Tagger Annotator Documentation

Written and maintained by the Apache UIMA Development Community

Version 2.3.1

Publication date August, 2011

Table of Contents

Introduction

Tagger Annotator is an Apache UIMA statistical analysis engine that annotates tokens with corresponding grammatical types (parts of speech, or just POS). The tagger is a standard hidden Markov model (HMM) tagger.

Chapter 1. Prerequisites

The UIMA HMM Tagger annotator assumes that sentences and tokens have already been annotated in the CAS with Sentence and Token annotations respectively (see e.g. `Whitespace Tokenizer Annotator`). Further, the tagger requires a parameter file which specifies a number of necessary parameters for tagging procedure (see Section 3.1, “Configuration Parameters” [5]). Two trained models for English and German are included in the package (in the `resources` folder). Other models can be trained outside of the UIMA framework (see Chapter 6, *Training Own Models* [11]).

Chapter 2. Processing Overview

The algorithm iterates over sentences and tokens in turn to accumulate a list of words. These are then sent to a processing engine of HMM tagger. For each `Token` , the `posTag` field is updated with the corresponding part of speech (e.g. `posTag = "NN"` where `NN` stands for *common noun*).

Chapter 3. Annotator Descriptor

Two descriptors are employed to configure tagger's functionality:

- `HmmTagger.xml` - is a primitive analysis engine descriptor, which defines the tagger basic functionality and can be combined in an aggregate analysis engine with an arbitrary tokenizer. This descriptor cannot be used on itself as the tagger alone does not perfom tokenization.
- `HmmTaggerTAE.xml` - is an aggregate analysis engine whose only function is to combine UIMA `Whitespace Tokenizer Annotator` with `HMM Tagger Annotator` and is thereby a "ready to use" tagging descriptor.

3.1. Configuration Parameters

The HMM tagger annotator (`HmmTagger.xml`) requires the following configuration parameters:

- `NGRAM_SIZE` - this parameter is an Integer, defining whether a bi- or trigram model should be used for tagging (default is N=3).

```
<configurationParameters>
  <configurationParameter>
    <name>NGRAM_SIZE</name>
    <type>Integer</type>
    <multiValued>false</multiValued>
    <mandatory>true</mandatory>
  </configurationParameter>
</configurationParameters>
<configurationParameterSettings>
  <nameValuePair>
    <name>NGRAM_SIZE</name>
    <value>
      <integer>3</integer>
    </value>
  </nameValuePair>
</configurationParameterSettings>
```

- `ModelFile` - binary file containing the statistical model which should be used for tagging is defined as an external resource

```
<externalResources>
  <externalResource>
    <name>ModelFile</name>
    <description>HMM Tagger model file</description>
    <fileResourceSpecifier>
      <fileUrl>file:german/TuebaModel.dat</fileUrl>
    </fileResourceSpecifier>
    <implementationName>
    org.apache.uima.examples.tagger.ModelResource
    </implementationName>
  </externalResource>
</externalResources>
```

Thus, one can easily use a different model by changing the `fileUrl` line: `file:german/TuebaModel.dat`. (NB. *New models must be located in the `resources` folder*.) After these two parameters have been set, the tagger is ready to use.

3.2. Capabilities

As the tagger inherits tokenization indexes from the CAS, `uima.SentenceAnnotation` and `uima.TokenAnnotation` with their `begin` and `end` features respectively have to be defined as input capabilities in the HMM Tagger annotator descriptor. `Token` receives also an additional `posTag` feature as an output capability.

```
<capabilities>
      <capability>
        <inputs>
          <type>org.apache.uima.TokenAnnotation</type>
          <type allAnnotatorFeatures="true">
          org.apache.uima.SentenceAnnotation
          </type>
          <feature>org.apache.uima.TokenAnnotation:end</feature>
          <feature>org.apache.uima.TokenAnnotation:begin</feature>
        </inputs>
        <outputs>
          <type>org.apache.uima.TokenAnnotation</type>
          <feature>org.apache.uima.TokenAnnotation:posTag</feature>
          <feature>org.apache.uima.TokenAnnotation:end</feature>
          <feature>org.apache.uima.TokenAnnotation:begin</feature>
        </outputs>
      </capability>
    </capabilities>
```

Chapter 4. Functionality Test

The `TaggerTest` is a JUnit test file (available in the `test` folder), which provides an opportunity to test provided models for English and German, as well as the basic functionality of the tagger. In order to check whether the tagger's configuration is correct, just run this file as JUnit Test and you should get the following output:

```
Tesing German Model...
The used model is:resources/german/TuebaModel.dat
61646 distinct words in the model
Number of part-of-speech tags used: 54
These are:  [$(, $,, $., ADJA, ADJD, ADV, APPO,
 APPR, APPRART, APZR, ART, CARD, ... ]
Testing German trigram tagger..
[Jerry, liebt, Wansley, .]
expected: [NE, VVFIN, NE, $.]
tagger output: [NE, VVFIN, NE, $.]
Very Good!
============================================================
Tesing English Model...
The used model is:resources/english/BrownModel.dat
56012 distinct words in the model
Number of part-of-speech tags used: 473
These are:  ['', '', (, ), *, ,, --, ., :, ``, abl,
 abn, abx, ap, ap$, at, be, bed, ...]
Testing English trigram tagger...
[Jerry, loves, Wansley, .]
expected: [np, vbz, np, .]
tagger output: [np, vbz, np, .]
Very Good!
```

Chapter 5. Overview of the Tagger package

The package `org.apache.uima.examples.tagger` contains:

- two interfaces:
    1. `IModelResource` - model resource interface
    2. `Tagger` - general tagger interface, in case one would want to integrate further tagger types.
- three classes:
    1. `HMMTagger` - hidden Markov model tagger for UIMA, that is using Viterbi algorithm to compute the most probable part-of-speech sequence for a given list of tokens.
    2. `Viterbi` - implementation of the Viterbi Algorithm. This class makes up the core of the tagger.
    3. `ModelResource.java` - implementation of the `IModelResource`

Chapter 6. Training Own Models

Though we decide not to include training directly into UIMA framework, one can easily train other models for different pre-annotated corpora outside of the UIMA using `ModelGeneration` class, available in the subpackage `org.apache.uima.examples.tagger.trainAndTest`. This subpackage includes some further files needed for training of own models:

- `MappingInterface` - defines mapping for a tagset. For example, one may wish to map a more detailed tagset to a less distinctive one (i.e. tell a program to tag all verbs as just `VERB` instead of differentiating between `verb infinitive`, `verb imperative`, etc. Two sample implementations for `MappingInterface` are included, namely `TagMappingBrown` (mapping reducing Brown corpus tagset from more than 400 tags to 93) and `GrobMappingTueba`(mapping German STTS tagset from 54 tags to basic 11 categories plus special symbols and punctuation)

- `ModelGeneration` - trains an N-gram model for the tagger, iterating over a List of `Token`s. Writes the resulting model to a binary file. At the moment, only bi-and trigram models are supported. Further N-grams can be easily integrated. `ModelGeneration` is not concerned with the fact, whether the training corpus is given as a single file or as a directory containing a number of files, as this is a `CORPUS_READER` implementation issue. Two supplied readers include both a reader for a corpus as a single file (`TT_FormatReader`code>) or as a directory (`BrownReader`code>)

- Interface `CorpusReader` - should be used to implement corpus readers for own corpora; the objective of the reader is to take charge of the preprocessing and transform tokenized units (usually *words*) into a List of `Token` objects. Two sample implementations of `CorpusReader` are included:

    1. `BrownReader` - for the Brown corpus from the nltk distribution (nltk.sourceforge.net)

    2. `TT_FormatReader` - for the corpora in TreeTagger format, i.e. one word per line with tags separated from the words by tabs.

To train a new model, one should adjust a number of parameters in the `"tagger.properties"` file, which is in Java properties file format (see tagger.properties file [11]). After the parameters are set, you just need to run `ModelGeneration.java`

```
######## This is the default tagger.properties file
######## This file is used for training and testing only,
######## The configuration for tagging is directly
######## tuned in the descriptor "HmmTagger.xml"


#########################  BOTH FOR TRAINING AND EVALUATION  ########

######## THESE ARE THE DEFAULT MODEL FILES FOR GERMAN AND ENGLISH
######## You can either uncomment one of them, if you want to replace
######## given models with your own one,

#MODEL_FILE = resources/german/TuebaModel.dat
#MODEL_FILE = resources/english/BrownModel.dat

######## or specify a completely different name
MODEL_FILE =
```

```
######## If mapping of tags is desired, uncomment the following
#DO_MAPPING = true


####### EXAMPLES OF MAPPING CLASSES

## Basic mapping for the Brown corpus (nltk distribution) tagset:
## to get 93 tags out of 473
#MAPPING = org.apache.uima.examples.tagger.TagMappingBrown
## Basic mapping for STTS tagset: from 54 tags onto the basic
## ca. 15 classes plus punctuation
#MAPPING = org.apache.uima.examples.tagger.GrobMappingTueba

## If you implement your own mapping, you should specify here in
## the same manner as above a java-path to the class
MAPPING =

####### FILE CONTAINING TRAINING CORPUS:
####### can be in specified either as an absolute or as a relative path
####### e.g. FILE = ../../tueba_tigerFormat.txt or FILE = C:/Data/tueba.txt
FILE =

######## If corpus is in a different format and
######## cannot be read with the provided READERS,
######## you should specify here a java-path to the
######## class (s. examples below)

#CORPUS_READER=org.apache.uima.examples.tagger.trainAndTest.TT_FormatReader
#CORPUS_READER=org.apache.uima.examples.tagger.trainAndTest.BrownReader
CORPUS_READER =

##################      ONLY FOR EVALUATION   ######################

########## GOLD STANDARD CORPUS FILE:
########## can be specified as an absolute or as a relative path
## e.g. GOLD_STANDARD = ../../tueba_tigerFormat.txt or
## GOLD_STANDARD = C:/Data/tueba.txt
GOLD_STANDARD =

########## Here we specify whether one intends to test a bi- or a
########## trigram model (default is a trigram model)
N=3
```

Chapter 7. Evaluation

To evaluate performance if a "gold standard" corpus is available, one can use the following provided file:

- `TaggerEvaluation.java` - can be used to evaluate the tagger and/or new models on a manually annotated corpus.

`HMMTagger` was evaluated for English and German. For English, it was trained on 80% of the Brown corpus (180,000 tokens) and tested on the rest unseen 20%. The achieved accuracy was about 96%, test corpus contained 4.5% of unknown tokens.

For German, it achieves between 95% and 96% accuracy when trained and tested on the same type of corpus, i.e. with 80% of corpus used for training and 20% for testing. The accuracy goes a bit down when tagging is performed for different types of corpora than the training one, mostly due to the growing number of unknown words.

Appendix A. Theory Behind

This chapter is just a sketch of the statistical model undelying the tagger. Hidden Markov Models (HMMs) are the mainstay of the applications employing statistical modeling in any form, like speech recognition and production systems, signal processing, part of speech tagging. A Hidden Markov Model is a probabilistic function of a Markov process. A Markov process is a process that fulfills Markov assumptions. Markov assumptions are:

- `limited horizon` - Markov processes are states without memory, except for condition of the current state. Though we usually consider sequences of variables that are not independent of each other, it often suffices to know the value of the current situation without going deep into the past happenings. As [ManningSchuetze99] put it, we do not really need to know, how many books were in the library last week or last year in order to predict how many books there will be tomorrow. It is often enough to know the current situation. Thereby, future states in the Markov process are independent of the past, they only depend on the present. Let $X = (X_1, ..., X_T)$ be a sequence of random variables taking the values from the finite state space $S = (s_1, ..., s_N)$, then a limited horizon property could be formalized by:

  $P(X_{t+1} = s_k | X_1, ..., X_t) = P(X_{t+1} = s_k | X_t)$

- `time invariance`

  The probabilities do not change over time, i.e. if we know that the probability of observing a rainbow after the rain is equal to 90\%, we know that it should be true for today as well as for tomorrow.

If `X` conforms to these two properties, then it is said to be a Markov chain. One can describe a Markov chain by a transition matrix:

$A = a_{i,j} = P(X_{t+1} = s_j | X_t = s_i)$

- with $a_{i,j} >= 0$ (for all *i,j*) and the sum of all transition probabilities from state *i* ($a_{i,j}$) should be equal to 1 (for all *i*)

Markov models can be used whenever one needs to model the probability of a linear sequence of variables. One distinguishes Visible Markov Models (VMM) vs. Hidden Markov Models. The difference is that when we work with "visible" events, we can directly estimate the corresponding probabilities (which is the case if training corpus is available to train own models for HMM taggers). Finding a sequence of part of speech tags (i.e. Viterbi part of the tagger) in contrast is a hidden Markov model as the states (tags) are not directly observable.

The goal of HMM - based tagger is to find part of speech tags (= hidden states) that generate a sequence of words (= observable states). Most of the known implementations of POS taggers are viewing text as being produced by a hidden Markov model, so that tagging can be regarded as a Markov process deciding which states the system went through to generate a given text.

General Form of HMM

A HMM is a five-tuple: (S, O, #, A, B)

where:

- `S` - the set of states (here: parts of speech)

- K - the set of observations (here: words)
- # - initial state probabilities
- A - state transitions probabilities
- B - symbol emissions probabilities

Further, X_t (state sequence) and O_t (output sequence) are given. Tagging procedure is then the following:

1. t := 1
2. Start in state s_i with probability $\#_i$ (i.e., $X_1 = i$)
3. forever do:
    - Move from s_i to s_j with probability $a_{i,j}$ (i.e. $X_{t+1} = j$)
    - Emit observation symbol $o_t = k$ with probability $b_{i,j,k}$
    - t := T+1
4. end

Despite their limitations, HMM-s are one of the most successful techniques in natural language processing and are widely used, especially in sequence tagging applications. The best statistical taggers all perform at about the same level of accuracy.

Glossary

HMM

Hidden Markov
Model

POS

Part of Speech

Bibliography

[ManningSchuetze99] Christopher Manning and Hinrich Schuetze. *Foundations of Statistical Natural Language Processing* . Copyright © 1999. MIT Press.