Ontology Generation and Document Classification

Technical Documentation (KNOWMAK WP2)

Adam Funk

11 October 2019

Contents

1. Overview

This document describes the technical aspects of the KNOWMAK classification tool in WP2. The classifier takes as input documents such as projects, patents and publications, and classifies them with topics from the ontology. Each topic returned also has a score indicating how "good" that topic is. The topics correspond to ontology classes. The topic association is made based on matching keywords against the text. These keywords are associated with ontology classes, and a scoring system decides which are the best match. In other words, documents are classified based on keyword matches, which are related to topics, so that topics can be assigned and scored. Various mechanisms are used for the scoring algorithms.

The stages of the work thus consist of:

1. building a core ontology structure (classes)
2. populating the ontology
    a) adding an initial set of keywords to classes, based on information in the ontology
    b) refining the ontology population (extending the initial set of keywords using other sources of information)
    c) adding scores to the keywords
3. creating the classifier (GATE application)
4. running the classifier on the documents (GATE web service)

In addition to this, two search demos are created based on the ontology - one for faceted search and one for filtered search. A number of other tools are created additionally which assist with evaluation.

The software described in §2 builds the KNOWMAK ontology, classifier gazetteer, and the JSON data file for the faceted and filtered search demos. Building the classifier gazetteer is *ontology population* (i.e., populating with keyphrases) in the project.

The software described in §3 (as well as in the separate user guide) provides a REST service with a POST endpoint which accepts documents, classifies them according to the KNOWMAK WP2 ontology, and returns classification and keyword data in JSON. (This service includes the enriched gazetteer described in §2.)

For each POST, the request body is turned into a GATE document and run through a GATE application (which uses the classification gazetteer mentioned above) drawn from a multithreaded pool of configurable size. When the application finishes, the service collects the classification and keyword data from the document and returns them in the response body, using the JSON format described in §3.3.3.

2. Generating the ontology and gazetteers

This section explains how to generate the RDF ontology, the gazetteer used by the classifier application, and the JSON data used in the search demonstrations. The ontology is built by reusing relevant parts of the nature.com ontology, which are combined with some manually generated sections of the ontology. Further information is added such as provenance (where each class comes from) and keywords from IPC descriptions (for patents) and H2020 objectives (for projects).

2.1. Requirements

- a current version of GATE
- Java 8 (tested with Oracle 1.8.0)
- Scala (tested with 2.11.8)
- subversion check-out of `gate-extras/knowmak`
- a git check-out[1] of `gatetool-runpipeline` at the same directory level as `gate-extras`
- Bash shell with the `SCALA_HOME` and `JAVA_HOME` environment variables set correctly.
- Python 3 (tested with 3.5.3)
- additional Python libraries (installable with pip3)
  - openpyxl (2.4.8)
  - rdflib (4.2.2)
  - requests (2.18.4)

2.2. Execution (not including enrichment)

The `bin/generate.sh` script runs the following tasks in the correct order. Paths in this section are relative to `gate-extras/knowmak` unless otherwise indicated.

1. Build the ontology (classes, labels, properties etc.): `bin/generate-ontology.py`

   This creates a basic RDF graph and reads the `KET`, `SGC` worksheets of `data/ontology.xlsx` to generate classes and property values. It uses the columns listed in Table 1 and requires the spreadsheet to be arranged in a specific way, with each class's information beginning on a separate line. Appendix A provides a sample.

   Class names can be duplicated in different places for multiple inheritance and to assign multiple property values (such as labels).

[1] `https://github.com/johann-petrak/gatetool-runpipeline.git`

column	use
A	class1
B	class2
C	class3
D	class4
E	prefLabel
F	label
G	extraKeywords
H	provenance
I	h2020objective
J	description
K	IPC code
L	projectSubjects

Table 1: Columns in the taxonomy spreadsheet

Values in columns E–L are applied to the most recently named class (on the same or a previous line). The rightmost non-blank class cell (in columns A–D) is always the current class. Indentation by columns is used to create subclasses.

It reads the `IPC` worksheet and sets `knowmak:patentKeywords` properties on classes with matching IPC codes. It also reads the `provenance` worksheet and adds values of `knowmak:provenance` to appropriate classes; and reads the `topics` worksheet to add topic ID values to each class. Any other worksheets in the file are ignored.

It also reads the file `data/extra-keywords-classes.xlsx` and uses the two columns of every sheet whose name ends in `-keywords` to add `knowmak:keywords` values to the classes.

This program produces the following outputs.

`ontologies/knowmak.rdf` The ontology in N3 format.

`ontologies/build/knowmak.json` This is an incomplete version of the JSON agreed for use in the faceted and keyword search demos. (The complete version is built after enrichment.)

`ontologies/gazetteer-src/descriptions.xml` An XML file containing the string values of the properties in Table 2 for each class, mapped to the listed flags. Each string is wrapped in a `p` tag with the following attributes:

- `uri` the class URI;
- `flags` the flag mapped from the property;
- `topicID` the topic ID number for the class;
- `property` the property;
- `leaf` the string `true` or `false` according as the class is a leaf node in the class hierarchy or not;

property	flag
rdf:Description	generated
skos:definition	generated
rdfs:label	preferred
skos:prefLabel	preferred
knowmak:keywords	key
knowmak:patentKeywords	patent
knowmak:projectKeywords	project

Table 2: Properties used for the description document

- provenance the provenance value for the class.

The output is written to ontologies/gazetteer-src/descriptions.xml, which should be the only file in that directory because the next tool always processes all the files it finds in a specified directory. This output is not in GATE XML but is structured so that the application can process it easily.

ontologies/knowmak.tree.txt A plain-text tree of the whole class hierarchy, using three-space indentation to indicate depth and subclass, with one URI per line. Some classes can appear more than once, sometimes at slightly different depths. This and the following simple version are used for documentation and description of the ontology.

ontologies/knowmak.tree-simple.txt The same thing but with only the last (distinctive) part of each class's URI.

ontologies/depth_test.tsv A TSV file listing for each ontology class its top class (KET, SGC, or Mission), the final part of its URI, and a space-separated list of the different depths at which it appears in the ontology (with KET and SGC as 0).

ontologies/depth_tables.tsv A file listing the number of classes that appear at each depth in the class tree, separated by top class. The first table counts each class only at its highest level (lowest number); the second counts each class at each depth at which it occurs. This file is used for documentation and description of the ontology.

2. Ontology population (creation of the classifier gazetteer)
```
../../gatetool-runpipeline/bin/runPipeline.sh
-r applications/gazetteer_generator.gapp
ontologies/gazetteer-src/
```

This runs the GATE application to generate the basic classifier gazetteer by extracting keywords and phrases from the plain-text of the previous stage's XML output (descriptions.xml) and associating them with information in the attributes of the enclosing p tags. The application consists of the following PRs.

- Document reset.
- ANNIE English tokenizer.
- ANNIE sentence splitter.
- ANNIE POS tagger (this uses the same non-standard lexicon as described in §§3.2.3 and 3.2.4, although it is probably not necessary here).
- GATE morphological analyser (for lemmatization).
- Flexible gazetteer configured to run over `Token.root` using `resources/gazetteer/stopwords.def` and the `stopwords-*.lst` files that it refers to. Note that the stopwords can include multi-token sequences (i.e., stop-phrases).
- JAPE to create `Stop` annotations over `Lookup.majorType==stop` and both `Stop` and `Unit` annotations over `Lookup.majorType==unit`.
- JAPE to turn `p` tags with `flags=preferred` or `key` into `Preferred` annotations; and other `p` tags into `General` annotations.
- Segment-controlled JAPE to assemble all the non-Split tokens in each `Preferred` annotation into a `MultiWord`.
- Segment-controlled JAPE in the `General` annotations.
  - JAPE to mark single-word term candidates, borrowed from TermRaider.
  - JAPE to mark multi-word term candidates, borrowed from TermRaider and adapted empirically for broader coverage.
  - JAPE to create `Knowmak` annotations over term candidates if they meet the following criteria:
    - single-word term candidates that are not covered by a `Stop` annotation;
    - multi-word term candidates that are not covered by a `Stop` annotation and do not end with a `Unit` annotation.

    This is designed so that the unit stopword *nanometre*, for example, would block *nanometre*, *100 nanometres*, and *few nanometres*, but would allow *nanometre-resolution topographic map* and *nanometre-scale structure*.
- The final Groovy PR scans the document for `Knowmak` annotations, and associates each one's `canonical` feature (downcased lemma or sequence of lemmata, as in TermRaider) as a keyword or phrase with the ontology class in its containing `p` annotation's `uri` feature (from `Original markups`). It writes the results in tab-separated gazetteer format[2] to the file `resources/gazetteer/knowmak-ontology.lst`, which is used by the classifier application described in §3.2.3. The key-value pairs are `topicID`, `flags`, `provenance`, `property`, `leaf` copied from the XML attributes described above.

[2] `keyword or phrase<TAB>key0=value0<TAB>key1=value1....`

The resulting file can be used in a GATE gazetteer, but contains only keywords and phrases extracted from the string properties of the ontology classes.

2.3. Enrichment

The generation of an enriched set of keywords is carried out on a separate server with the `gate-extras` subversion checkout, using `knowmak-ontology.lst` (the unenriched gazetteer) and a separate corpus as input to produce three TSV files as output. This process is documented separately.[3]

After this has been done, the resulting set of enriched keywords needs to be integrated with the ontology. This is done as follows.

1. Create the new enriched keyword gazetteer:
   `bin/enrich-gazetteer.py`

   This uses the following files as input:

   - `resources/gazetteer/knowmak-ontology.lst` the unenriched gazetteer generated earlier;
   - `data/enrichment/combined.tsv` enrichment scores for those keywords in the unenriched gazetteer that are also found in the enrichment corpus and for new keywords derived from the corpus, in three tab-separated columns (term, class URI, score);
   - `ontologies/knowmak.json` the JSON file produced earlier;
   - `ontologies/keywords-blacklist.tsv` a file of class-keyword combinations to suppress from the output, in two tab-separated columns (class URI, keyword).

   Class-keyword combinations in the blacklist are removed from the gazetteer data before the output is written. They do **not** apply to entries with `key` or `preferred` flags. The same keyword string can be blacklisted for some classes but not for others.

   This process produces one output file, `resources/gazetteer/knowmak-enriched.lst`, the enriched gazetteer.

   The output gazetteer is written in the same TSV format as the original one, with the following key-value pairs for each term:

   - `uri` the class URI;
   - `flags` a comma-separatied list of the flags from the original gazetteer plus `enrichment` if the term is also found in the enrichment output, or just `enrichment` for new terms;
   - `topicID` the topic ID number for the class;

[3] `https://github.com/johann-petrak/knowmak-doc2onto-semsim/blob/master/documentation/iteration-1806/iteration-1806.pdf`

- `property` the property from the original gazetteer (this pair is not used for enrichment-only terms);
- `leaf` the string `true` or `false` according as the class is a leaf node in the class hierarchy or not;
- `provenance` the provenance value from the original gazetteer, or `enrichment` if the term was added by the enrichment process;
- `score` the score produced by the enrichment, or (for terms that were not found in the enrichment corpus) the mean of all the enrichment scores.

2. Use the new gazetteer to create the final JSON version of the ontology: `bin/make-enriched-json.py`

   Input:

   - `ontologies/build/knowmak.json` (not checked into svn)
   - `resources/gazetteer/knowmak-enriched.lst`

   Output:

   - `ontologies/knowmak.json` the final JSON file for use in the faceted and filtered search demos, including all the the original keywords and those added by the enrichment.

3. REST service for document classification

This section describes the classification tool which takes a document (such as a project, patent, or publication) as input and returns relevant topics and keywords with scores.

3.1. Building and deploying the service

3.1.1. Requirements

- Java 8 (tested with Oracle 1.8.0)
- Grails 3 (this does not need to be installed if the `grailsw` script wrapper is used)
- ant (tested with Apache Ant 1.10.3)
- maven (tested with Apache Maven 3.5.0)
- Apache Tomcat 8 on the server (tested with version 8.0.37)

3.1.2. Instructions

3.1.2.1. Remote build The following instructions assume a subversion check-out on your local machine.

1. Run the main ant build (`knowmak/build.xml`) to build the KnowmakScoring plugin (using a subsidiary build file), repackage `applications/classifier.gapp` and its resources and plugins into `services/knowmak/src/main/resources/gate-files/`; use `./grailsw war` to build the war file; upload the war file to the server.

   ```
   cd gate-extras/knowmak
   ant
   cd services/knowmak
   ./grailsw war
   scp  ./build/libs/knowmak-0.1.war.original \
        gateservice10:~/knowmak.war
   ```

   The service can also be tested locally with `./grailsw run-app`. The `.war.original` file contains everything needed to run the service in Tomcat; the `.war` file also contains an embedded server and can be run with the `java -jar` command.

2. Log into the server, stop Tomcat, remove the old version of the service, unpack the new one, and restart Tomcat. Be careful to run `rm -fr` *inside the correct directory.* It is important to unpack the war file manually into the correct directory (if Tomcat unpacks it, the ownership and permissions issues mean that Ian will have to fix it the next time we want to update the service). The files also need to have `gate` as the group owner with write permissions. The `yml` configuration file checked into svn has a small number of threads; the service running on the server needs a configuration file with `applicationPoolSize` at the end set to `15`.

```
ssh gateservice10
sudo /sbin/stop tomcat-knowmak-dev
cd /opt/tomcat-knowmak-dev/webapps/knowmak#classifier
rm -fr *
unzip ~/knowmak.war
cp ../application.yml WEB-INF/classes/application.yml
chown -R .gate .
chmod -R a+rX .
chmod -R g+w .
sudo /sbin/start tomcat-knowmak-dev
```

The example above is for updating the *test service*. Change each occurrence of `tomcat-knowmak-dev` to `tomcat-knowmak` to update the production service.

3.1.2.2. Server build It is easier and faster to build the whole thing on the server.

```
ssh gateservice10
# following only needed the first time
svn co svn+ssh://gateservice2.dcs.shef.ac.uk/data/svn/gate-extras-svnrep/trunk/knowmak
cd knowmak
svn up
ant
cd services/knowmak
./grailsw war
sudo /sbin/stop tomcat-knowmak-dev
cd /opt/tomcat-knowmak-dev/webapps/knowmak#classifier
rm -fr *
unzip ~/knowmak/services/knowmak/build/libs/knowmak-0.1.war.original
cd ..
# you should now be in the webapps directory
./fix-perms.sh
sudo /sbin/start tomcat-knowmak-dev
```

As before, this is for updating the *test service*.

3.1.3. Log files

The logs are on `gateservice10` in `/opt/tomcat-knowmak/logs/` and `/opt/tomcat-knowmak-dev/logs/` for the main and test services respectively. The messages generated by the grails service and the embedded GATE application appear in `classifier-YYYY-MM-dd.log.gz` files, except for the current one, which is not yet gzipped. These can be grepped and piped to find out start and end times and number of documents processed, as in the following examples.

```
$ egrep -oh 'type=\S+' classifier-2019-03-07.log |sort |uniq -c
 799506 type=publication
```

```
$ for X in `ls classifier-2019-03-*gz` ; do gunzip -c $X \
   | egrep -oh 'type=\S+' |sort |uniq -c  ; done
   4582 type=project
      7 type=publication
      1 type=knowmak
      2 type=publication
      2 type=knowmak
  71252 type=project
      1 type=projects
 413067 type=publication
     30 type=project
1005904 type=publication
1684672 type=publication
```

3.1.4. Old log files

The messages generated by the grails service and the embedded GATE application used to appear in `catalina.out.YYYY-MM-dd-HH_MM_SS` files. The following examples show how to grep start and end times and number of documents processed.

```
$ grep -h ' publication ' catalina.out.2018-07-* |head -2
2018-07-01 01:21:29.852  INFO --- ... : \
  5017ccf4-0c9a-4695-aaf7-ed16469c8722  publication  000363436000016  text/plain
2018-07-01 01:21:30.338  INFO --- ... : \
  f9495169-85bd-40eb-823c-457289ed97f5  publication  000363436300002  text/plain

$ grep -h ' publication ' catalina.out.2018-07-* |tail -2
2018-07-16 09:11:23.847  INFO --- ... : \
  8cf28f2d-01be-412f-abe3-af2e22ff54b4  publication  000289446900023  text/plain
2018-07-16 09:11:24.635  INFO --- ... : \
  26247afa-61f0-4a14-881b-26de271956f8  publication  000289446900024  text/plain

$ grep ' publication ' catalina.out.2018-07-* |wc -l
4300812
```

The spaces in the grep argument are used because document names supplied by the clients sometimes contained words like *patent*, *publication*, or *project*.

3.1.5. Proxying

Ian has set proxying to redirect these URLs

`http://services.gate.ac.uk/knowmak/classifier/<docType>/<docId>`
`http://services.gate.ac.uk/knowmak/classifier/<docType>`

`http://services.gate.ac.uk/knowmak/classifier-dev/<docType>/<docId>`
`http://services.gate.ac.uk/knowmak/classifier-dev/<docType>`

(where `<docType>` and `<docId>` are any valid URL path elements) to the `tomcat-knowmak` service on `gateservice10`. The second pair of URLs points to the test service.

3.2. Internals

3.2.1. Initialization

The service loads the GATE application from the location defined in `grails-app/conf/spring/resources.xml`, relative to the `main/resources` directory.

It then duplicates that application to make a pool of them which can be used to process documents in parallel threads. The pool size is determined by the value of `knowmak.applicationPoolSize` configured near the end of the `grails-app/conf/application.yml` file (as mentioned in §3.1.2, this needs to be changed on the production server for a larger pool).

3.2.2. Controller and service

The `UrlMappings.groovy` file defines the following:

```
"/$docType/$docId?" (controller: "knowmak", action: "process")
```

so that the controller's *process* method has access to the `docType` and `docId` parameters. If the last slash and `docId` are not provided, the latter's value will be `null` when passed to the controller.

The controller is very simple and has one method, which replaces a null `docId` parameter with the string `_none_`, generates an internal identifier (a UUID, used in the server logs for consistency in debugging), and calls the processing service's method (passing it the request's body and `Content-Type` header, the `docType` and `docId` parameters, and the internal ID). The `docType` parameter is required.

The processing service creates a GATE document from the request body (using the specified MIME type and encoding if they are available), and sets the following document features:

- `knowmakType`: the `docType` from the URL;
- `identifier`: the `docId` from the URL (or the `_none_` default);
- `internalID`: the UUID.

It then executes the GATE application (described in §3.2.3) over the document and picks up the `output` and `internalID` document features to generate the output, which the controller renders as JSON in the response body. (The JSON format is explained in §3.3.3.)

3.2.3. GATE application

The classifier application is a Groovy scriptable controller consisting of the following PRs. They currently all run in the same order every time, but future versions will carry out conditional processing based on the `knowmakType` document feature.

1. Before the PRs run, the control script checks the `knowmakType` document feature and copies it into the `matchedType` document feature if it matches any of the strings *patent*, *project*, or *publication*.

2. Document reset.

3. ANNIE English tokenizer.

4. ANNIE sentence splitter.

5. ANNIE English POS tagger using a special lexicon file for consistent treatment of all-uppercase and normally cased texts (see §3.2.4 for details).

6. GATE morphological analyser (for lemmatization).

7. ANNIE gazetteer, JAPE grammars, and orthomatcher—used to mark named entities for exclusion later.

8. Extended gazetteer from the StringAnnotation plugin, configured to use the enriched gazetteer list (generated in §2.3, including the one generated in §2.2) over `Token.root` with case-insensitivity.

9. The `jape` JAPE grammar creates an `Entity` annotation over each ANNIE `Person`, `Location`, and `Organization`.

   It then converts each `Lookup` from the KNOWMAK gazetteer to a `Knowmak` annotation with the following features.

   - `uri`
   - `class` with the same value as `uri`, provided for ontology-aware JAPE use.
   - `topicID`
   - `gaz_score` the score from the gazetteer.
   - `multiplier` 2.0 if the document type is included in the match's flags value (i.e., a patent-related keyword in a patent document, or a project-related keyword in a project document); or 1.1 if the flag's value includes "preferred"; otherwise 1.0.
   - `base_score` the gazetteer score times the multiplier.
   - `string` the `_string` feature from the gazetteer match.
   - `kind` a copy of the flags value.

   It will not create a `Knowmak` annotation inside a larger, covering `Entity` annotation or a larger, covering `Lookup` annotation from the Knowmak gazetteer.

10. The combined *knowmak scoring* Java PR[4] carries out the following functions.

[4] `plugins/KnowmakScoring`

a) It looks up pairs of matches in the document in the `resources/scoring/pairs.npmi.tsv` table (which is loaded once for all instances of the PR in memory). The PMI used for each term is the highest value in the table for any of its pairs found in the document, or 0 if no pairs in the document are found in the table. It then adds to each `Knowmak` annotation two features:

- `pmiBoost` with a value of $1 + pmi$, and
- `pmiWith` containing the other term from the highest scoring pair, or an empty string if no matching pairs were found in the table.

b) It catalogues and scores all the `Knowmak` annotations in the document by their `uri` features. Each keyphrase gets two scores: $100\frac{bp}{d}$ where b is the `base_score` feature, p is the `pmiBoost` feature, and d is the length of the document (in tokens); and $100\frac{b}{d}$ (without the PMI boost). (Note that $p \geq 1$ always.)

Each ontology class found in the document gets two scores (with and without PMI boosting), according to the sum of its matches

$$\sum_{m_i \in M(c)} 100\frac{b_i p_i}{d}$$

and

$$\sum_{m_i \in M(c)} 100\frac{b_i}{d}$$

where $M(c)$ is the set of `Knowmak` annotations for class c. Each class is also associated with a list of keyphrases (each with individual scores). One keyphrase in the document (several `Knowmak` annotations with the same span but different values of the `uri` feature) can contribute to more than one ontology class, and the base scores can be different for different classes.

c) It builds a table of both kinds of total class scores for the document and scans it for classes with non-zero scores whose direct superclasses also have non-zero scores. Both scores for each qualifying class are boosted by $\frac{1}{2}$ of the score of the highest-scoring direct superclass.

The final output is a Java/Groovy `Map` with the following keys:

- `classification`: the results of the class, keyphrase, and score processing, including the following for each class:
  - `score` the two final scores after boosting from superclasses;
  - `unboostedScore` the two final scores without boosting from superclasses (this is the same as the score whenever no direct superclass is matched);
  - `boostedBy` the URI of the direct superclass used (this key-value pair is absent if the score is not boosted);
- `identifier`: the `docId` passed in the URL (or `_none_`);

input	output	comment
`PANTS VBS`	`PANTS NNS`	overwritten from `pants`
`Pants NNPS`	`Pants NNPS`	
`pants NNS`	`pants NNS`	
`PANT VB`	`PANT NNP`	overwritten from `Pant`
`Pant NNP`	`Pant NNP`	
`pant VB`	`Pant NNP`	

Table 3: Example of POS lexicon modification

- `docType`: the `docType` passed in the URL;
- `matchedType`: the `matchedType` feature mentioned above or `_none_`.

This is stored in the `output` document feature, collected by the service, and forms most of the REST service's JSON output (as described in §3.3.3).

3.2.4. Special POS-tagging lexicon

The classifier's POS tagger uses a specially modified version of the usual English lexicon in order to produce consistent POS-tagging and lemmatization between documents with normal case and those that have been completely upcased.

The program `bin/tweak_pos_lexicon.py` reads `resources/heptag/annie.lexicon` and classifies all the entries according to whether the token is all-uppercase, mixed-case, or all-lowercase. It keeps any all-uppercase entries that do not have mixed-case or all-lowercase versions, overwrites all-uppercase entries with upcased versions of mixed-case entries where they exist, then overwrites all-uppercase entries with upcased versions of lower-case entries, as shown by the examples in Table 3. The result is written to `resources/heptag/lexicon`, which is checked into svn so the tweaking program should not be needed again.

3.3. Using the service

3.3.1. Endpoints

We expect the following POST endpoints to be used:

```
http://services.gate.ac.uk/knowmak/classifier/project/<docId>
http://services.gate.ac.uk/knowmak/classifier/project
http://services.gate.ac.uk/knowmak/classifier/patent/<docId>
http://services.gate.ac.uk/knowmak/classifier/patent
http://services.gate.ac.uk/knowmak/classifier/publication/<docId>
http://services.gate.ac.uk/knowmak/classifier/publication
```

but the service will actually accept a POST request to any URL in either of the following formats:

`http://services.gate.ac.uk/knowmak/classifier/<docType>/<docId>`
`http://services.gate.ac.uk/knowmak/classifier/<docType>`

where `<docType>` and `<docId>` are valid URL path elements. The required `docType` will be passed as a string to the GATE application, which will use it for conditional processing to handle *project*, *patent*, and *publication* documents slightly differently (by scoring up relevant vocabulary matches and ignoring certain irrelevant ones). The system is designed to accept any `docType` string in order to allow for testing and future expansion; an unexpected type is flagged in the output (in the `doc_type_applied` feature, as described in §3.3.3).

3.3.2. Input data

The service can accept any format that GATE can process by default[5], in particular:

- plain text,
- HTML (including XHTML),
- XML and SGML,
- RTF, and
- some Microsoft Office and OpenOffice formats.

The service will work much more reliably if the client supplies the correct `Content-Type` header.

In practice, the other project partners have developed tools that read the documents from their own databases and send plain text to the classifier.

[5] `https://gate.ac.uk/sale/tao/splitch5.html#sec:corpora:formats`

3.3.3. JSON annotation output format

The output is a JSON map with the following keys.

- `classification` The value is a map whose keys are ontology classes; each value is a map with the following keys.
  - `score` A list of two float values, showing the scores with boosting from superclasses for this class (without and with PMI boosting).
  - `unboostedScore` A list of two float values, showing the scores without boosting from superclasses for this class (without and with PMI boosting).
  - `boostedBy` This has a string value with the URI of the boosting superclass. If the value is the empty string, no direct superclass was found and the boosted scores are the same as the unboosted ones.
  - `keywords` The value is a map whose keys are keywords (including multi-word phrases); each value is a float showing that keyword's contribution to the class score (base score multiplied by number of occurrences, without PMI). Note that the same keyword may contribute to more than one class and can have different scores for different classes.
- `doc_type` The string value shows the `DOCTYPE` parameter passed in the endpoint URL (see §3.3.1).
- `doc_type_applied` The string value shows the `DOCTYPE` parameter if it was validated by the service[6] or `_none_` if it was invalid.
- `error` The string value shows an error encountered within the GATE application while processing the document, or `_none_` for no error.
- `identifier` The string value shows the `DOCID` parameter if it was provided in the endpoint URL, or `_none_`.
- `internalID` The string value shows an internal identifier generated within the service and used for debugging (this ID appears in the server log files).

Appendix B gives an abridged example.

4. Quick guide to updating the service

Follow the instructions in §4.1 and then either §4.2 or §4.3.

[6] As mentioned above, current valid values are *paper*, *project*, and *publication*.

4.1. Applying ontology updates

Refer to §2 for a detailed explanation of how the following works.

```
cd path/to/gate-extras/knowmak
svn up
bin/generate.sh
bin/enrich-gazetteer.py -s
bin/make-enriched-json.py
```

Depending on what has changed, this can alter some or all of the following files that are checked into subversion. (Some svn differences can occur even if nothing has been changed, because one value of *provenance* is chosen non-deterministically for the gazetteers when a class has more than one in the ontology.)

```
ontologies/gazetteer-src/descriptions.xml
ontologies/knowmak.json
resources/gazetteer/knowmak-enriched.lst
resources/gazetteer/knowmak-ontology.lst
```

To build the war file remotely (§4.3), the changed output files need to be checked in (`svn ci`).

4.2. Building the war file locally and deploying it on the server

Refer to §3.1 for a detailed explanation of how the following instructions work.

```
cd path/to/gate-extras/knowmak
ant
cd services/knowmak
./grailsw war
scp  ./build/libs/knowmak-0.1.war.original \
      gateservice10:~/knowmak.war
```

Then log into the server and deploy it. The following instructions are for the test service; replace every occurrence `tomcat-knowmak-dev` with `tomcat-knowmak` to deploy the production service.

```
sudo /sbin/stop tomcat-knowmak-dev
cd /opt/tomcat-knowmak-dev/webapps/knowmak#classifier
rm -fr *
unzip ~/knowmak.war
cd ..
# you should now be in the webapps directory
./fix-perms.sh
sudo /sbin/start tomcat-knowmak-dev
```

4.3. Building the war file on the server and deploying it there

Refer to §3.1 for a detailed explanation of how the following instructions work.

The following instructions assume that all changes have been checked into svn. They are for the test service; replace every occurrence `tomcat-knowmak-dev` with `tomcat-knowmak` to deploy the production service.

```
ssh gateservice10
# following only needed the first time
svn co svn+ssh://gateservice2.dcs.shef.ac.uk/data/svn/gate-extras-svnrep/trunk/knowmak
cd knowmak
svn up
ant
cd services/knowmak
./grailsw war
sudo /sbin/stop tomcat-knowmak-dev
cd /opt/tomcat-knowmak-dev/webapps/knowmak#classifier
rm -fr *
unzip ~/knowmak/services/knowmak/build/libs/knowmak-0.1.war.original
cd ..
# you should now be in the webapps directory
./fix-perms.sh
sudo /sbin/start tomcat-knowmak-dev
```

5. Annotation thresholds

Because the GATE classification tool gives scores for every possible class match, a threshold needs to be set for determining which of these should be accepted. A single strategy is now used for all document types, as follows:

- Out of the 4 scoring options (with or without PMI, and with or without boosted scoring), the option with both PMI and the boosted score should be used.
- Based on this score, the final document-class assignment is based on the following restrictions:
  - More than a single keyword must be allocated for the class After single-keyword classes have been removed:
    - The class score must be higher than 2x the mean of the class in the whole corpus
    - The class score must be about 0.8x the highest class score for the document

6. Additional tools

The following tools are found in `gate-extras/knowmak/bin`; they require Python 3 and the requests and rdflib libraries.

- `client.py`

  Command-line client for processing files through the REST service. Use `-h` for an explanation of the options and arguments. By default this connects to the production service but options are provided for a local `grails run-app` service, for the test service, and for customizing the URL.

- `demo.py`

  GUI client for processing files through the REST service and displaying the JSON output in a scrollable text box. Use `-h` for an explanation of the options and arguments. By default this connects to the production service but options are provided for a local `grails run-app` service, for the test service, and for customizing the URL.

- `generate-ontology-matrix.py`

  This concatenates the descriptive string properties of each ontology class and runs each class's set through the REST service, then puts the classification scores in a sparse table (mostly 0 values), which is written in CSV format to `ontology/matrix.csv`. The table has one row and column for each ontology class, and each numeric cell indicates the row-class's strings' score using the column-class's keywords.

  It also stores the top n matches and writes the results in a 3-column CSV file at `ontologies/matrix-top-n.csv`.

  Use `-h` for an explanation of the options and arguments. By default this connects to the production service and the second file contains the top 5 matches.

- `ontomak.py`

  This is a Python module containing shared code, constants, etc., used by the other programs here and in §2.2.

- `samples`

  This directory contains sample documents for demonstrating the service.

- `test_server.py`

  This carries out a set of tests on the service using various URLs, to ensure that they are all working as expected.

- `matrix2network.py`

  This reads the comparative matrix CSV file mentioned above and generates a JSON file containing a (somewhat) simplified representation of the similarity network for the visualization demo.

- `graph.py`

  This is a simple graph library used by the `matrix2network` program.

- `fishiness-reports.py`

  This reads the comparative matrix CSV file mentioned above and the `knowmak.json` file, and generates two CSV reports used to locate and identify dubious keywords.

  - `ontologies/keyword-collisions.csv` lists every keyword and the number of classes that refer to it, in descending order of the latter.
  - `ontologies/matrix-analysis.csv` lists every non-zero, non-diagonal matrix entry with the following fields:
    * source class;
    * target class;
    * score of the source description against the target class (the list in sorted in descending order by this field);
    * score of the source description against the source class (for comparison); and
    * intersection of the source's and target's sets of keywords.

A. Spreadsheet input example

A	B	C	D	E	F	G	H	I	J	K	L
class1	class2	class3	class4	prefLabel	label	extra keywords	provenance	h2020 objective	description	IPC code	project subjects
Advanced Manufacturing Technology				advanced manufacturing technology					AMT encompasses the use of innovative technology. . .		AUTOMATION
											ICT APPLICATIONS
											INFORMATION PROCESSING, INFORMATION SYSTEMS
											NETWORK TECHNOLOGIES
											ROBOTICS
	AMT Biotechnology			manufacturing biotechnology	biotechnology for manufacturing						
		DNA/RNA technologies		DNA/RNA technologies	RNA technologies						
		DNA/RNA technologies		DNA/RNA technologies	DNA technologies						
		Molecular biotechnologies		molecular biotechnologies							
		Cell and tissue engineering		cell and tissue engineering	cell engineering						

B. JSON output example


```
{
  "classification": {
    "http://www.gate.ac.uk/ns/ontologies/knowmak/antibiotics": {
      "boostedBy": "http://www.gate.ac.uk/ns/ontologies/knowmak/antimicrobials",
      "keywords": {
        "antibiotics": {
          "kinds": [ "generated", "preferred" ],
          "score": 1.1527377521613833
        },
        "bacteria": {
          "kinds": ["generated"],
          "score": 0.5763688760806917
...
      },
      "score": [ 4.322766570605188, 4.4159785333 ],
      "topicID": "38",
      "unboostedScore": [ 2.5936599423631126, 3.75354899915 ],
    },
    "http://www.gate.ac.uk/ns/ontologies/knowmak/antimicrobial_resistance": {
      "boostedBy": "http://www.gate.ac.uk/ns/ontologies/knowmak/antimicrobials",
      "keywords": {
...
        }
      },
      "score": [ 8.069164265129682, 9.12545454545 ],
      "topicID": "42",
      "unboostedScore": [ 6.340057636887607, 7.35454545454 ]
    },
    "http://www.gate.ac.uk/ns/ontologies/knowmak/antimicrobials": {
      "keywords": {
...
        }
      },
      "score": [ 3.4582132564841506, 4.54545452388 ],
      "topicID": "43",
      "unboostedScore": [ 3.4582132564841506, 4.54545452388 ],
    },
 },
  "doc_type": "publication",
  "doc_type_applied": "publication",
  "error": "_none_",
  "identifier": "12348874",
  "internalID": "4dec1ce0-cead-4a94-b16c-6fada1a26f49"
}
```