NILE User Manual

Java Version

Welcome

NILE (Narrative Information Linear Extraction) is an efficient and effective software for natural language processing (NLP) of clinical narrative texts, developed by Sheng Yu, Tianxi Cai, and Tianrun Cai. It is distributed free of charge by the President and Fellows of Harvard College for academic and non-commercial research use. NILE has a Java version and a Windows executable version. The Java version is intended for data scientist who are familiar with Java programming, operated via simple APIs for maximum flexibility. The Windows executable version is intended for non-Java programmers; it is converted from the Java version and is configured using a property file. This manual is for the Java version, for the introduction of its APIs.

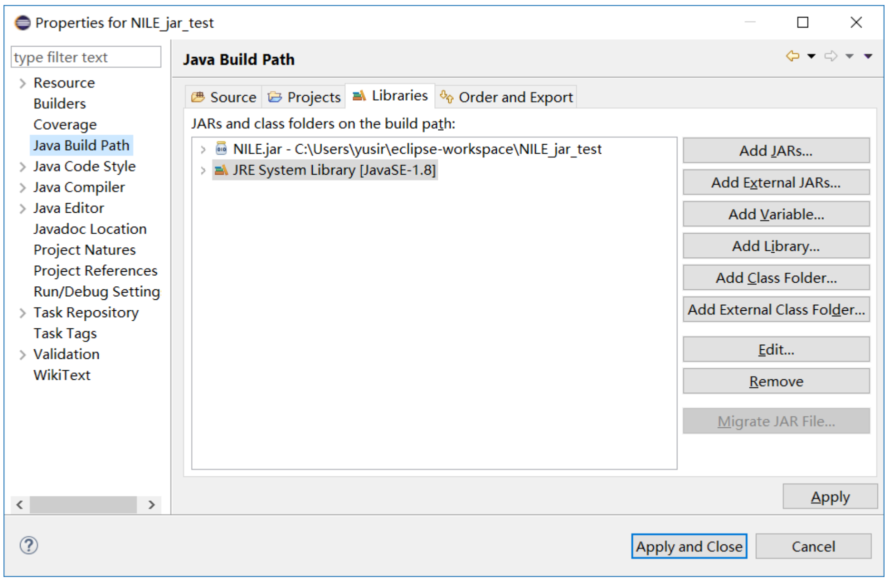
System requirements

The NILE Java library was developed and tested under Java-SE 8. We encourage users to use NILE under Java 8, too, though newer Java versions will likely work as well.

There is not a minimum requirement for the RAM. Memory consumption is determined by the dictionary size. When using a domain specific dictionary, like the accompanied example, the memory footprint is negligible. When using a dictionary with millions of terms, NILE will consume gigabytes of memory, and the user will need to allocate heap space accordingly.

Setup NILE

The NILE Java version comes as a JAR package. You may put the JAR under your project folder (or anywhere you wish), and add the NILE library as an external JAR. Then you can import and use NILE's classes in your code.



JARs and class folders on the build path:

NILE.jar - C:\Users\yusir\eclipse-workspace\NILE_jar_test

JRE System Library [JavaSE-1.8]

Common steps for using NILE

The accompanied file Example.java is a working example for using NILE. Using NILE for electronic health record NLP typically involves the following steps:

1. Create the NILE NLP environment;
2. Load the dictionary;
3. Parse the notes;
4. Record the results.

Create the NILE NLP environment

The NILE NLP environment is created using the class constructor:

```
    NaturalLanguageProcessor nlp = new NaturalLanguageProcessor();
```

It is necessary to enclose it within a try-catch block because the constructor may throw InconsistentPhraseDefinitionException. This exception is thrown when there is an attempt to define a term with two distinct semantic roles. The environment constructor will load the built-in dictionary for grammatical words and cues, thus exception handling is mandatory. However, in practice, the released version of NILE will not trigger such exceptions at this step.

Load the dictionary

There are two ways to load the dictionary. One way is the batch mode, using the addVocabulary() method, such as:

```
nlp.addVocabulary("dict_obs.txt", SemanticRole.OBSERVATION);
```

The first argument is the path of a dictionary file, in which each line contains a term and a code, delimited by '|', such as "type 2 diabetes|C0011860". The second argument means to assign this semantic role (here, Observation) to all the terms in the specified file. Normally, users should only add terms of the roles Observation, Location, and Modifier. Other roles are for the semantic analyzers, and do not show up in the result. The terms and codes are case sensitive. Terms should be lower-cased, because NILE will lower-case the sentences before parsing.

Two exceptions may be thrown and must be handled: the FileNotFoundException and the InconsistentPhraseDefinitionException. The latter exception is usually encountered when the user tries to load terms from vocabularies such as the UMLS, because massively exported terms usually contain conflicts with the built-in grammatical words and cues. Unfortunately, when a conflict happens, the entire loading will abort. To skip only those conflicts, it is necessary to load the dictionary file line by line (term by term), using the second method:

```
nlp.addPhrase("type 2 diabetes", "C0011860", SemanticRole.OBSERVATION);
```

where the first argument is the term string, the second argument is the code, and the third is the semantic role. In this way, the InconsistentPhraseDefinitionException will only affect a single term instead of the entire file.

NILE allows a term to have multiple codes – new codes to a term will be appended to the code list. Adding the same (*term, code, role*) triple again is fine, and the code won't be duplicated.

Parse the notes

After creating the environment and loading the dictionary, one typically connects to a database or a file folder to access the notes, and note parsing and result recording typically happen in a while loop. For users' convenience, note parsing in NILE is done by paragraphs, as each paragraph is usually a text line from the note that contain multiple sentences. Suppose String text is such a line. Then nlp.digTextLine(text) returns a List<Sentence>, and each Sentence contains a List<SemanticObject>, accessed via getSemanticObjs(). Semantic objects are recognized named entities with additional properties from semantic analyses. See the next section for accessing its contents.

Record the results

An advantage of using NILE as a Java library is that it gives the user the freedom to design what to output and its format. The Sentence and SemanticObject objects come with their toString() and toShortString() methods, but typically only part of the information, such as the code and the presence status, are relevant to the user's application. So, in most cases, the user may want to design what to output using SemanticObject's APIs. For a SemanticObject, that is, a recognized named entity, one can access its text via getText() (String), its codes via getCode() (List<String>), its semantic role via getSemanticRole() (enum), its presence via getCertainty() (the negation analysis, enum), and whether the mention is a family history via isFamilyHistory() (boolean). If your dictionary contains Location or Modifier terms, then a SemanticObject may also contain modifiers, accessed via getModifiers() (List<SemanticObject>). And since each modifier is also a SemanticObject, it also has the above APIs, but typically getCertainty() and isFamilyHistory() would not make sense for modifiers. Also note that location modifiers can be nested, which is a feature of NILE. See Example.java for how to print or access nested modifiers.

Finally, note that NILE analyzes whether a mention of an Observation object actually indicates its presence, and removes all mentions that do not. For example, "patient came to rule out rheumatoid arthritis" and "Breast Cancer Center" do not express presence or absence of "rheumatoid arthritis" or "breast cancer", so these Observation objects will be removed, and the user will not see them in getSemanticObjs(). This is a feature we find helpful in analyzing clinical notes, but keep in mind of this if you want to do a recall test for NILE.