Erik Andersen
Capstone Project
19 Aug 2020

A LAPPSGrid WebService Wrapper for DKPro Core™ Morphological Analyzers

I. Purpose

In this assignment, I wrapped three DKPro Core™ morphological analyzers as web services in the LAPPSGrid using Maven. This allows for a standardized interpretation of each analyzer within the greater context of the LAPPSGrid NLP platform.

II. Morphological Analyzers: A Description

The DKPro Core software collection currently contains support for three different morphological analyzers: The Mate Tools Morphological Analyzer (MateMorphTagger), the RF Tagger Morphological Analyzer (RFTagger), and the SFST Morphological Analyzer (SFSTAnnotator). A fourth morphological analyzer, the COGROO Morphological Analyzer, was available in earlier releases of DKPro but is no longer part of the software. Thus, there are no current Maven dependencies through DKPro that include COGROO. For this reason, I could find no way to directly wrap the COGROO Morphological Analyzer as a web service through DKPro, and as a result, I did not attempt to include this analyzer. Still, the related Cogroo Featurizer exists as a separate entity, and could conceivably be wrapped as a web service within the LAPPSGrid.

III. Methodology

Using the existing LAPPSGrid tutorial for wrapping a web service, available starting at https://github.com/lapps/org.lappsgrid.examples/tree/step1, I built up a template for a web service wrapper within the LAPPSGrid.

Understanding the structure of the morphological analyzers themselves, however, proved more difficult. Each analyzer contains two basic methods, *initialize* and *process*. The former method takes an UIMAContext variable, which initializes the given context and finds the model to be used. The latter method, *process*, takes a JCas variable and runs the morphological analyzer, filling the JCas variable with the resulting annotations. These are written as UIMA Annotators, and it is important that all variables are initialized correctly, which can be difficult to do by using *initialize* and *process* by themselves. Looking at the test files for each of the morphological analyzers, the *runTest* method from *de.tudarmstadt.ukp.dkpro.core.testing.TestRunner* proved to be the best way to implement the *process* method without any setbacks. This only left creating an AnalysisEngine as an initialization method to run over the desired model pipeline. While I initially assumed this pipeline only required one model (the analyzer itself), I discovered that the MateMorphTagger would not run as expected without first calling the MateTools lemmatizer to provide lemmas for each word in the sentence. This process is shown in the MateMorphTagger test file and repeated in my implementation.

Another issue involved the exact format of the input string that would be fed to the *execute* method of the LAPPSGrid wrapper. The *execute* method is an overridden method, consisting only of one parameter, the String *input*, and thus it prohibits the inclusion of other parameters. It seems most logical to run the analyzer within the *execute* method, as the analyzer must be run each time a new sentence is introduced. However, the analyzer must be fed the language (and in some cases, the model variant) in addition to the input string. For this reason, the language and input string were fused together using the string sequence "; ". The RFTagger and SFSTAnnotator analyzers both introduce the concept of model variants; these are concatenated between the language and the input string, using the same string sequence. Sample input strings for each analyzer are as follows. Note that even though the variants for all existing RFTagger models are *null*, this option allows the basic structure of the wrapper to still work if new model variants are introduced at a later time.

MateMorphTagger	"de; Wir brauchen ein sehr kompliziertes Beispiel , welches möglichst viele Konstituenten und Dependenzen beinhaltet ."
RFTagger	"de; null; Er nahm meine Fackel und schlug sie dem Bär ins Gesicht ."
SFSTAnnotator	"de; zmorge-orig-ca; Der Arzt arbeitet im Krankenhaus ."

Since the ultimate goal of this project is a web service, it might be beneficial to write an HTML interface that masks the string concatenation so that users do not need to do this themselves. A potential solution might be to create a dropdown menu for languages, a dependent dropdown menu for variants (only in the RFTagger and SFSTAnnotator), and a textbox to write the input string.

Finally, it was important to choose the correct Maven dependencies in order to make this project work. For example, due to some UIMA compatibility issues with DKPro 2.0, all DKPro-related dependencies needed to be from version 1, or else inescapable Java errors would be tripped.

IV. MateMorphTagger

The first wrapper was built around the MateMorphTagger. Looking at the tests provided, there are three language-models available, one each for German, French, and Spanish. As a basis for the wrapper, I wrote a loop over all the Token annotations (the broadest annotation class present) and extracted the information I could from it, focusing on the word, lemma, part of speech (POS), and morphological information. The lemmas are easily accessible because of this analyzer's requirement of running the MateTools Lemmatizer as part of the pipeline for the MateMorphTagger. Similarly, the word can be delineated by the start and end offsets found in each annotation.

One subfield of the Token annotation is represented as an annotation of the class MorphologicalFeatures, accessible through the method *getMorph()*. While the MorphologicalFeatures represents a sub-annotation of the broader Token, it itself has many subfields. However, only the "value" is filled out by the analyzer in any of the tests. Since the value contains many different features of form *key=value* separated by a "|", I split on "|" in cases with more than one feature and indexed all available features within the annotation.

Unfortunately, the POS tag for each annotation remains null when just using a pipeline with the Lemmatizer and MateMorphTagger, as the MateMorphTagger does not record the POS tag in its default form. There is also no way to grab the POS tag value from the MorphologicalFeatures value (though with some added rules there might be some way to "guess" the tag). However, the MatePosTagger is a viable system which includes models corresponding to the three languages available for the MateMorphTagger.

For this reason, it was possible to add the MatePosTagger to the AnalysisEngine pipeline, introducing POS tags to our web service.

Upon checking over the tests again, it is apparent that the Spanish model trips an EOFException Error. This might be a specific bug, because I did not find this error on either of the other models, nor did this error carry over to the other two wrappers.

In all, 4 tests exist for this tagger; one for each language, and one for the metadata based on the LAPPSGrid wrapping tutorial.

V. RFTagger

Next, I created a wrapper for the RFTagger. Upon consulting the available tests and the DKPro website, it is evident that 6 languages exist for this analyzer, each using 1 model. The languages in question are Czech, German, Hungarian, Russian, Slovak, and Slovene.

For this tagger, the wrapping process was similar to that of the MateMorphTagger, except that the RFTagger permits the MorphologicalFeatures part of the Token annotation to have more non-null subfields, such as gender and number. To access these, I wrote a method to grab the value of any given field if it exists, or return *null* if no such value is present. Though non-null subfields mainly appear only in the German model, this is somewhat fortunate, because the MorphologicalFeatures tags (found in the *value*) are less easily divisible than their MateMorphTagger counterparts. For this reason, no splitting is conducted on these tags. Instead, I simply store the *value* in the annotations for our web service. As a disadvantage, the RFTagger does not provide lemmas, and it does not appear to have a sister lemmatizer like the MateMorphTagger, so I leave the lemma field blank.

However, this model is a POS tagger at heart, so POS tags are returned by the analyzer and are thus not difficult to access.

In all, 7 tests exist for this tagger; one for each language, and one for the metadata based on the LAPPSGrid wrapping tutorial.

VI. SFSTAnnotator

Finally, I created a LAPPSGrid wrapper for the SFSTAnnotator. The available languages for this analyzer are German, Italian, and Turkish, but the German language consists of 4 different models, whereas the Italian and Turkish each have 1 model.

This analyzer differs somewhat from the other two analyzers. Not only are legitimate variant names introduced, but multiple MorphologicalFeatures annotations also exist for each token. As the analyzer basically functions as a finite state transducer, each model produces all possible finite state sequences instead of a best path. In this case, it is no longer sufficient to loop through the annotations of the Token class, because doing so will only provide access to the top finite state sequence for each token. To access all possible sequences, looping through all annotations of class MorphologicalFeatures is required.

For each of the MorphologicalFeatures annotations present, there remains the possibility that subfields (such as gender or number) might be filled, as was the case for the RFTagger above. Thus, a similar selection process is used to only gather non-null values. To each unique MorphologicalFeatures annotation, a special token id is assigned, derived from the position of the token within the sentence and the ordinal number of the finite state sequence for that token. Thus, the sixth finite state sequence for the third token would be defined as *tok2_5*.

Sadly, the SFSTAnnotator provides neither lemmas nor POS tags. However, these are hidden within the MorphologicalFeatures annotation tags. Each of these tags represents the finite state sequence for that unique morphological interpretation. As these consist of morpheme combinations and POS tags, it is possible to derive a lemma and an appropriate POS tag from each tag sequence. To capture the lemma, all the morphemes (with any intermediate tags included) are strung together. The tag directly following the characters of the last morpheme is then designated as the POS tag. Two examples of this process are detailed below. Both of these examples come from the *zmorge-newlemma-ca* model.

MorphologicalFeatures Tag	Resulting Lemma	Resulting POS Tag
Kran<#>ken<#>haus<+NN><Neut><Nom><Sg>	Kran<#>ken<#>haus	<+NN>
<CAP>die<+REL><Subst><Fem><Dat><Sg><St>	<CAP>die	<+REL>

As with the RFTagger, 7 tests exist for the SFSTAnnotator LAPPSGrid wrapper. To be more specific, there is one test for each language, and one for the metadata as well.

VII. Future Work

In the future, I would like to expand across other morphological analyzers. One potential way to begin would be to write a wrapper for the COGROO Featurizer, whose Maven dependencies appear to be defunct within the DKPro system at this time. As stated above, it is theoretically possible to create a wrapper for the existing COGROO Featurizer. However, doing so will require venturing outside of the realm of DKPro.

In addition, it might be beneficial to work further on the interactivity of the web service itself. As described above, the addition of an HTML interface to mask the unwieldy format of the current input string could greatly increase user-friendliness.

VIII. Conclusion

This project represents an important step in the standardization of a wide variety of NLP tools. Though these three analyzers all exist within the same software and are all structured as UIMA annotators, they all have very different functionality and available information. For this reason, no single wrapper that I built was as easy to adapt to another as I had initially expected. As more and more NLP tools become available on the internet, it will become increasingly important for them to interact with one another.