Section 4.2/Appendix C)
Return value: non-negative integer: The identifier of the returned sparse matrix.
Status: Not implemented yet.

int GetStateProbability (int instanceId, int[] relationIds, int numStateTransitions, double stopProbability) Returns the id of the state-transition probability vector for the specified vertex/instance.
Parameters: instanceId: The instance. relationIds: The list of relations. numStateTransitions: Maximum number of state transitions. stopProbability: The probability that the machine stops in its current state, β .
Return value: non-negative integer: The identifier of the returned sparse vector.
Status: Not implemented yet.

A.3 Dynamic objects

bool DeleteObject (int objectId) Deletes the specified dynamic object.
Parameters: objectId: The identifier of the object.
Return value: true: The object was deleted. false: The object was not found.
Status: Implemented.

A.3.1 *Sparse matrix*

int GetNumNonZeroSparseMatrixRows(int sparseMatrixId) Returns the number of non-zero rows in the sparse matrix.
Parameters: sparseMatrixId: The identifier of the matrix.
Return value: non-negative integer: The number of non-zero rows in the sparse matrix. -1: The sparse matrix was not found.
Status: Implemented.

int GetSparseMatrixRowIndex(int sparseMatrixId, int nonZeroRowSeqNum) Returns the index of the sparse matrix row which is specified by the sequential number.
Parameters: sparseMatrixId: The identifier of the matrix. nonZeroRowSeqNum: The sequential number of the non-zero row.
Return value: non-negative integer: The index of the sparse matrix row which is specified by the sequential number. -1: The sparse matrix was not found.
Status: Implemented.

int GetSparseMatrixRow(int sparseMatrixId, int rowIndex) Returns the identifier of the returned sparse vector – the sparse vector represents the specified sparse matrix row.
Parameters: sparseMatrixId: The identifier of the matrix. rowIndex: The index of the matrix row.
Return value: non-negative integer: The identifier of the returned sparse vector – the sparse vector represents the specified sparse matrix row. -1: The sparse matrix was not found. -2: The specified sparse matrix row is empty, i.e. consists of zeros.
Status: Implemented.

A.3.2 *Sparse vector*

int GetNumNonZeroSparseVectorElements(int sparseVectorId) Returns the number of non-zero elements in the sparse vector.
Parameters: sparseVectorId: The identifier of the sparse vector.
Return value: non-negative integer: The number of non-zero elements in the specified sparse vector. -1: The sparse vector was not found.
Status: Implemented.

int GetSparseVectorElementIndex (int sparseVectorId, int nonZeroElementSeqNum) Returns the index of the element specified by the sequential number.
Parameters: sparseVectorId: The identifier of the sparse vector.
Return value: non-negative integer: The index of the element specified by the sequential number. -1: The sparse vector was not found.
Status: Implemented.

double GetSparseVectorElement (int sparseVectorId, int index) Returns the specified sparse vector value.
Parameters: sparseVectorId: The identifier of the sparse vector. index: The index of the element to retrieve.
Return value: valid double: The requested sparse vector value. NaN: The sparse vector was not found.
Status: Implemented.

int GetSparseVectorLastNonZeroElementIndex (int sparseVectorId) Returns the index of the last non-zero element in the sparse vector.
Parameters: sparseVectorId: The identifier of the sparse vector.
Return value: non-negative integer: The index of the last non-zero element in the sparse vector. -1: The sparse vector was not found.
Status: Implemented.

A.4 Feature vectors

int **CreateBows**(bool excludeStopWords, bool applyStemming, int maxNGramLen, int minNGramFreq, double cutWordWgtSumPerc, int minWordFreq)
 Created a TF-IDF bag-of-words representations of vertices/instances (see Section 4.1).

Parameters:

excludeStopWords: If set to true, English stop words are ignored.
 applyStemming: If set to true, Porter stemming for English is applied.
 maxNGramLen: The maximum length of extracted n -grams.
 minNGramFreq: The minimum frequency of extracted n -grams. n -grams with lower frequencies are ignored.
 cutWordWgtSumPerc: Sets the threshold for word weights. Words with low weights are excluded from the TF-IDF vector.
 minWordFreq: The minimum word frequency. Words with lower frequencies are ignored.

Return value:

0: Success.

Status: **Implemented.**

int **DeleteBows**()

Deletes the bag-of-words representations of vertices/instances.

Return value:

0: Success.

Status: **Implemented.**

int **GetInstanceBow**(int instanceId)

Returns the TF-IDF bag-of-words representation of the specified vertex/instance.

Parameters:

instanceId: The instance.

Return value:

non-negative integer: The identifier of the sparse vector that represents the TF-IDF bag-of-words representation of the specified vertex/instance.
 -1: Bag-of-words representations do not exist. CreateBows(...) needs to be invoked prior to this function.

Status: **Implemented.**

int **GetNumWords**()

Returns the number of words in the vocabulary. Each TF-IDF bag-of-words vector has the size of the vocabulary and the features in the vector represent the words from the vocabulary. To determine which word corresponds to which feature, first invoke $n \leftarrow \text{GetNumWords}()$ to get the length of TF-IDF vectors and then invoke $\text{GetWord}(i)$, $i=1\dots n$, to get the word that corresponds to the i -th feature in the vector.

Return value:

non-negative integer: The number of words in the vocabulary.

Status: **Implemented.**

string GetWord (int wordSeqNum) Returns the word that corresponds to the given sequential number.
Parameters: wordSeqNum: The sequential number of the word.
Return value: string: The word from the vocabulary. null: The given sequential number is out of bounds.
Status: Implemented.

int GetFeatureVectors (int[] relationIds, int typeOfStructFeatures, int m, int beta, double[] weights, int structJoinMethod, double contentWeight) Returns the sparse matrix of feature vectors. The feature vectors are computed as explained in Section 4.2 and combined as explained in Section 4.3.
Parameters: relationIds: The list of relations to take into account. typeOfStructFeatures: Determines the method that is used to compute the structure feature vectors. 0: Computes factors $1/2^d$. 1: Computes the scent conduit vectors. (Not supported yet.) 2: Computes the state transition probabilities. (Not supported yet.) m: The maximum distance (if typeOfFeatures is set to 0) or the number of steps/state transitions (if typeOfFeatures is set to 1 or 2). beta: The scent decay parameter (if typeOfFeatures is set to 1) or the machine-stop probability (if typeOfFeatures is set to 2). This parameter is ignored if typeOfFeatures is set to 0. weights: Represents the weights $\alpha_1 \dots \alpha_M$ (see Section 4.3). structJoinMethod: Determines the method that is used to join different structure representations into a single feature vector (see Section 4.3). 0: Summing the feature vectors component-wise. 1: Concatenating the feature vectors. contentWeight: Represents the weight α_{M+1} (see Section 4.3).
Return value: non-negative integer: The identifier of the sparse matrix that contains feature vectors. -1: typeOfFeatures parameter out of bounds. -2: Incorrect number of weights. -3: joinMethod parameter out of bounds. -4: Bag-of-words representations do not exist. CreateBows(...) needs to be invoked prior to this function. This cannot happen if contentWeight is set to 0.
Status: Implemented.

int GetFeatureVector (int instanceId, int[] relationIds, int typeOfStructFeatures, int m, int beta, double[] weights, int structJoinMethod) The same as GetStructureFeatureVectors(...) but it only returns the feature vector that corresponds to the instance that is specified with instanceId.
Status: Implemented.

A.5 Input and output

int SaveBowFile (string fileName) Saves the bag-of-words representations into a binary file suitable for OntoGen. The file is created in a remote directory. This directory can be accessed either via FTP or via HTTP.
Parameters: fileName: The name of the bag-of-words binary file.
Return value: non-negative integer: The file was created successfully. -1: Bag-of-words representations do not exist. CreateBows(...) needs to be invoked prior to this function.
Status: Implemented.

int SaveBowWgtFile (string fileName, int[] relationIds, int typeOfStructFeatures, int m, int beta, double[] weights, int structJoinMethod, double contentWeight) Saves the bag-of-words weights into a binary file suitable for OntoGen. The file is created in a remote directory. This directory can be accessed either via FTP or via HTTP. See GetFeatureVectors for the description of the parameters.
Parameters: fileName: The name of the bag-of-words weights binary file.
Return value: non-negative integer: The file was created successfully. -1: Bag-of-words representations do not exist. CreateBows(...) needs to be invoked prior to this function. This cannot happen if contentWeight is set to 0.
Status: Implemented.

int SavePajekNetFile (string fileName, int[] relationIds) Saves the graphs that correspond to the specified relations into a .net[work] file suitable for Pajek. Pajek is a software for [social] network analyses and is freely available at http://vlado.fmf.uni-lj.si/pub/networks/pajek/ .
Parameters: fileName: The name of the net[work] file.
Return value: non-negative integer: The file was created successfully.
Status: Implemented.

Appendix B: Installing and running OntoGen

OntoGen is installed and launched as follows:

- Download *Microsoft .NET Framework 2.0* from <http://www.microsoft.com/downloads/details.aspx?familyid=0856eacb-4362-4b0d-8edd-aab15c5e04f5&displaylang=en> by clicking on the *Download* button. Install the framework: execute the downloaded file (*dotnetfx.exe*) and follow the instructions provided by the setup application.
- Create a folder for OntoGen, e.g. *C:\OntoGen*.
- Download OntoGen from <http://ontogen.ijs.si/> by selecting *OntoGen 2.0 (new)* from the left-hand side menu. Unzip the downloaded package into the previously created OntoGen folder.
- Launch OntoGen by executing *OntoGen.exe* located in the OntoGen folder. For an easier access to OntoGen we recommend you to create a shortcut and place it onto the desktop.

Appendix C: State transition probability computation

In this appendix we formally derive and further explain the equations presented in Section 4.2. Suppose we have a finite state machine with states $1, 2, \dots, n$. In this light we can define a random variable S_t which tells us the state of the machine at time t .

Let us now create the adjacency matrix \mathbf{A} . The matrix element a_{ij} denotes the weight of the arc (or the edge) from vertex i to vertex j , if such arc (or edge) exists, or is set to zero if such arc (or edge) does not exist. We then normalize the rows of matrix \mathbf{A} so that they sum to 1 (i.e. we transform the outgoing-arc weights of every vertex into a probability distribution). To put this more formally, a_{ij} now becomes:

$$a_{ij} = \mathbb{P}(S_{t+1} = j \mid S_t = i) .$$

Let $\mathbf{b}^{(t)}$ be a vector in which the i -th element tells us the probability that the machine is in state i at time t :

$$b_i^{(t)} = \mathbb{P}(S_t = i) .$$

We set $\mathbf{b}^{(0)}$ according to the chosen source vertex. We set all its elements to zero except for the element that corresponds to the source vertex which we set to 1. Let us take a look at how to compute $\mathbf{b}^{(t)}$ for $t > 0$:

$$\begin{aligned} b_j^{(t)} &= \mathbb{P}(S_t = j) \\ &= \sum_{i=1}^n \mathbb{P}(S_t = j \wedge S_{t-1} = i) \\ &= \sum_{i=1}^n \mathbb{P}(S_t = j \mid S_{t-1} = i) \mathbb{P}(S_{t-1} = i) \\ &= \sum_{i=1}^n a_{ij} b_i^{(t-1)} = (\mathbf{A}\mathbf{b}^{(t-1)})_j , \end{aligned}$$

and therefore:

$$\mathbf{b}^{(t)} = \mathbf{A}\mathbf{b}^{(t-1)} = \mathbf{A}^2\mathbf{b}^{(t-2)} = \dots = \mathbf{A}^t\mathbf{b}^{(0)} .$$

Let us denote the probability that the target state (denoted by c) is reached at least once in at most m steps with $q(c, m)$. For $m = 0$, clearly $q(c, 0) = \mathbb{P}(S_0 = c) = b_c^{(0)}$. For $m > 0$ we can compute $q(c, m)$ as follows:

$$\begin{aligned} q(c, m) &= \mathbb{P}(S_0 = c \vee S_1 = c \vee \dots \vee S_{m-1} = c \vee S_m = c \vee S_m = c) \\ &= \mathbb{P}(U \vee V)^{13} \\ &= \mathbb{P}(U \wedge V) + \mathbb{P}(U \wedge \neg V) + \mathbb{P}(V \wedge \neg U) \\ &= \mathbb{P}(U) + \mathbb{P}(V \wedge \neg U) \\ &= \mathbb{P}(S_0 = c \vee \dots \vee S_{m-1} = c) + \mathbb{P}(S_0 \neq c \wedge S_1 \neq c \wedge \dots \wedge S_{m-1} \neq c \wedge S_m = c \wedge S_m = j) \\ &= q(c, j, m-1) + \sum_{j=1}^n \mathbb{P}(S_0 \neq c \wedge S_1 \neq c \wedge \dots \wedge S_{m-1} \neq c \wedge S_{m-1} = j \wedge S_m = c) \end{aligned}$$

¹³ $U = (S_0 = c \vee S_1 = c \vee \dots \vee S_{m-1} = c)$, $V = (S_m = c)$

$$= q(c, j, m-1) + \sum_{j=1}^n P(S_0 \neq c \wedge S_1 \neq c \wedge \dots \wedge S_{m-1} \neq c \wedge S_{m-1} = j) P(S_m = c | S_{m-1} = j) ,$$

and therefore:

$$q(c, m) = q(c, m-1) + \sum_{j=1}^n r(c, j, m-1) a_{jc} \quad (3)$$

$r(c, j, m)$ denotes the probability that the target state c was not once reached within m steps and that the machine ended up in state j after the last step. If $c = j$, then $r(c, j, m)$ is clearly 0. If $c \neq j$, on the other hand, we have $r(c, j, 0) = b_j^{(0)}$ for $m = 0$, and for $m > 0$ we have:

$$\begin{aligned} r(c, j, m) &= P(S_0 \neq c \wedge S_1 \neq c \wedge \dots \wedge S_{m-1} \neq c \wedge S_m \neq c \wedge S_m = j) \\ &= \sum_{i=1}^n P(S_0 \neq c \wedge S_1 \neq c \wedge \dots \wedge S_{m-1} \neq c \wedge S_{m-1} = i \wedge S_m \neq c \wedge S_m = j) \\ &= \sum_{i=1}^n P(S_0 \neq c \wedge S_1 \neq c \wedge \dots \wedge S_{m-1} \neq c \wedge S_{m-1} = i \wedge S_m = j) \\ &= \sum_{i=1}^n P(S_0 \neq c \wedge S_1 \neq c \wedge \dots \wedge S_{m-1} \neq c \wedge S_{m-1} = i) P(S_m = j | S_{m-1} = i) , \end{aligned}$$

and therefore:

$$r(c, j, m) = \sum_{i=1}^n r(c, i, m-1) a_{ij} .$$

To sum up:

$$r(c, j, m) = \begin{cases} 0 & c = j \\ b_j^{(0)} & c \neq j, m = 0 \\ \sum_{i=1}^n r(c, i, m-1) a_{ij} & c \neq j, m > 0 \end{cases} \quad (4)$$

If we now arrange values $r(\dots)$ into a matrix \mathbf{R} so that $R_{cj}^{(m)} = r(c, j, m)$, Equation (4) corresponds to the term $\mathbf{R}^{(m)} = \text{zdiag}(\mathbf{R}^{(m-1)})\mathbf{A}$ ($\text{zdiag}(\dots)$ sets the diagonal entries of the matrix to zero). Similarly, we can arrange values $q(\dots)$ into a vector \mathbf{q} so that $q_c^{(m)} = q(c, m)$. Then, with respect to Equation (3), we get:

$$\begin{aligned} q_c^{(m)} &= q_c^{(m-1)} + \sum_{j=1}^n R_{cj}^{(m-1)} a_{jc} \\ &= q_c^{(m-1)} + (\mathbf{R}^{(m-1)} \mathbf{A})_{cc} . \end{aligned}$$

Note that the second term represents the diagonal of $\mathbf{R}^{(m-1)}\mathbf{A}$. Therefore the procedure of computing $\mathbf{q}^{(t)} = (q(1, t), q(2, t), \dots, q(n, t))$ is as follows:

1. $\mathbf{q}^{(0)} \leftarrow \mathbf{b}^{(0)}$;
2. $q_c^{(0)} \leftarrow 0$;

¹⁴ $P(S_0 \neq c \wedge S_1 \neq c \wedge \dots \wedge S_{m-1} \neq c \wedge S_{m-1} = j) P(S_m = c | S_{m-1} = j) = r(c, j, m-1) a_{jc}$.

3. $\mathbf{R}^{(0)} \leftarrow$ a matrix with n columns; all the columns are initially set to $\mathbf{b}^{(0)}$;
4. set the diagonal elements of $\mathbf{R}^{(0)}$ to zeros (i.e. $\mathbf{R}^{(0)} \leftarrow \text{zdiag}(\mathbf{R}^{(0)})$);
5. for $t \leftarrow 1$ to m do:
 - a. $\mathbf{R}^{(t)} \leftarrow \mathbf{R}^{(t-1)}\mathbf{A}$;
 - b. $\mathbf{d} \leftarrow$ vector of diagonal elements of matrix $\mathbf{R}^{(t)}$;
 - c. $\mathbf{q}^{(t)} \leftarrow \mathbf{q}^{(t-1)} + \mathbf{d}$;
 - d. set the diagonal elements of $\mathbf{R}^{(t)}$ to zeros (i.e. $\mathbf{R}^{(t)} \leftarrow \text{zdiag}(\mathbf{R}^{(t)})$);
6. $q_c^{(m)}$ now represents the probability that the machine reaches state c in at most m state transitions.

It is also possible to incorporate a “decay” parameter β into the equations by extending the state machine with one additional state $n + 1$. This new state serves as an additional choice in each of the original states. In any of the original states this new choice is selected with probability β . The new state prevents the machine from going back to one of the original states thus allowing the machine to “stop”. To incorporate β , we need to extend the state-transition probability matrix \mathbf{A} with one additional row and one additional column. The additional row consists of zeros except for the last element which is 1. The additional column contains values β except for the last element which is 1. Furthermore, we need to multiply the original values in matrix \mathbf{A} with $(1 - \beta)$ to normalize the rows of the extended matrix. Figure 23 illustrates how matrix \mathbf{A} is extended into matrix \mathbf{A}' which incorporates the decay parameter.

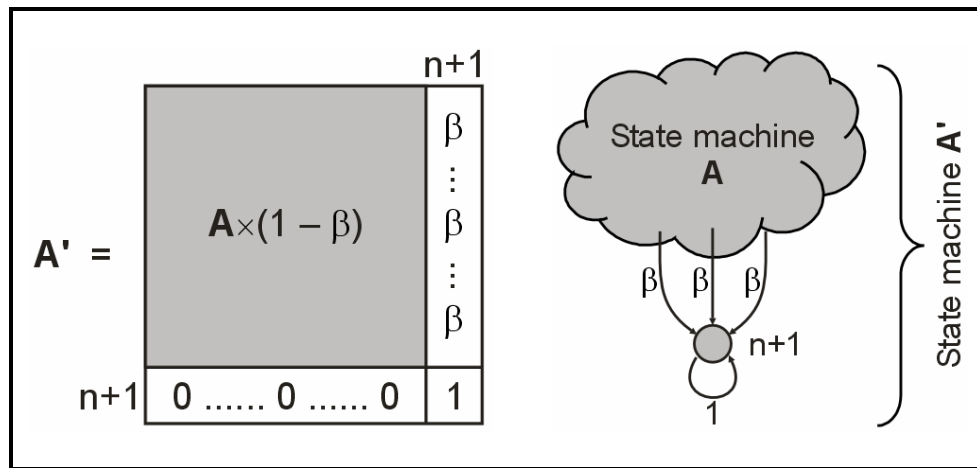


Figure 23: Matrix \mathbf{A} is extended into matrix \mathbf{A}' which incorporates the decay parameter β .