11-791 Design and Engineering of Intelligent
Information System &
11-693 Software Methods for Biotechnology
Homework 2
Logical Architecture and UIMA Analysis Engines Design
& Implementation

Report

Hao Zhang
haoz1@andrew.cmu.edu

1. System design

1.1 General idea

Our is task is to annotate bio-name entity in a given sentence. The basic id ea is to design multiple analysis engines to implement several different an notation methods. The final result is the combining of results from each se parate analysis engine. Analysis engines can be complementary to each ot her so that one analysis engine can capture the information which other an alysis engine can not.

1.2 Type design

My system pipeline basically calls external package to generate bio-name entity annotation. So I did not use text features such as N-gram. I use edu. cmu.deiis.types.Annotation to store information from different analysis eng ine. The casProcessorid feature is used to denote which analysis engine p roduce this annotatoin. The SentenceID feature is used to record the sour ce sentence ID. NameEntity feature is for storing the recognized name ent ity and finally the confidence feature is to store the score for this entry.

1.3 CollectionReader

The CollectionReader design remains the same as in homework 1. It simply reads in a line from input file sampie.in and initialize it as a CAS for later analysis.

1.4 Aggregated Analysis Engine

There are three principle analysis engine in the pipeline.

1.4.1 The Hmm based Lingpipe bio-name entity recognition.

Lingpipe offered three statistical modelling based name entity recognition Classes. On sample input file, the three statistical learning based chunker yields the following performance shown in Table below. On our task the Hmm based HmmChunker class yields the best precision and recall performance. So I choose to incorporate Hmm based model in the aggregate analysis engine.

	Precision	Recall	F_score
HmmChunker	0.7097	0.7839	0.745
TokenShapeChunker	0.1615	0.506	0.2448
CharLmRescoreChunker	0.1500	0.236	0.1836

1.4.2 Rule based exact matching name entity recognition.

Lingpipe offers two other NER API for dictionary based matching. One is exact matching and the other is approximate matching with a measure of edit distance. The HMM model is not trained on our given data so overfitting does not occur here. I hope the dictionary based name entity recognition can capture the information which HMM model cannot. After I get the output from HMM based analysis engine. I compared it with gold truth and see what kind of name entity it can not recognize (without considering the context of name entity). Then I put those name entity as hard-core entry in the dictionary and do exact matching, merge the result from dictionary based analy

sis engine to HMM based analysis engine output. I also implement the app roximate matching analysis engine, but I find it is too slow and the approxi mate matching generate too many false alarms and therefore the precision is greatly reduced. So the approximate dictionary matching analysis engin e is not put into aggregated analysis engine.

With the help of dictionary based exact matching, the system performance could be boosted. See the Table below:

	Precision	Recall	F_score
HmmChunker	0.7097	0.7839	0.745
HmmChunker+E xact Dictionary match	0.704	0.8870	0.785

1.4.3 CRF based sequential labelling analysis engine for name enti ty recognition.

I trained the a stanford name entity recognizer for our task using the sampl e.in file as training data. The underlying model of stanford name entity reco gnition toolkit is a linear chain conditional random field. Unlike generative m odel like HMM, CRF is purely discriminative model. It models the probabilit y of the label of a word directly without worrying about the probability of the work alone.

In our task, I treat the name entity recognition as a sequential labelling prob lem. I adopt three labels here, namely: BEGIN, INTERMEDIATE, OTHER (B,I,O). "B" stands for the beginning of a name entity and "I" represents th e rest part of name entity followed "B". "O" simply denotes other non-nam e entity words. In terms of feature design in CRF, I did not do too much tu ning here. Basic features include N-gram, word shape, etc. I followed the demo training feature setup online:

http://nlp.stanford.edu/software/crf-faq.shtml

The training around 400 iteration to converge and the model is named MY OWN.ser.gz.

The training error on the CRF model is:

	Precision	Recall	F_score
Trained CRF	0.9057	0.7933	0.8458

The training performance is great but I suspect this might be caused with overfitting. So in the final result combination, I decide to not to include result from CRF model directly, but to enhance the result from HMM based model, which should be more trustable and stable on the unseen testing data.

1.5 Combination strategy (CAS consumer).

In the final combination, we start from result collected from HMM model and adding possible recognition entry from other two models. I check if each entry in other two models has been included in HMM model, if not, we simply add that entry to final result. This could further improve the recall but the precision is hurt a little bit. see the Table below:

	Precision	Recall	F_score
HmmChunker	0.7097	0.7839	0.745
HmmChunker+Exact Dictionary match	0.704	0.8870	0.785
HmmChunker+Exact Dictionary match + CRF	0.6951	0.9111	0.7886

2. Implementation Reference

Collection reader:
CollectionReader.java collectionReaderDescriptor.xml

HMM based analysis engine:
HmmAE.java HmmAE.xml

DIctionary based analysis engine:
DictionaryAE.java DictionaryAE.xml

CRF based analysis engine:
CrfAE.java CrfAE.xml

CAS consumer:
CollectionConsumer.java casConsumerDescriptor.xml

Aggregated Analysis Engine:
hw2-haoz1-aae.xml

Utility:
Generate training data for Stanford NER package:
Dataprepare.java
Score:
score.py (under "data/" dir)