Effective Development with GATE and Reusable Code for Semantically Analysing Heterogeneous Documents

Adam Funk, Kalina Bontcheva

Department of Computer Science
University of Sheffield
Regent Court, Sheffield, S1 4DP, UK
a.funk@dcs.shef.ac.uk, k.bontcheva@dcs.shef.ac.uk

Abstract

We present a practical problem that involves the analysis of a large dataset of heterogeneous documents obtained by crawling the web for information related to web services. This analysis includes information extraction from natural-language (HTML and PDF) and machine-readable (WSDL) documents using NLP and other techniques, classifying documents as well as services (defined by sets of documents), and exporting the results as RDF for use in the back-end of a portal that uses Web 2.0 and Semantic Web technology. Triples representing manual annotations made on the portal are also exported back to our application to evaluate parts of our analysis and for use as training data for machine learning (ML). This application was implemented in the GATE framework and successfully incorporated into an integrated project, and included a number of components shared with our group's other projects.

1. Introduction

The Service-Finder project addresses the problem of web service discovery for a wide audience through a portal[1] which presents automatic semantic descriptions of a wide range of publicly available web services and also enables service consumers (not just providers) to enrich them according to Web 2.0 principles (manual annotation according to the project ontology, tagging, and wiki-like editing of free text fields). In this project, the Service Crawler (SC) carries out focused crawling for web services and archives WSDL files and related HTML files, then passes monthly batches of these data to the Automatic Annotator (AA), which analyses them to produce semantic annotations to the Conceptual Indexer and Matchmaker (CIM), the semantic repository and back end for the web portal. Additional components include the portal itself and the clustering engine that provides recommendations. (**?**)
Here we present the implementation of the Automatic Annotator using the versatile GATE[2] (**?**) framework for NLP and related applications.

1.1. Input and output

The SC (Service Crawler) component delivers to the AA a monthly batch of data, consisting of a number of compressed Heritrix (**?**) Internet Archive files (up to 100 MB each), along with an index of all the documents in the batch. The documents for each service include one or more WSDL files, an abstract[3], and zero or more HTML and PDF files (especially those with contact details, links to WSDL files, pricing information, terms and conditions, and other useful information).
Figure 1 shows a sample extract from the index, which lists every document in the crawler output, along with the archive file number and offset where it can be found and type codes (e.g., `w` for WSDL or `a` for abstract). Each stanza is headed by a service URI[4], which is also used in the Heritrix archives as the URL of the abstract. The same service URI may occur several times in this index with different documents listed below it.
This collection of files is downloaded onto one of our servers for processing, as described in the rest of the paper. The results are exported as RDF-XML to the CIM. Table 1 summarizes the input and output of a typical monthly batch of data.

Input from the SC	
Number of `.arc.gz` files	5
Total size of compressed files	441 MB
Number of documents	∼ 250 000
Output to the CIM	
Number of RDF-XML files	30
Total size of compressed files	40 MB
Number of RDF triples	∼ 4 500 000
Number of Providers	∼ 8 700
Number of Services	∼ 25 000

Table 1: Typical AA input and output

1.2. Annotation tasks

The Automatic Annotator's principal tasks are as follows:

- analyse WSDL files to produce *Endpoint*, *Interface*, and *Operation* instances as well as properties associating them with each other and with the relevant *Service* instances;

[1] `http://demo.service-finder.eu/`

[2] `http://gate.ac.uk/`

[3] The *abstract* is an HTML file that the SC compiles from various elements' and attributes' strings (the service name, documentation, operation names, input and output parameters, etc.) in the best WSDL file for the service.

[4] The crawler generates URIs for instances of the *Service* and *Provider* classes. The service URI always consists of the provider URI followed by one path-element, so the provider URI can be easily obtained from the service URI.

```
http://seekda.com/providers/dp2003.com/FileService  1   100684561  a
 http://dp2003.com/filews/filews.asmx?WSDL          1   19796469   w
 http://dp2003.com/filews/ListUser                  3   29130320   f o
 http://dp2003.com/filews/Logout                    3   29131388   f o
 http://microsoft.com/wsdl/mime/textMatching/       4   104841754  f o
 http://dp2003.com/filews/UserInfo                  3   29132932   f o
```

Figure 1: Excerpt from the input index

Stage		Approx. number of documents
Input	total documents	250 000
Preprocessing reductions		
	HTTP error codes	49 000
	unwanted provider IDs	5 000
	empty documents	3 000
	reduced duplicates	23 000
	faulty XML	< 30
Output	HTML	37 000
	WSDL	110 000
	abstract	25 000
	total (31% reduction)	173 000

Table 2: Typical results of preprocessing

	Time in hours	
Tool	Before	After
Preprocessor	3.0	18.3
Analysis	72.9	18.1
Archiving	6.5	2.5
Total	82.4	38.9

Table 3: Examples of AA performance times

- classify documents by type (e.g., documentation, pricing, contact details) and rate them as low-, medium-, or high-interest;
- carry out information extraction to identify providers' addresses, phone numbers, e-mail addresses, etc.;
- carry out information extraction over services to identify service level agreements, free trials, etc.;
- categorize each service in one or more of the 59 subclasses of *ServiceCategory*.

Some tasks require information to be amalgamated across various sets of documents, and all the output is expressed as RDF-XML according to the project ontology. Figure 2 highlights several types of keywords and annotated pieces of information in GATE's GUI.

2. Implementation

We implemented the Automatic Annotator tools using GATE, a versatile, extensible library and framework for NLP and text processing. We used the *GATE Developer* GUI environment to develop some of the pipelines (using the ANNIE information extraction pipeline as a starting point), GATE's gazetteer and JAPE tools for rapid development of processing resources (PRs) to mark keywords and phrases with weights according to context, and PRs for more complicated functions functions that could not be easily coded in JAPE, along with control programs, both written in Java using the *GATE Embedded* library. (**?**)

2.1. Preprocessing

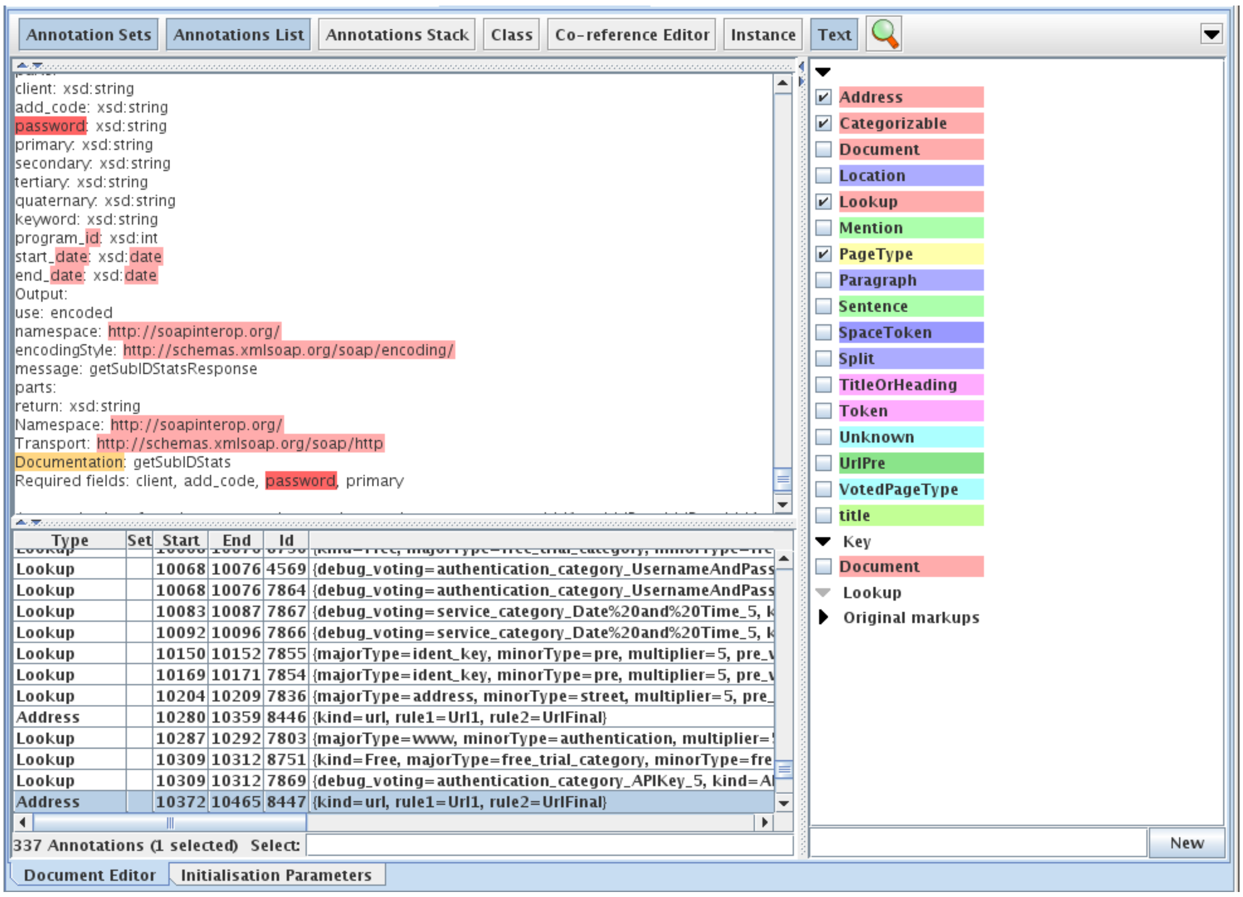
First of all, we recognized the need to split the large input dataset into manageable chunks and took advantage of the independent nature of the data about each provider. We developed a preprocessor in Java that can be run from the command-line or a shell script in GNU screen[5] on a server. This preprocessor reads the index file into memory, creates several (typically 30) serially numbered GATE Serial DataStores[6], then iterates through all the documents in the input archive files. For each document, it checks the HTTP status code and discards the document if the code is 3xx, 4xx, or 5xx; otherwise, it uses the index to identify the document type (WSDL, abstract, HTML, PDF) and the service and provider URIs. It then calculates the MD5 hash (**?**) of the document and checks the list of already hashed documents; in case of a match, it reloads the existing serialized GATE document, merges the additional document URL and service URI into it, re-saves it, and goes on to the next document. Table 2 shows the effects of these suppression and de-duplication steps, and Table 3 shows the striking "before and after" effects on performance of making more effort in the preprocessor to eliminate unnecessary documents from analysis (on a Xeon X3220 server with Java's `-Xmx 4000m` setting). The preprocessing time increased sixfold but the total time decreased by 53%.

For each valid, non-duplicate document, the preprocessor instantiates a GATE document in a mark-up-aware manner, with the plain-text content, the HTML mark-up or XML tags, and the metadata all stored in the appropriate parts of GATE's document model (the document content, *Original markups* annotation set, and document feature map, respectively). (GATE uses a modified form of the TIPSTER and Atlas formats (**?**; **?**) as stand-off mark-up, as shown in Figure 2.) (WSDL documents are also analysed by software developed mainly by another project partner, seekda[7], integrated so that it stores its results as an RDF-XML document feature.) It adds this document to the provider's corpus (which it creates when it first encounters that provider). The

[5] `http://www.gnu.org/software/screen/`

[6] A Serial DataStore provides disk-based persistence for documents and corpora using Java serialization.

[7] `http://seekda.com/`



```
client: xsd:string
add_code: xsd:string
password: xsd:string
primary: xsd:string
secondary: xsd:string
tertiary: xsd:string
quaternary: xsd:string
keyword: xsd:string
program_id: xsd:int
start_date: xsd:date
end_date: xsd:date
Output:
use: encoded
namespace: http://soapinterop.org/
encodingStyle: http://schemas.xmlsoap.org/soap/encoding/
message: getSubIDStatsResponse
parts:
return: xsd:string
Namespace: http://soapinterop.org/
Transport: http://schemas.xmlsoap.org/soap/http
Documentation: getSubIDStats
Required fields: client, add_code, password, primary
```

Type	Set	Start	End	Id	
Lookup		10068	10076	4569	{debug_voting=authentication_category_UsernameAndPass
Lookup		10068	10076	7864	{debug_voting=authentication_category_UsernameAndPass
Lookup		10083	10087	7867	{debug_voting=service_category_Date%20and%20Time_5, k
Lookup		10092	10096	7866	{debug_voting=service_category_Date%20and%20Time_5, k
Lookup		10150	10152	7855	{majorType=ident_key, minorType=pre, multiplier=5, pre_v
Lookup		10169	10171	7854	{majorType=ident_key, minorType=pre, multiplier=5, pre_v
Lookup		10204	10209	7836	{majorType=address, minorType=street, multiplier=5, pre_
Address		10280	10359	8446	{kind=url, rule1=Url1, rule2=UrlFinal}
Lookup		10287	10292	7803	{majorType=www, minorType=authentication, multiplier=
Lookup		10309	10312	8751	{kind=Free, majorType=free_trial_category, minorType=fre
Lookup		10309	10312	7869	{debug_voting=authentication_category_APIKey_5, kind=Al
Address		10372	10465	8447	{kind=url, rule1=Url1, rule2=UrlFinal}

337 Annotations (1 selected) Select:

- ✓ Address
- ✓ Categorizable
- ☐ Document
- ☐ Location
- ✓ Lookup
- ☐ Mention
- ✓ PageType
- ☐ Paragraph
- ☐ Sentence
- ☐ SpaceToken
- ☐ Split
- ☐ TitleOrHeading
- ☐ Token
- ☐ Unknown
- ☐ UrlPre
- ☐ VotedPageType
- ☐ title

Key
- ☐ Document

Lookup

Original markups

Figure 2: Annotated document in GATE

corpora are allocated in a loop over the datastores so that the latter end up with roughly the same sizes. The provider–corpus and document–MD5 mappings are kept in memory since they are used so frequently. Finally, the preprocessor closes all the corpora (synchronizing them on disk), closes the datastores, and writes several index files that indicate which providers and services are in which corpora and datastores.

2.2. Analysis

We then run a shell script that carries out the main analysis tasks and generates the consolidated RDF-XML file separately over each datastore.

As mentioned earlier, we took ANNIE as the starting point for the information extraction tasks, but modified it so that most of its PRs run only on HTML and PDF documents.

Each datastore produced by the preprocessor is processed through a Java tool that includes ANNIE[8] with additional gazetteers and rules developed within Service-Finder to identify relevant terms in the web service domain and annotate interesting sections and documents, consolidate the information from various documents for each provider, merge in the RDF-XML snippets generated by the preprocessor and attached to the corpora and documents, and produce one large RDF-XML file for each block (datastore).

2.2.1. Overview

The analysis pipeline consists of the following series of processing resources (PRs), as illustrated in Figure 3.

1. Standard ANNIE components tokenize and sentence-split the HTML and PDF documents (creating *Token* and *Sentence* annotations on the document). Abstracts and WSDLs are processed with a *source-code tokenizer* (developed in the TAO project[9]), a version of the ANNIE tokenizer with the rule files modified to split camel-cased strings (e.g., `getUnsplicedSequence` → `get Unspliced Sequence`) as well as tokens separated by whitespace.

2. The ANNIE gazetteers and NER (named-entity recognition) module (consisting of JAPE transducers) identify and annotate a range of entities such as *Date*, *Person*, *Organization*, and *Address*.

3. Gazetteers developed for this application mark keywords relating to web services, such as those used to indicate free trials, terms and conditions, pricing, and categories of services. Figure 4 shows keywords as-

[8]ANNIE is the information extraction system supplied with GATE; it includes standard NLP tools for English (such as a tokenizer and POS tagger) and gazetteers and rule-based tools for named entity recognition.

[9]`http://www.tao-project.eu/`

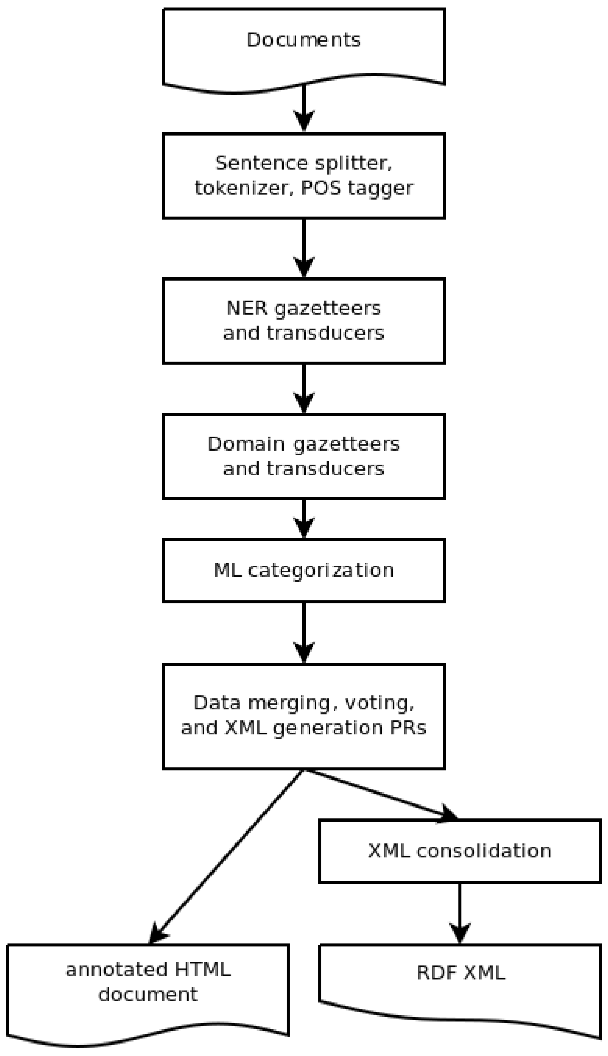

Documents
Sentence splitter, tokenizer, POS tagger
NER gazetteers and transducers
Domain gazetteers and transducers
ML categorization
Data merging, voting, and XML generation PRs
XML consolidation
annotated HTML document
RDF XML

Figure 3: Overview of the IE pipeline

terms and conditions	user agreement
terms & conditions	terms of use
licence agreement	TOU
license agreement	T&C
licencing agreement	AUP
licensing agreement	T&Cs
acceptable use policy	TOS
terms of service	

Figure 4: Examples of keywords that vote for *TermsAndConditionsPage*

sociated with the *TermsAndConditionsPage* document type.

4. A series of custom JAPE transducers compare the annotations produced by ANNIE and the custom gazetteers with the HTML mark-up in order to evaluate their role and set a *multiplier* feature accordingly, which is used by the voting processor in step 6. (For example, keywords and named-entity annotations in `title` or `h1` elements are more important than elsewhere in the document, and `p` elements that begin with the keyword *Description* or *Price* are more likely to relate to the description of the service or its pricing. Such annotations are assigned a *multiplier* feature $1 \leq m$, typically $m < 6$.) Other JAPE rules try to identify company details such as country of origin.

   These rules also have access to document features containing metadata such as the service, provider, and document URIs, which they can copy into features on the annotations they create. They also adjust an *interesting* feature[10].

5. An instance of the GATE machine learning PR (**?**) aims to label each document with a class from the service category ontology. Section 2.2.2 describes the integration of this component in more detail.

6. "Voting" PRs (extensions of *AbstractLanguageAnalyser* from the *GATE Embedded* library) compare the weighted frequency of significant keywords to assign certain ontology classes and properties as follows.

   - For each WSDL document and interesting (see step 4 above) textual (i.e., HTML or PDF) document, create a potential instance of the general class *Document* or one of its subclasses (*DocumentationPage*, *TermsAndConditionsPage*, etc.) and assert the properties *hasSize*, *hasTitle*, and *retrievedAt*; also create a *DocumentAnnotation* associating the document with a service or provider. (Some document types, such as *ContactDetailsPage*, are associated with providers; others are associated with services.)
   - For each service, assertions of the properties *supportsAuthenticationModel*, *hasServiceLevelAgreement*, *allowsFreeTrials*, etc.; these properties are not asserted if no votes (keywords) are found.
   - For each service, the two best categories for *CategoryAnnotation* (or just one category if all the votes were for the same one); the categories assigned by machine-learning outweigh those generated by keywords and rules (as §2.2.2 explains in detail).
   - For each provider, one or two best values for *hasHomepage*, *fromCountry*, *hasEmail*, *hasAddress*, and *hasTelephone* (these properties are not asserted when no values are found in the documents).

All GATE PRs override the `execute` method, which is called on each document in the corpus. Each voting PR uses additional hooks, as shown in Figure 5, to initialize the set of "ballots" at the beginning of the `execute` on the first document in the corpus; and to compute the results, generate the corresponding RDF-XML, and store it as a corpus feature at the

[10] $0.0 \leq i \leq 3.0$, interpreted as low ($0.0 \leq i < 1.0$), medium ($1.0 \leq i < 2.0$), or high ($2.0 \leq i \leq 3.0$). All documents start with 0.0.

```
public void execute() throws ExecutionException {
   if (corpus.indexOf(document) == 0) {
      // Before analysing the first document: initialize the data for the corpus
   }

   // Analyse this document and record the votes

   if (corpus.indexOf(document) == (corpus.size() - 1)) {
      // After analysing the last document: compute the results,
      // generate RDF-XML, and store it as a corpus feature
   }
}
```

Figure 5: Hooks in a voting PR's *execute()* method. Note that *CorpusController.execute()* iterates through the corpus's documents in list order: opening each document, calling each PR's *execute()* method, and closing the document.

end of the `execute` on the last document in the corpus. (The RDF-XML is generated by matching and filling in templates, as described in §2.2.3.)

7. Some important annotations on the document are translated into RDF-XML according to a set of templates based on the features of each annotations. (This RDF-XML is also generated from templates.)

8. All the RDF-XML snippets are collected from the document and corpus features and consolidated into one output file for the datastore.

The "pipeline" outlined above actually consists of two GATE *SerialAnalyserController* instances (conditional corpus pipelines) serialized as `gapp` files and an XML configuration file for GATE's *Batch Learning PR* (**?**). Like the preprocessor, the main analysis tool is a command-line Java program that can be run in GNU screen on a server; it instantiates the two pipelines from the `gapp` files and creates a separate pipeline for the learning PR (which it instantiates from the configuration file); it then iterates through the corpora in a datastore, runs the pipelines in the correct sequence (taking advantage of GATE's serialization to save memory: only one document is loaded at a time), and finally collects together all the RDF-XML snippets stored as document and corpus features in the datastore and consolidates them into one RDF-XML file, which constitutes this tool's output to the CIM for that block (datastore).
A modified version is also used in the NeOn project[11] for batch processing large datastores through pipelines (both specified by command-line arguments) on a server.

2.2.2. Service categorization

Although the portal's Web 2.0 features encourage users to add, correct, and otherwise improve the category annotations of services, it is important to provide a number of reasonably good ones to start with so users will find the portal useful and interesting and then contribute to it—otherwise they would have to face 23 000 uncategorized services with only keyword searching. We therefore placed strong emphasis in the AA on rapid development and then refinement of service categorization. In this paper, we will summarize this task and describe the integration of the components into the pipeline above. Our scientific results here are interesting in their own right and are published in detail elsewhere (**?**).
Briefly, the task is to annotate each service with one or more of the 59 subclasses of *Category* for web services, such as *Genetics*, *Address Information*, and *Media Management*, arranged in a shallow tree (down to three levels below the top class) in seven main branches, such as *Business*, *Consumer*, and *Science* (**?**; **?**).
In the early stages, we had no training data but needed to generate some category annotations quickly for the portal, so we created *ad hoc* gazetteers of keywords and phrases based on the category names, synonyms, and related words. These were affected by the multipliers described in step 4 in §2.2.1. The IE pipeline at this stage was as shown in Figure 3 but without the machine learning (ML) categorization PR (step 5 in §2.2.1); the voting process treated keyword or phrase match as m (the multiplier) votes for the relevant category for each service associated with the document, and annotated each service with the two highest-scoring categories (an arbitrary limit agreed within the project).
After the first release of the portal, we manually annotated a few hundred services with the same portal features that users have for manually adding or correcting categories, and used these annotations to evaluate the AA's categorization and then as training data for machine-learning. We trained the ML PR to classify documents, carried out evaluations with various ML parameters, and integrated the ML PR into the pipeline by assigning a very high multiplier ($m = 100$) to the ML annotations so that in the subsequent voting PR, they will outweigh any gazetteer-based categorizations, although the latter can still be used to make the total number of categories per service up to two if the ML PR fails to classify any of a service's documents or provides only one category, since it is more user-friendly for us to provide approximate categorizations than none at all. We have published elsewhere (**?**) the full technical details and results of our ML experiments.

2.2.3. Approaches to ontology development and population

One can develop and populate ontologies from GATE applications using the GATE Ontology API (**?**), and we have

[11] `http://www.neon-project.org/`

used this technique in our SPRAT and SARDINE[12] applications (**?**; **?**), which use ontology design patterns to recognize new concepts, instances, and properties, and add them to the seed ontology (which can be empty) used to initialize the application. The principal output of both applications is the extended ontology produced from a corpus of documents. As an evaluated example, SPRAT processed 25 Wikipedia articles and produced 1058 classes, 659 subclasses, 23 instances, and 55 properties as output.

The CLOnE[13] and RoundTrip Oontology Authoring software (**?**; **?**) developed and used in SEKT[14] and NEPOMUK[15], also used the GATE Ontology API.

In the Service-Finder and MUSING tasks, however, the ontologies' class and property structures are fixed and our applications only need to create instances and property assertions, specifically in RDF-XML (as requested by other developers in the projects). The volume of data generated is also much larger and more time-consuming (as shown in Tables 1 and 3), so we use a template-filling technique which requires relatively little memory.

We originally developed this GATE PR for generating XML (principally RDF-XML) in the MUSING[16] business intelligence research project, and have used modified and improved versions of it in Service-Finder and CLARIN[17]. (A version of it will probably be integrated into GATE in the future, once we have settled the list of configurable features.) The PR's configuration file consists mainly of a series of template specifications, as Figure 6 shows. Each entry lists the annotation features that are required for the template to match, and the values of those features are substituted for the variables in the template. (The annotation can have other features, which are ignored.) The PR's output for a matching annotation consists of the "filled-in" copy of the element(s) in the `template` element. Figures 7 and 8 show the characteristics of a matching annotation and the resulting XML snippet, respectively. (In addition to the `feature` element, the entry can have a `generated` specification, which names a variable for which a UUID string is substituted. This is useful for generating unique `rdf:id` values.)

When the configuration file is loaded (during initialization of the PR), the order of the entries is preserved, so that for each annotation, the first match is used. It is therefore possible to use a template that requires an annotation with three features, followed by a simpler template which requires only two of the three (and uses a default value for the third, for example); the first template will "fire" whenever all three features appear, but the PR will drop back to the second one if the first does not match, and so on. (Another version of this component takes a *Map<String, String>* rather than a GATE Annotation; the voting PR generates the maps from the winning ballots and obtains the corresponding RDF-XML from this generator.)

2.3. Miscellaneous tools

We developed two other separate tools for the Service-Finder AA. The archiving tool iterates through the documents in the GATE datastores after the analysis tool has been run (leaving its annotations on the serialized documents) and produces Heritrix archives of the plain-text content of "interesting" (see step 4 in §2.2) HTML and PDF documents and selected strings from WSDLs and abstracts; the CIM produces Lucene indexes from these files to support keyword searches for services on the portal. The quantitative evaluation tool loads the RDF-XML files into a Sesame repository, executes a series of SERQL or SPARQL queries specified in a control file, and produces an output file containing the number of results for each query or a list of those results (according to the specifications in the control file).

To provide a datastore suitable for training the ML classifier, we added extra features to the preprocessor (§2.1), activated by additional command-line options, so that it reads a file of manual category annotations (exported from the CIM) as well as a set of archives from the SC and produces a datastore containing only the documents related to the manually annotated services, with document features representing the categories. We also developed a pipeline for training the classifier, which carries out steps 1 through 4 in §2.2 and treats the annotated documents as instances, and then saves the learned model for use in the complete integrated analysis tool.

3. Conclusion

The immediate result of the development presented here was its contribution to the successful completion of the Automatic Annotator tasks and their integration in the Service-Finder project, which received good intermediate and final project reviews. The relevant public deliverables (**?**; **?**) describe the AA software in much greater detail. We evaluated the AA software itself in two ways: IE measures (precision, recall, and F_1) for the especially important and scientifically interesting service categorization task, which we present in detail elsewhere (**?**); and quantitative measures of the instances and property assertions created at various stages of development (**?**).

The broader results included the dissemination of GATE as a tool for semantically annotating the results of focused web crawling—in particular at a *Future Internet Symposium* tutorial on web service crawling and annotation (**?**), where we demonstrated the suitability of the *GATE Developer* IDE and *GATE Embedded* library for rapid application development and effective code re-use—and the development of useful, reusable code shared with other projects (NeOn, MUSING, and CLARIN).

4. Acknowledgements

This research is partially supported by the EU Sixth Framework Program projects NeOn (IST-2005-027595) and MUSING (FP6-027097) and the Seventh Framework Program project Service-Finder (FP7-215876).

[12]Semantic Pattern Recognition and Annotation Tool; Species Annotation and Recognition and Indexing of Named Entities; both developed in NeOn.

[13]Controlled Language for Ontology Editing.

[14]`http://www.sekt-project.com/`

[15]`http://nepomuk.semanticdesktop.org/`

[16]`http://www.musing.eu/`

[17]`http://www.clarin.eu/`

```
<?xml version="1.0" encoding="UTF-8"?>
<root xmlns:sfso="http://www.service-finder.eu/ontologies/ServiceOntology#"
      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema#" >
...
<entry id="provider_homepage_link">
<feature name="provider_URL"  />
<feature name="homepage_link" />
<template>
  <sfso:Provider  rdf:about="${provider_URL}">
    <sfso:hasHomepage  rdf:datatype="xsd:string">${homepage_link}</sfso:hasHomepage>
  </sfso:Provider>
</template>
</entry>
...
</root>
```

Figure 6: Excerpt from the RDF-XML template file. To match this template, at least the `provider_URL` and `homepage_link` features must be provided.

Type	`Mention`
Start and end offsets	117–121
Underlying text	`Home`
Features	`doc_URL="http://www.serviceobjects.net/products/dots_phone_exchange_details.asp"`
	`homepage_link="http://www.serviceobjects.net/default.asp"`
	`provider_URL="http://seekda.com/providers/serviceobjects.com"`
	`service_URL="http://seekda.com/providers/serviceobjects.com/DOTSPhoneExchange"`

Figure 7: Annotation matching the template in Figure 6

5. References

S. Bird and M. Liberman. 1999. A Formal Framework for Linguistic Annotation. Technical Report MS-CIS-99-01, Department of Computer and Information Science, University of Pennsylvania. `http://xxx.lanl.gov/abs/cs.CL/9903003`.

K. Bontcheva, V. Tablan, D. Maynard, and H. Cunningham. 2004. Evolving GATE to Meet New Challenges in Language Engineering. *Natural Language Engineering*, 10(3/4):349—373.

S. Brockmans, M. Erdmann, and W. Schoch. 2008. Hybrid matchmaker and Service-Finder ontologies (alpha release). Deliverable D4.2, Service-Finder Consortium.

Saartje Brockmans, Irene Celino, Dario Cerizza, Daniele Dell'Aglio, Emanuele Della Valle, Michael Erdmann, Adam Funk, Holger Lausen, and Nathalie Steinmetz. 2010. Final report on assessment of tests for beta release. Deliverable D7.5, Service-Finder Consortium, January.

H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. 2002. GATE: A Framework and Graphical Development Environment for Robust NLP Tools and Applications. In *Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics (ACL'02)*.

Brian Davis, Ahmad Ali Iqbal, Adam Funk, Valentin Tablan, Kalina Bontcheva, Hamish Cunningham, and Siegfried Handschuh. 2008. Roundtrip ontology authoring. In *Proceedings of the 7th International Semantic Web Conference (ISWC)*, Karlsruhe, Germany, October.

Emanuele Della Valle, Dario Cerizza, Irene Celino, Andrea Turati, Holger Lausen, Nathalie Steinmetz, Michael Erdmann, and Adam Funk. 2008. Realizing Service-Finder: Web service discovery at web scale. In *European Semantic Technology Conference (ESTC)*, Vienna, September.

A. Funk and K. Bontcheva. 2010. Ontology-based categorization of web services with machine learning. In *Proceedings of the seventh international conference on Language Resources and Evaluation (LREC)*, Valetta, Malta, May.

A. Funk, V. Tablan, K. Bontcheva, H. Cunningham, B. Davis, and S. Handschuh. 2007. CLOnE: Controlled Language for Ontology Editing. In *Proceedings of the 6th International Semantic Web Conference (ISWC 2007)*, Busan, Korea, November.

Adam Funk, Holger Lausen, and Nathalie Steinmetz. 2009a. Automatic semantic annotation component—beta release. Deliverable D3.4, Service-Finder Consortium, November.

Adam Funk, Holger Lausen, Nathalie Steinmetz, and Kalina Bontcheva. 2009b. Automatic semantic annotation research report—version 2. Deliverable D3.3, Service-Finder Consortium, October.

R. Grishman. 1997. TIPSTER Architecture Design Document Version 2.3. Technical report, DARPA. `http://www.itl.nist.gov/div894/894.02/-related_projects/tipster/`.

Y. Li, K. Bontcheva, and H. Cunningham. 2009. Adapting SVM for Data Sparseness and Imbalance: A Case Study on Information Extraction. *Natural Language Engineering*, 15(2):241–271.

```
<sfso:Provider rdf:about="http://seekda.com/providers/serviceobjects.com">
  <sfso:hasHomepage
    rdf:datatype="xsd:string">http://www.serviceobjects.com/default.asp
      </sfso:hasHomepage>
</sfso:Provider>
```

Figure 8: RDF-XML snippet output produced from the template in Figure 6 and the annotation in Figure 7

D. Maynard, A. Funk, and W. Peters. 2009a. Nlp-based support for ontology lifecycle development. In *Workshop on Collaborative Construction, Management and Linking of Structured Knowledge (CK 2009) at ISWC 2009*, October.

D. Maynard, A. Funk, and W. Peters. 2009b. Using lexico-syntactic ontology design patterns for ontology creation and population. In *Workshop on Ontology Patterns (WOP 2009) at ISWC 2009*, October.

R. Rivest. 1992. The MD5 message-digest algorithm. RFC 1321, Internet Engineering Task Force, April.

Kristinn Sigurðsson, Michael Stack, and Igor Ranitovic. 2008. Heritrix user manual. Software documentation, Internet Archive.

N. Steinmetz, A. Funk, and M. Maleshkova. 2009. Web service crawling and annotation (tutorial). In *Future Internet Symposium (FIS 2009)*, September.