Joseph Chee Chang (josephc1)

11-791 Software Engineering of IIS

September 24, 2014

Homework2

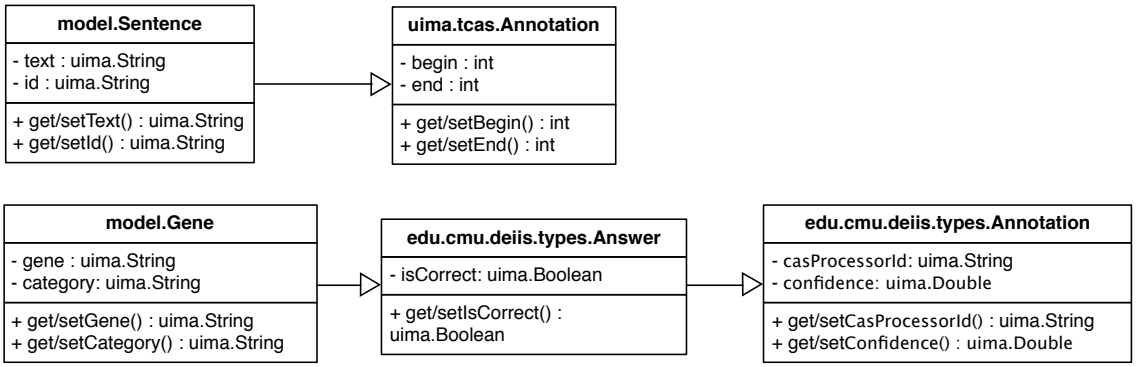
Performance

My implementation consists of four different pre-trained taggers: 1) GeneTag HMM Chunker, 2) GENIA Token Shape Chunker, 3) Stanford POS tagger, and; 4) A CRF model trained using different features including the output of the first three mode. Below is the performance of using different taggers independently comparing to the final CRF model. Some taggers also produce category names (POS tag or genomics type), we show the performance of each categories and also the tagger as a whole.

As shown in the Table below, the CRF model using all tagger outputs as feature performed significantly better than all other models. However, this is tested on the training set, as training time is too long (20+ hours) for me to do any cross validation.

Systems	F1	Precision	Recall
My CRF Model	**0.8840**	**0.8646**	**0.9042**
GeneTag (all)	0.8067	0.7685	0.8488
Genia (protein_molecule)	0.3674	0.5669	0.2718
Genia (all)	0.2637	0.1739	0.5452
Genia (DNA_domain_or_region)	0.1368	0.2777	0.0908
StanfordPos_NN	0.1242	0.0731	0.4119
Genia (protein_family_or_group)	0.1064	0.3462	0.0629
Genia (other_name)	0.0326	0.0321	0.0332
Genia (other_organic_compound)	0.0252	0.0810	0.0149
StanfordPos_NNP	0.0245	0.0605	0.0154
Genia (protein_complex)	0.0196	0.3740	0.0101

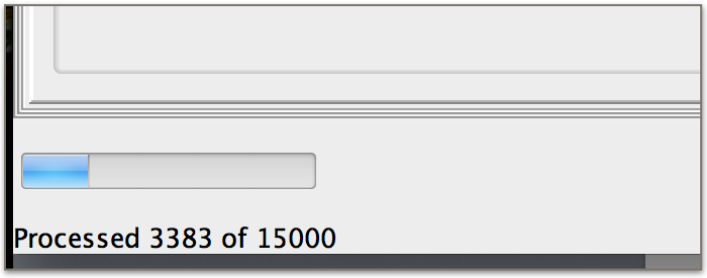
Type System



System Architecture

The system consists of 1 Collection Reader, 3 Primitive Annotator under 1 Aggregated Annotator, and 1 Consumer.

COLLECTION READER



Class SingleFileCollectionReader inherits the abstract class CollectionReader_ImplBase, which reads from the input file line by line. For each line in the input file, the ID and sentence part is parsed, and one CAS and model.Sentence object is created to store the sentence ID. After each line read the data object is added to the jcas index and moved on to later components. **One object is created for each call to getNext()**. **No file contents nor data object is stored in the memory of the object. Progress is calculated by number of lines read and the total number of lines in the file**. This give us a good idea of the progress during execution. However, it also requires determining the total number of lines in the input file in

the initializer, which can be time consuming if the input file is large. The input file stream is opened once, when the object initializes, and closed when the closed function is called by the UIMA framework.

ANALYSIS ENGINE / ANNOTATOR 1

The first AE / Annotator uses the Stanford NLP Core to predict part-of-speech tag for each word token in the input sentences. Model loading and memory management is handled by the library. This is the modified version of the NN Gene tagger from homework1.

ANALYSIS ENGINE / ANNOTATOR 2 AND 3

Class RangeExtractionAnnotator inherits the abstract class JCasAnnotator_ImplBase, which uses the pre-trained GENETAG and Genia taggers for identifying gene sequences in the input sentence. The input of this component is the output of the collection reader, i.e. model.Sentence objects each with a sentence ID and its text. For each of the model.Sentence object the component outputs zero or more model.Gene objects depending on how many gene sequence is identified. The output object has identical sentence ID and text of its corresponding input model.Sentence object, and an additional gene sequence name. Each model.Gene object indicates one identified gene sequence. The range attributes is now used to store the index range of where the gene sequence appeared in the sentence text. **The detail implementation uses the strategy pattern and the singleton pattern, see Design Pattern Used Section for more information.**

AGGREGATED ANALYSIS ENGINE / ANNOTATOR

The aggregated analysis engine runs the three primitive AEs in order, and is referenced by the CPE descriptor.

CAS CONSUMER

Class GeneConsumer inherits the abstract class CasConsumer_ImplBase, which receives model.Gene objects from the analysis engine as inputs. The CRF model is used here to combine all annotations from the aggregated AE to produced the final output. In my opinion I think it makes more sense to implement the CRF model as the forth AE, but I read it on Piazza that we should combine results in the consumer, so thats what I did. **The the range indexes (begin / end) is fixed here before writing to file.** The output file stream is opened once in the class initializer. Since processCas may be called multiple times by the UIMA framework, the output file stream is closed when when the destroy() function is called by the framework.

CRF MODEL

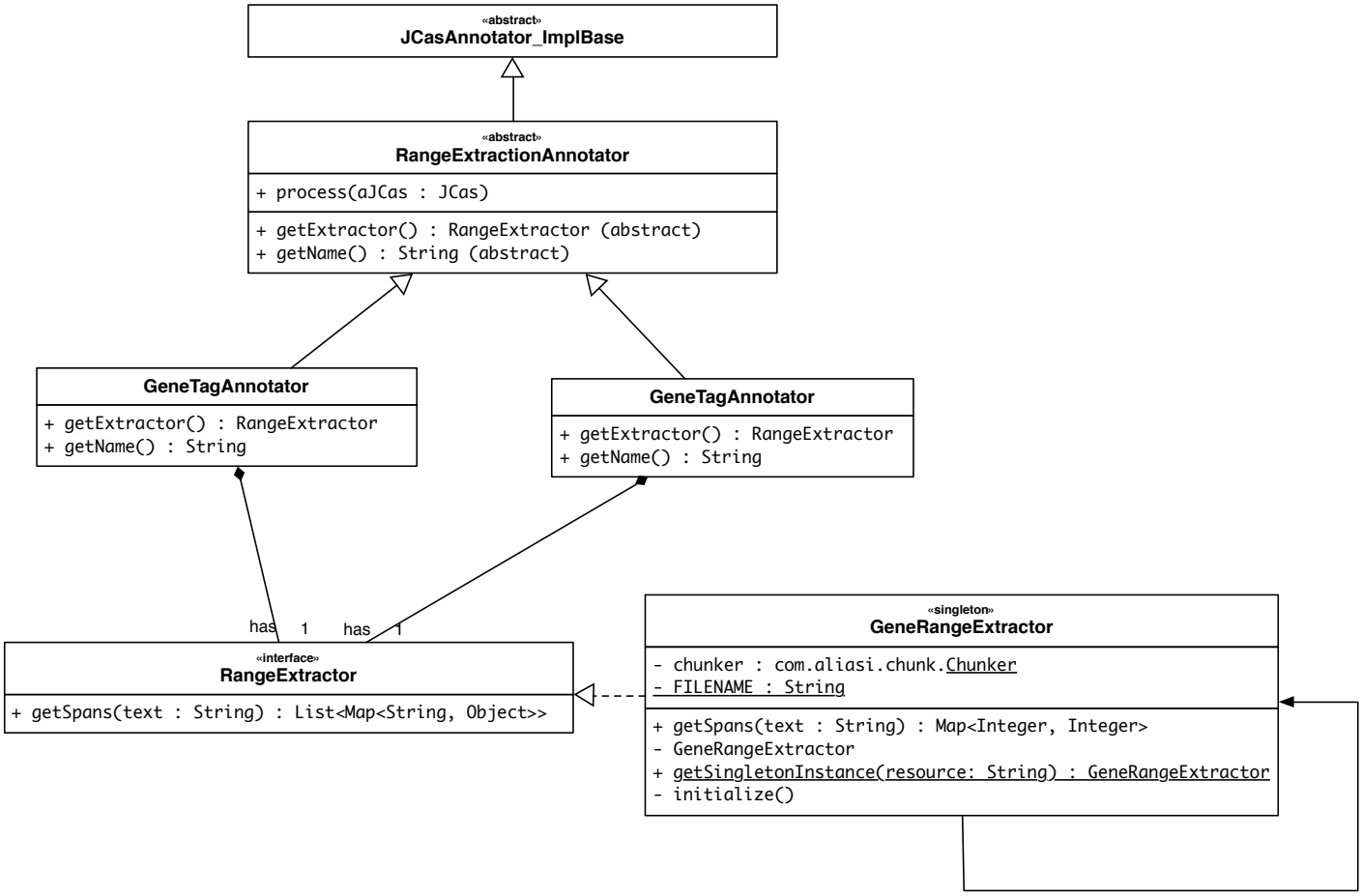
The CRF Model is trained on the output of all three annotators under the AAE. The observables is the character stream of the input sentence. For example, the features for the 16th character "*a*" in the sentence "*Comparison with alkaline phosphatases and 5-nucleotidase*" (P00001606T0076) is shown below:

```
CHAR_IS_a, GeneTag_GENE, Genia_protein_molecule, StanfordPos_NN
```

Feature template is implemented by SimpleChainCrfFeatureExtractor.java. For node features I used: 1) current token, 2) previous token, 3) next token, 4) previous bigram, 5) next bigram 6) surrounding trigram, and for edge features I used: 7) previous tag,

and; 8) current token and previous tag. The training time is 24 hours and 2 minutes, and model converged after 988 iterations. The details are in the version4.1000.log file.

Design Patterns Used



THE STRATEGY PATTERN

The RangeExtractionAnnotator is a generic range annotator that for a given input text and an extractor object that implements the RangeExtractor interface, will used the input strategy to extract ranges in the input text. In our case, the realization of the extrac-

tor is the GeneRangeExtractor class that uses the LingPipe package with a pre-trained model to extract gene mentions in the given text. However, any other classes that implements the RangeExtractor interface can also by plug into the annotator at runtime.

I choose to use a abstract getExtractor() function call to obtained different extractor strategy in the concreted classes instead of using the UIMA context, because I have models from different libraries that needs to be loaded using different codes. Using Java inheritance to load different model gives me more flexibility than using the UIMA context.

THE SINGLETON PATTERN

The GeneRangeExtractor strategy is also implemented according to the singleton pattern. Since it is a generic algorithm that can be used throughout different parts of a large system, and it requires loading a memory consuming model, we use the singleton pattern to insure that only one instance of this class can be created. This way, the models can only be load once into the memory.

Since in homework2 I am using more than one model, this class is slightly modified to hold more than one “singleton” instance. A Map object that map resources name of the model and its corresponding singleton object is maintained to make sure each model under resources is only loaded once, and only one corresponding object is created throughout runtime.