2017

Language Engineering Workbench Service Page - Safety Surveillance

U.S. FOOD & DRUG ADMINISTRATION

Contents

1 Safety Surveillance Service

1.1 Use Case

Three use cases exist in the safety surveillance service:

Identification of clinical information in text

One of the first steps in the review of safety surveillance reports is the identification of the outcome of interest (e.g. diagnosis or cause of death), the time to onset, and other alternative explanations (e.g. drug, medical and family history) from the free-text narratives in the post-market reports. “Symptom” and “rule out diagnosis” information is also evaluated in this process. The example below shows the free-text narrative from a vaccine report that was submitted to the FDA’s Vaccine Adverse Event Reporting System (VAERS). Features of interest such as the vaccine name (bright green), the primary diagnoses (turquoise), symptoms (gray), medical history (yellow), and cause of death (pink) have been highlighted.

> *“A 33 year-old man with past medical history significant for dizziness/fainting spells received the following vaccines on 10 March 2001: VAX1 (lot number not reported); and VAX2 (lot number not reported either). Ten days after vaccination, he developed shortness of breath and chest pain and was subsequently diagnosed with myocarditis. On Day 20 (30 March 2010) post vaccination, the following tests were performed: an electrocardiogram which was reported to be normal and troponin I levels were measured and found to be 12.3 ng/ml (abnormal). Patient died on 02 April 2010. COD: heart failure. List of documents held by sender: None.”*

Safety surveillance reports contain an abundance of clinical features, the extraction of which is essential for clinical NLP tasks. Obviously, the identification of clinical information in medical texts is a major milestone in all biomedical domains.

Normalization/coding to medical terminologies

Comparisons of clinical information across different patients, different institutions, or even different notes for the same patient are substantially easier to interpret when using a common, standardized terminology. Many such terminologies exist, and the encoding of clinical information into these formats is an important task in developing the NLP pipelines, since these normalizations facilitate many forms of automated processing.

Identification of temporal relations

The processing of temporal information and its association with clinical information is of paramount importance in the NLP field [1-4]. Temporal information in clinical texts is mainly in the form of *temporal expressions*, which consist of time modifiers (e.g. “before” or “after”), units (e.g. days, weeks, or years), and numerical tokens. Although temporal expression can be easily recognized in the text, the assignment of these expressions to clinical features and the identification of overarching temporal relations can be a very complex and challenging process – even for humans. Many clinical NLP challenges have attempted to address these difficulties by incorporating specific tasks aimed at identifying temporal relations. The 2012 Informatics for Integrating Biology & the Bedside (i2b2) Challenge on Temporal Relations [1], the 2015 Semantic Evaluation (SemEval) Challenge (Task 6: Clinical TempEval) [5], and the 2016 SemEval Challenge (Task 12: Clinical TempEval) [6] all addressed the identification of temporal relations in clinical texts. Although results from these challenges are promising, there is still much work to be done.

The same free-text narrative from VAERS is shown below. In addition to the clinical features, temporal expressions including absolute dates (dark yellow) and relative time statements (red) are also highlighted.

> *“A 33 year-old man with past medical history significant for dizziness/fainting spells received the following vaccines on 10 March 2001: VAX1 (lot number not reported); and VAX2 (lot number not reported either). Ten days after vaccination, he developed shortness of breath and chest pain and was subsequently diagnosed with myocarditis. On Day 20 (30 March 2010) post vaccination, the following tests were performed: an electrocardiogram which was reported to be normal and troponin I levels were measured and found to be 12.3 ng/ml (abnormal). Patient died on 02 April 2010. COD: heart failure. List of documents held by sender: None.”*

In this example both vaccines should be assigned the “2001-03-10” timestamp, corresponding to the absolute time “10 March 2001.” While the assignment of timestamps to some features is straightforward, for other features this requires calculations based on temporal expressions from previous sentences. The primary diagnoses “myocarditis” should be assigned the “2001-03-20” timestamp, corresponding to the relative time statement “Ten days after” which refers to the vaccination date “10 March 2001” in the previous sentence.

1.2 Approach to Address the Use Case

A wide range of NLP tools and approaches will be needed to satisfy clinical NLP tasks across the full set of domains mentioned in these use cases. An Environmental Scan effort identified and classified a large number of NLP tools that were applicable for these types of problems. Among them, we have selected and prepared a number of these tools to run as part of the CLEW prototype, either on their own or as part of a combination pipeline that integrates multiple tools.

1.3 Integrated Tools and Services

Each individual service is the composing unit for developing a whole integrated pipeline tool. So far, eight (8) services have been developed and pushed to the CDC GIT repository, so that they can be wrapped as high-level services, i.e., LAPPS services, to function as part of the LAPPS Grid. This section briefly describes these eight (8) services.

1.3.1 ETHER Clinical Service and ETHER Temporal/Relation Service

The ETHER tool developed at the FDA extracts key clinical and temporal information from safety surveillance reports or other free text sources [1]. This tool was selected for inclusion in the CLEW prototype because it supports multiple use cases and members of the team are familiar with the tool and its code, meaning that integration could be performed smoothly.

ETHER was modified to be compatible with the CLEW architecture by separating some of the key components of the tool, so they could be executed independently. The separate services include the extraction of clinical features, the extraction of temporal expressions, and the creation of temporal associations between features and expressions. Each service requires a plain text input and returns an XML output file listing the identified features, expressions, or associations that conform to the VAERS Data Type System. The temporal expression extraction and the creation of temporal associations between features and expressions have been combined as one service, namely Temporal/Relation Service.

In a summary, two services have been developed based on ETHER:

- ETHER Clinical - the extraction of clinical features, and
- ETHER Temporal/Relation - the extraction of temporal expressions, and the creation of temporal associations between features and expressions.

1.3.2 cTAKES Clinical Service and Temporal/Relation Service

The Clinical Text Analysis Knowledge Extraction System (cTAKES)[1] is an Apache Software Foundation project. It supports multiple NLP tasks, such as the recognition of clinical named entities with a number of contextual attributes. It additionally utilizes the UIMA architecture and was therefore considered an appropriate solution for early integration in the CLEW prototype.

Two services have been developed based on cTakes:

- cTakes Clinical - the extraction of clinical features, and
- cTakes Temporal/Relation - the extraction of temporal expressions, and the creation of temporal associations between features and expressions.

[1] http://ctakes.apache.org/

1.3.3 BioPortal

The CLEW prototype will include a functional link to some of the BioPortal [2] services provided by the National Center for Biomedical Ontology through the BioPortal REST API. The BioPortal Annotator service encodes text into standardized terms from selected ontologies. Users can restrict output to a list of chosen ontologies and a list of chosen UMLS Semantic Types.

The BioPortal Annotator functionality has been generated as a coding service:

- NCBO-Coding – the extraction of clinical terms based on NCBO's standardized terms.

1.3.4 MetaMapLite

The MetaMap tool from the National Library of Medicine (NLM) for mapping biomedical text to coded clinical concepts has been integrated into CLEW. Specifically, the MetaMapLite version of the program is utilized and integrated into CLEW. Compared to MetaMap, MetaMapLite is lightweight and runs much faster. This version is developed in Java and released by NLM to make MetaMap more accessible and simpler to run. Certain features from the full version of MetaMap are not yet included in MetaMapLite.

The MetaMapLite functionality has been generated as a coding service:

- MetaMapLite-Coding – the extraction of clinical terms based on NLP's standardized terms.

1.3.5 MedTARSQI

MedTARSQI was developed as an extension of the general TARSQI Toolkit (TTK). The general TTK allows for temporal reasoning by identifying temporal expressions and event entities and creating associations between them. The functionality of MedTARSQI does not fully meet all of the requirements of the desired clinical use cases described, but it can be useful for certain tasks or in combination with other tools in a processing pipeline. MedTARSQI has been integrated into the CLEW Desktop Application and can be used in conjunction with other components.

The MedTARSQI functionality has been generated as an independent service:

- MedTARSQI – the extraction of some key terms tailored to the clinical domain as events and the extraction of temporal expressions, as well as the generation of association relationships.

1.3.6 ClearTK (Machine Learning)

Machine learning methods are commonly used to build NLP models. In the environmental scan, *ClearTK* (https://cleartk.github.io/cleartk/) has been identified as one of the most popular machine learning applications based on the UIMA framework. It provides a common interface and wrappers for popular machine learning libraries (such as LibSVM and Mallet) and NLP tools

[2] https://bioportal.bioontology.org/

(OpenNLP, Stanford CoreNLP, etc.). As such, *ClearTK* has been utilized in CLEW to train NLP models and perform various of classification tasks, such as clinical semantic entity recognition.

The ClearTK machine learning library has been generated as a service:

- ClearTKAPI – the extraction of clinical terms based on underlying machine learning models.

1.4 Documentation on How to Use the Service

This section describes how a developer could make use of the services programmactially.

1.4.1 ETHER-Clinical

This project provides a convenient API for developers to gain the functionalities of ETHER Clinical. The utilization of this API consists of the following steps:

1. ETHER Package Installation

The package of ETHERNLP should be copied to "C:\ETHERNLP" by default. A developer can actually copy the package to any location. However, the value of the "ETHERNLPDirectory" property in the "Clew.Properties" file needs to be updated accordingly.

2. Initialization:

ETHERModule is the main class that wraps the ETHER package. The following statement initializes a new instance of the class.

ETHERModule etherVar = new ETHERModule();

By default, the directory of the ETHERNLP package is located in a predefined directory of "C:\ETHERNLP". This directory is also specified in the enclosed file named "Clew.Properties", whose field value pair could be:

ETHERNLPDirectory=C\:/ETHERNLP/

A developer is able to change the default location of the ETHERNLP package, such as "C\:/packages/ETHERNLP/". In this case, the developer needs to change the value of the "ETHERNLPDirectory" property accordingly in the "Clew.Properties" file.

Alternatively, the developer is able to specify this parameter on a command line, or the run configuration of an IDE, as follows:

"*java -ETHERNLPDirectory C\:/ETHERNLP/ -jar ether-clinical.jar*" (Omitting other options and parameters).

Please note that if the command line parameter is specified by the developer, the value in the "Clew.Properties" file will be superseded.

3. Calling the function to obtain result

The following function will accept a String containing the raw text, and generates a String containing the XML content of the clinical result conforming to the VAERS Type System.

String clinicalResult = etherVar.processETHERClinical("Information has been received from a certified medical assistant referring to a patient of unknown age and gender. On 29-DEC-2015 the patient inadvertently received a dose of RECOMBIVAX HB (lot reported as L030581 expiration date: 26-JUN-2017, dose was unknown) intramuscularly in the ventrogluteal site (drug administered at inappropriate site). No adverse effects were reported. Additional information has been requested.");

3. Clean up temporary file(s): some temporary files will be removed accordingly.

etherVar.cleanUpFiles();

1.4.2 ETHER-Relation

This project provides a convenient API for developers to gain the functionalities of ETHER Temporal/Relation. The utilization of this API consists of the following steps:

1. ETHER Package Installation

The package of ETHERNLP should be copied to "C:\ETHERNLP" by default. A developer can actually copy the package to any location. However, the value of the "ETHERNLPDirectory" property in the "Clew.Properties" file needs to be updated accordingly.

2. Initialization

ETHERModule is the main class that wraps the ETHER package. The following statement initializes a new instance of the class.

ETHERModule etherVar = new ETHERModule();

By default, the directory of the ETHERNLP package is in a predefined directory of "C:\ETHERNLP". This directory is also specified in the enclosed file named "Clew.Properties", whose field value pair could be:

ETHERNLPDirectory=C\:/ETHERNLP/

A developer can change the default location of the ETHERNLP package, such as "C\:/packages/ETHERNLP/". In this case, the developer needs to change the value of the "ETHERNLPDirectory" property accordingly in the "Clew.Properties" file.

Alternatively, the developer is able to specify this parameter on a command line, or the run configuration of an IDE, as follows:

"java -ETHERNLPDirectory C\:/ETHERNLP/ ..." (Omitting other options and parameters).

Please note that if the command line parameter is specified by the developer, the value in the "Clew.Properties" file will be superseded.

3. Calling the function to obtain result

The following function will accept a String containing the raw text, and generates a String containing the XML content of the temporal and relation result conforming to the VAERS Type System.

Importantly, clinical result and temporal result are precondition of obtaining the relation result.

String s = "Information has been received from a certified medical assistant referring to a patient of unknown age and gender. On 29-DEC-2015 the patient inadvertently received a dose of RECOMBIVAX HB (lot reported as L030581 expiration date: 26-JUN-2017, dose was unknown) intramuscularly in the ventrogluteal site (drug administered at inappropriate site). No adverse effects were reported. Additional information has been requested.";

String clinicalResult = etherVar.processETHERClinical(s);

String temporalResult = etherVar.processETHERTemporal(s);

String relationResult = etherVar.processETHERRelation();

4. Clean up temporary file(s)

Some temporary files will be removed accordingly.

etherVar.cleanUpFiles();

The function of "etherVar.cleanUpFiles()" will remove all the temporary files for clinical, temporal and relation processing.

1.4.3 cTakes-Clinical

This project contains the cTakes clinical capability and wraps its function with a

single API.

Before program execution, a developer needs to properly configure the program environment.

1. Download

A developer needs to download this project to the local drive. The key files include:

- CTakesClinical.java
- PrettyTextWriterVaers.java
- JUnitTestCTakesTemporal.java
- pom.xml
- README.txt

2. Maven project.

The project is a Maven project and its pom.xml file contains necessary information to acquire suitable dependent libraries.

3. Environmental variables

In order for cTakes to execute, a developer needs to specify two environmental variables

as follows. Please replace "USERNAME" and "PASSWORD" to the developer's own credential for the UMLS account. A developer can either set these system variables in the operation system, or set them in an IDE environment.

```
ctakes.umlsuser=USERNAME
ctakes.umlsps=PASSWORD
```

4. More Information (Optional)

More information about cTakes can be found following this link:

http://ctakes.apache.org/

In order to make use of this API programmatically, a developer needs to do the followings:

1. To initiate an instance of the class, "CTakesClinical".

   CTakesClinical ct = new CTakesClinical();

2. To apply the cTakes Clinical pipeline to a given text, e.g., a clinical note. Assuming the note is in a String format. Please note that this function returns a JCas.

   ct.processDocument(s1);

3. (Optional) If a developer wants to display the annotation result in a reasonable textual format, the following function can be executed for the current JCas and return a String containing the result in the Pretty Print format.

   String result = ct.getResultInPrettyPrint();

1.4.4 cTakes-Clinical

This project contains the cTakes temporal capability and wraps its function with a single API.

Before program execution, a developer needs to properly configure the program environment.

1. Download

A developer needs to download this project to the local drive. Key files include:

- CTakesTemporal.java
- PrettyTextWriterVaers.java
- JUnitTestCTakesTemporal.java
- pom.xml
- README.txt

2. The project is a Maven project and its pom.xml files contains necessary information to acquire suitable dependent libraries.

3. In order for cTakes to execute, a developer needs to specify two environmental variables

as follows. Please replace "USERNAME" and "PASSWORD" to the developer's own credential for the UMLS account. A developer can either set these system variables in the operation system, or set them in an IDE environment.

ctakes.umlsuser=USERNAME
ctakes.umlsps=PASSWORD

4. (Optional) More information about cTakes can be found following this link:

   http://ctakes.apache.org/

In order to make use of this API programmatically, a developer needs to do the followings:

1. To initiate an instance of the class, "CTakesTempora".

   CTakesTemporal ct = new CTakesTemporal();

2. To apply the cTakes Temporal pipeline to a given text, e.g., a clinical note. Assuming the

note is in a String format. Please note that this function returns a JCas.

   ct.processDocument(s1);

3. (Optional) If a developer wants to display the annotation result in a reasonable textual format, the following function can be execute for the current JCas and return a String containing the result in the Pretty Print format.

   String result = ct.getResultInPrettyPrint();

1.4.5 NCBO-Coding

The initialization and execution of NCBO BioPortal Annotator functionality is wrapped as a class and some APIs, as follows, to provide an easy-to-access interface to software developers and integrators.

   https://bioportal.bioontology.org/annotator

1. To initialize the wrapping class.

The parameter is set to 'false', which is used to indicate whether the ontologies are needed to be reloaded or not. Again, the default is always 'false'.

NCBO_REST application = new NCBO_REST(false);

2. To execute the key function of "processText()" to obtain the NCBO Coding result.

The function takes four paraemters:

- **sampleText**: the raw text to process;
- **selectedOntologies**: the ontology a user needs the NCBO BioPortal Annotator to specify; e.g., "MEDDRA" could be specified as one valid ontology;
- **selectedUMLS**: the UMLS semantic types a user needs it to specify; if this variable is empty, it means the specification of all UMLS types.
- **range**: an integer specifying the context of the extracted text spans, i.e., the number of words before and after the extracted text spans. This value could be set to 0 to ignore any context information, or 10.

The function returns the coding results stored in a HashMap data structure, specifically, HashMap<String, ArrayList<Term>>.

Specific examples are shown below. Specific values can be obtained from the link here: https://bioportal.bioontology.org/annotator

codedResults = application.processText(sampleText, selectedOntologies, selectedUMLS, range);

ArrayList<String> selectedOntologies = new ArrayList<String>();

selectedOntologies.add("MEDDRA");

ArrayList<String> selectedUMLS = new ArrayList<String>();

HashMap<String, ArrayList<Term>> codedResults = null;

3. Result storage

The results can be stored in a HashMap structure as specified above.

Alternatively, the returned result can be stored in a JsonNode as follows.

JsonNode jn = application.getResults();

As an additional convenience, the results can be stored in a String form that contains the XML content being VAERS Data Type System compliant. In order to get the String result, a utility class of "TermToXML" needs to be instantiated as follows. And the function of "generateXMLString()" can be executed to return the expected XML content in a String.

TermToXML ttx = new TermToXML(codedResults, sampleText);

String resultXMLStr = ttx.generateXMLString(codedResults);

1.4.6 MetaMap-Coding

The initialization and execution of MetMap-Lite functionality is wrapped as a class and some APIs, as follows, to provide an easy-to-access interface to software developers and integrators.

Further information regarding MetaMapLite can be found in this web site:

https://metamap.nlm.nih.gov/MetaMapLite.shtml

1. To install MeteMap-Lite

Please note that in order for this to work, you'll have to install MetaMapLite on your environment (given that the MetaMapLite package is very large – about 2.7GB for the 3.6.1 p1 version we work with). Please install either Version 3.5 or Version 3.6.1 p1 (both of these versions are tested for successful execution). We may explore other versions to ensure the compatibility of our codes with them in the future. Please refer to the following link for details:

https://metamap.nlm.nih.gov/MetaMapLite.shtml

Please note that MetaMapLite 2017 3.6.1p1 with Category 0+4+9 (USAbase) 2017AA UMLS dataset is good, but MetaMapLite 2017 3.6.1p1 with Category 0 (Base) 2017AA UMLS dataset is NOT good since some needed database is not available in the 'base' version.

One more thing about MetaMapLite is that we need to specify its installation location either in the "MetaMapLiteAPI.Properties" file at the root of the development environment, or passing the directory as a String parameter to the "MetaMap" class constructor. Again, please see README.txt for details.

2. To initialize the wrapping class.
   (a) MetaMap metaMap = new MetaMap();

The program reads the value from "MetaMapLiteAPI.Properties" without taking any parameter to the constructor as shown above.

(b) *MetaMap metaMap = new MetaMap("C:/software/public_mm_lite/");*

Alternatively, a specific path can be set as the parameter to the constructor.

3. To execute the key function of "processText()" to obtain the MetaMapLite Coding result.

    HashMap<String, ArrayList<Term>> codedResults = metaMap.processText(sampleText, null, null, 0);

The function takes four paraemters:

- **sampleText**: the raw text to process;
- **selectedOntologies**: the ontology a user needs the MetaMapLite ontology to specify; this value is recommended to set as 'null';
- **resources**: the resources of the coding a user needs it to specify; if this variable is empty, it means the specification of all resource types; this value is recommended to set as 'null'.
- **range**: an integer specifying the context of the extracted text spans, this value is recommended to set as '0'.

The function returns the coding results stored in a HashMap data structure, specifically, HashMap<String, ArrayList<Term>>.

    HashMap<String, ArrayList<Term>> codedResults = null;

Specific examples are shown below. Specific values can be obtained from the link here: https://metamap.nlm.nih.gov/MetaMapLite.shtml

4. Result storage
    (a) HashMap<String, ArrayList<Term>> codedResults = metaMap.processText(sampleText, null, null, 0);

The results can be stored in a HashMap structure as specified above.

    (b) TermToXML ttx = new TermToXML(codedResults, sampleText);

    String resultXMLStr = ttx.generateXMLString(codedResults);

As an additional convenience, the results can be stored in a String form that contains the XML content being VAERS Data Type System compliant. In order to get the String result, a utility class of "TermToXML" needs to be instantiated. And the function of "generateXMLString()" can be executed to return the expected XML content in a String.

5. Some execution commands

- mvn clean
- mvn build
- mvn -Dtest=TestMetaMapLite test

1.4.7 MedTARSQI

This project contains the MedTARSQI package and wraps its function with a single API.

In order to make use of this API programmatically, a developer needs to do the followings:

1. To initiate an instance of the class, "CTakesTempora", where inputString contains raw text to process.

   MedTARSQI medTARSQI = new MedTARSQI(inputString);

2. To run the following three APIs to get the String form of the XML content for Clinical, Temporal and Association Relationship respectively.

   String result = medTARSQI.getClinicalXMLContent();
   result = medTARSQI.getTemporalXMLContent();
   result = medTARSQI.getRelationXMLContent();

1.4.8 ClearTKAPI

The initialization and execution of the ClearTK Machine Learning Classifier is wrapped as a class and a simple API, as follows, to provide an easy-to-access interface to software developers and integrators.

Further information regarding ClearTK can be found in this web site:

https://cleartk.github.io/cleartk/

1. To configure the parameters properly.

A couple of key values need to be specified for the program to run successfully in the file named "ClearTKAPI.Properties", including

(a) *MetaMapLiteDirectory=C:/software/public_mm_lite/*

The specification of the MetaMapLite installation directory. Please also see the related project of "metamaplite-coding" project for details.

(b) *ModelDirectory=C:/software/Data/Models/*

The specification of the directory that contains the Machine Learning models. This directory can be any place in the target environment, such as "C:/software/Data/Models/" that has been tested to function correctly.

2. To initialize the wrapping class.
   (a) RunClearTK runClearTK = new RunClearTK();

The RunClearTK wrapper class is intialized.

(b) JCas jCas = runClearTK.runClearTKModel(report, modelName);

The execution of the key API, "runClearTK.runClearTKModel(String report, modelName)". The parameters are:

- String report: the raw text for processing; and
- String modelName: one of the machine learning model names, i.e., "svm" and "crf". SVM stands for Support Vector Machine; CRF stands for Conditional Random Field.

The return of the API is a standard JCas structure as shown above.

1. Botsis, T., et al., *Decision support environment for medical product safety surveillance.* J Biomed Inform, 2016. **64**: p. 354-362.