Technical Report

Temporal Expression identification, Event identification and Relation identification in Clinical Domain

1. Preprocessing

- For preprocessing we have used cTAKES, MedTime and few scipts in java .
- We first process the files using cTAKES to get domain specific knowledge required to adapt the temporal evaluation challenge to clinical domain.
- We process the cTAKES output to extract features like Medlex normalized value, Semantic type (tui), Concept Unique identifier (cui), whether a token was a concept/ medication/ sign/ symptom, etc.
- cTAKES generated output is then parsed to get the relevant information.
- We use MedTime to get the corresponding span and type output which is further used in our Hybrid model as a feature.
- We used SentLex to get the polarity of words.

MedTime

- Default documentation for MedTime can be found in the MedTime directory of the project.

- Download all the jar files needed to run MedTime from the latest version of MedTime available into the lib folder from the following site: https://sourceforge.net/projects/ohnlp/files/MedTime/

- Change current directry to MedTime-1.0.2

```
$ cd MedTime-1.0.2
```

- Run the following java command to open Collection processing engine for MedTime.

```
$ java -Xms512M -Xmx2000M -cp
resources:desc:descsrc:medtaggerdescsrc:MedTime-1.0.2.jar
```

org.apache.uima.tools.cpm.CpmFrame

- After the window open load the CPE descriptor from the File menu.
  File => Open CPE descriptor => Select the File
  MedTime-1.0.2/desc/medtimedesc/collection_processing_engine/MedTimeCPE.xml

- This file configures the MedTime to use the Aggregate processing engine and set parameters like batch size, number of threads, etc. It also sets the default input and the output directories.

- The aggregate processing engine sets the modules that are to be used by the MedTime and sets the paths to be used by MedTime.

- Once the Aggregate processing engine is loaded. Select the input and the output directory turn by turn first for training files and then for testing files.

- This requires the training and the testing files to be present in the input folder and the outputs are generated in the output folder specified. We specify input folders as "*testdata/medtimetest/input/train*" and "*testdata/medtimetest/input/test*". The corresponding output folders are specified as "*testdata/medtimetest/output/train*" and "*testdata/medtimetest/input/test*".

- Now, click on Run button on the bottom centre of the window turn by turn for test and the train data.

- Then, we have the TIMEML annotated date with the temporal expressions and their type which is to be used as feature in our Hybrid model for Temporal Span reasoning.

<u>cTAKES</u>

- The default documentation of cTAKES is available in the apache cTAKES folder in our project
- Download all the jar file needed to run cTAKES from cTAKES website (http://ctakes.apache.org/downloads.cgi) into the lib folder.
- Move the the home directory of the project and change current directry to apache-ctakes-3.2.2

  $ cd apache-ctakes-3.2.2
- UMLS dictionary access in cTAKES requires UMLS Metathesaurus License which can be obtained from (https://uts.nlm.nih.gov/license.html). To process the files using cTAKES, create an account and request for the license.
- Add the UMLS username and password to bin/runctakesCPE_train.sh and bin/runctakesCPE_test.sh by replacing UMLS_USERNAME and UMLS_PASSWORD present at the end of these files in the java command.
- Copy the training and testing files in folders "input/train" and "input/test" respectively.
- Run for training files:

  $./bin/runctakesCPE_train.sh
- Run for testing files:

  $./bin/runctakesCPE_test.sh
- The *runctakesCPE_train.sh* and *runctakesCPE_test.sh* files specify that clinical pipeline of apache cTAKES is to be used and the processing engine is plaintext aggregate processing

engine with UMLS support.

- The output is generated in "output/train" and "output/test".

Parsing cTAKES output

- Move to the home directory and then change current directory to CtakesProcessing.

```
$ cd CtakesProcessing
```

- Copy the cTAKES generated output in the last step to folders "*Ctakesoutput/train*" and "*Ctakesoutput/test*".

```
$ cp  -r ../apache-ctakes-3.2.2/output/train Ctakesoutput/train
$ cp  -r ../apache-ctakes-3.2.2/output/test Ctakesoutput/test
```

- Compile the file CtakesAttributes.java

```
$ javac CtakesAttributes.java
```

- The *CtakesAttributes.java* file does the preprocessing of the cTAKES generated output to extract the cui, tui and normalized values. It checks for each token whether it is a *AnatomicalSiteMention, DiseaseDisorderMention, LabMention, MedicationEventMention, MedicationMention, ProcedureMention, SignSymptomMention.* Then, it checks whether these extracted elements have the attribute *_ref_ontologyConceptArr* which contains the information for cui and tui for a token.

- Execute the program in CtakesAttributes

```
$ java CtakesAttributes
```

- This will generate parsed outputs in "*ctakesProcessed/train*" and "*ctakesProcessed/train*".

Preprocessing the gold annotated and the raw data to get the tags required for training and testing

- Move to the home directory of the project and change the current directory to Python-CRF-SVM.

  $ cd Python-CRF-SVM

- Create folders and copy the necessary files
  - '***ctakesProcessed***': cTAKES processed output already generated earlier.
  - '***gold_annotated***': the annotated train and test data in corresponding train and test folders.
  - '***MedTime-output***': the output generated from MedTime that we have already obtained.
  - '***raw_data***': the raw text files in subdirectories train and test respectively.

- Create empty directories each with subfolders train and test.
  - '***MedTimeTemp-output***': Used for converting the MedTime output from TimeML format to Anafora format for preprocessing.
  - '***DocTimedumpPickle***s': for dump files required for DocTime relation identification.
  - '***dumpPickles***': for preprocessed data for Event Span detection.
  - '***SVMdumpPickles***': Used for preprocessed data for SVM Event attributes detection.
  - '***System-output***': Used to generate the final processed Anafora format files with the identified events, temporal expressions and relations.
  - '***TimedumpPickles***': for preprocessed data used by Time attributes detection.

    - '***TimeSpandumpPickles***': for preprocessed data used by Time Span detection.
- Run the preproceesing files each to get the preprocessed information in the above created folders first with for test files with no argument passed and then with train passed as an argument to process training files.

  For the test file:

  ```
  $ python dumpTuples.py
  $ python dumpTuplesSVM.py
  $ python dumpTuplesTime.py
  $ python dumpTuplesDoctime.py
  $ python dumpTuplesTimeSpan.py
  ```

  For the training files:

  ```
  $ python dumpTuples.py train
  $ python dumpTuplesSVM.py train
  $ python dumpTuplesTime.py train
  $ python dumpTuplesDoctime.py train
  $ python dumpTuplesTimeSpan.py train
  ```

- This completes the preprocessing part of our project.
- Every dumpTulples file uses a corresponding tokenizer file used to get the tuples of preprocessed data from each file as per the requirement of the task. For example, if we have **dumpTupleDocTime.py** then we have a corresponding **tokenizerDocTime.py** that is used to get the information requires for the task in form of a tuple for each token. The token may be of the form as shown in an example below: *(word, pos[1], norm, cui, tui, label, word_offset, running_offset-1, fileName, Type, Degree, Polarity, Modality, Aspect)*
- *Thus we have the following files for preprocessing:*

- *dumpTuples.py – for Event Span*
- *dumpTuplesSVM.py – for event attribute*
- *dumpTuplesTime.py – for time attribute*
- *dumpTuplesDoctime.py – for doctime relation*
- *dumpTuplesTimeSpan.py – for time span*

- *tokenizer.py – for Event Span*
- *tokenizerSVM.py – for event attribute*
- *tokenizerTime.py – for time attribute*
- *tokenizerDoctime.py – for doctime relation*
- *tokenizerTimeSpan.py – for time span*

Running models for classification

Timex Span Identification:

CRF for Timex Span detection

```
$ python trainCRF-TimexSpan.py
$ python testCRF-TimexSpan.py
```

SVM for Timex Span detection

```
$ python trainSVM-TimexSpan.py
$ python testSVM-TimexSpan.py
```

Timex Attribute Classification:

The Timex Span detection using CRF is used in attribute classification. So, run TimexSpan identification using CRF first and then attribute detection.

Type:

```
$ python trainSVM-TimexType.py
$ python testSVM-TimexType.py
```

Event Span Identification:

CRF for Event Span detection

```
    $ python trainCRF-EventSpan.py
    $ python testCRF-EventSpan.py
```

SVM for Event Span detection

```
    $ python trainSVM-EventSpan.py
    $ python testSVM-EventSpan.py
```

Event Attribute Classification:

The Event Span detection using CRF is used in attribute classification. So, run EventSpan identification using CRF first and then attribute detection.

Type:

```
    $ python trainSVM-EventType.py
    $ python testSVM-EventType.py
```

Modality:

```
    $ python trainSVM-Modality.py
    $ python testSVM-Modality.py
```

Polarity:

```
    $ python trainSVM-Polarity.py
    $ python testSVM-Polarity.py
```

Degree:

```
    $ python trainSVM-Degree.py
    $ python testSVM-Degree.py
```

Document Relation Identification:

Event Span identification and Timex span identification to be performed before this step.

```
$ python trainCRF-DoctimeRelation.py
$ python testCRF-DoctimeRelation.py
```

Major function present in the above files:

- ***word2features(sent, i)**:*

  gets the features corresponding to a word in a sentence at a particular position i.

  Args:

  sent: the sentence whose word is to be considered
  i: the position of the word in the sentence

  Returns:

  the dictionary containing the features for the classifier

- ***sent2features(sent)**:*

  get feauture vector for the sentence

  Args:

  sent: the sentence correposnding to which feauture vector is to be extracted

  Returns:

  feature vector for a sentence

- ***sent2labels(sent)**:*

  get a vector of labels for the sentence

  Args:

  sent: the sentence correposnding to which label vector is to be extracted

Returns:

a vector of labels for the sentence

- ***sent2tokens(sent)**:*

  get a vector of tokens for the sentence

  Args:

  sent: the sentence correposnding to which tokens vector is to be extracted

  Returns:

  a vector of tokens for the sentence

- ***eventEvaluate(cor,pred)**:(present in test model files)*

  Evaluates using partial matching

  Args:

  cor: list of the correct label
  pred: list of the predicted label

- ***exactEvaluate(cor,pred)**:(present in test model files)*

  Evaluates using partial matching

  Args:

  cor: list of the correct label
  pred: list of the predicted label

- ***load(trainOrTest)**: (present in loadTuples.py)*

  loads the date dumped in pickle files for each file into a vector

  Args:

  trainOrTest: whether it is training data or testing

Returns:

the vector containing data in dumped pickle for each file

- ***getSpans(filename)****: (present in parseGold.py and similar files)*

  parses the anafora files to get the properties for entities from gold annotated data

  Args:

  filename: the file to be parsed

  Returns:

  the spans of all the tokens

- ***getMedSpans(filename,trainOrtest="test")****: (present in parseMedTime.py)*

  parses the anafora files to get the properties for entities from MedTime processed data

  Args:

  filename: the file to be parsed

  trainOrtest: whether its training data or test data

  Returns:

  the properties of all the temporal expressions

- ***getIsSpell(word)****: (used for temporal expressions)*

  Checks whether the word is a spelling of common numbers

  Args:

  word: the word to be checked for spelling of num

Returns:

True if the word is spelling of common number

- ***getIsQuant(word)**: (used for temporal expressions)*

  Checks whether the word is common quantitative descriptor

  Args:

  word: the word to be checkd for quantitative descriptor

  Returns:

  True if the word is spelling of common quantitative descriptor

- ***getIsPrePost(word)**: (used for temporal expressions)*

  Checks whether the word is common pre-post expression

  Args:

  word: the word to be checkd for common pre-post expression

  Returns:

  True if the word is spelling of common pre-post expression

- ***getNum(label)**: (used in case SVM is the model)*

  get a unique number corresponding to each label

  Args:

  label: the label correposnding to which a number is to be alloted

  Returns:

  a unique number corresponding to each label