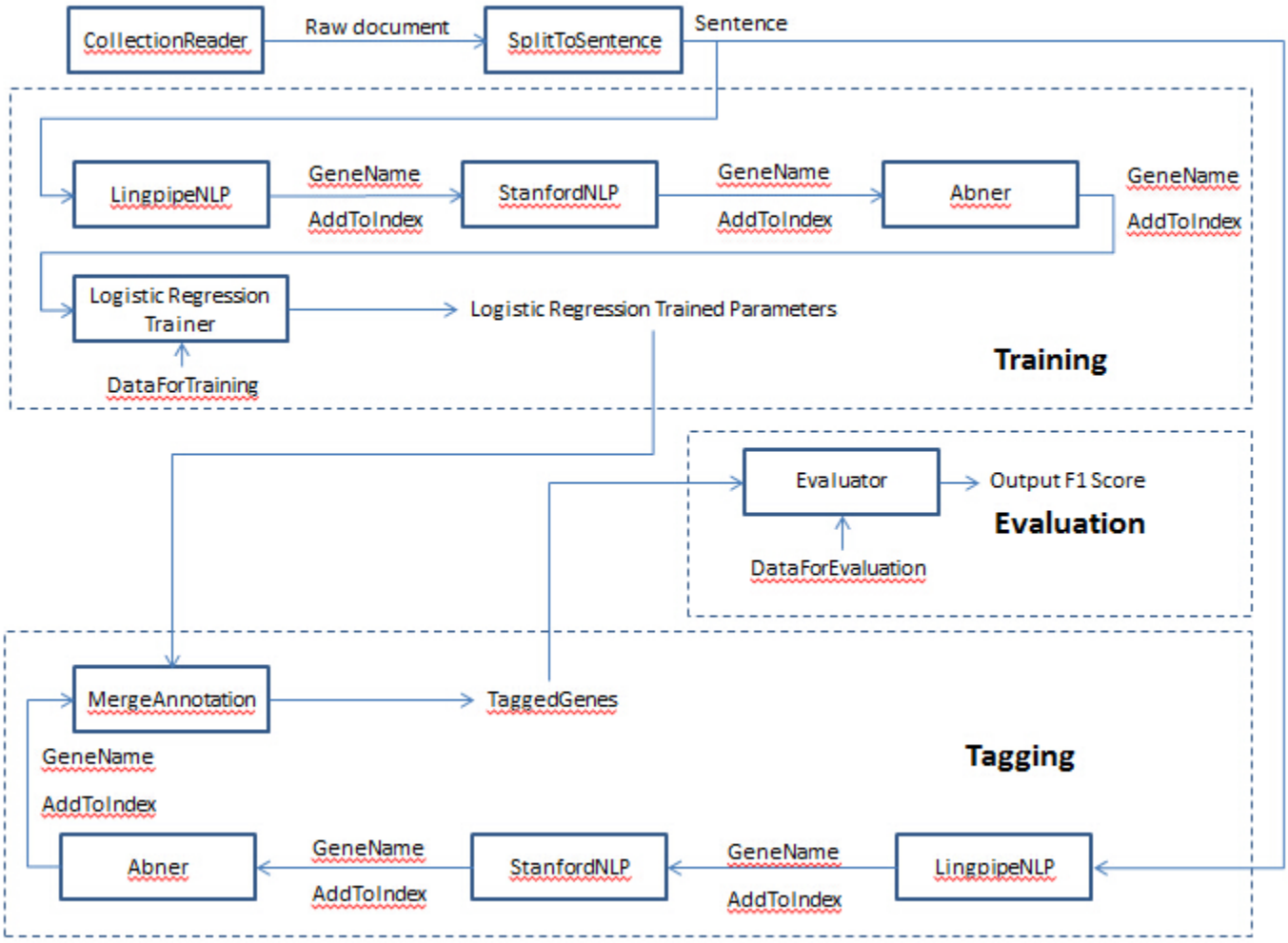
1: Architecture of Collection Processing Engine



There are 3 CPE descriptors in my project. TrainCPEDescriptor is just for training. TaggingCPEDescriptor is only for tagging. TrainAndTaggingCPEDescriptor is for both training and tagging, that is, do the training first, and with the weights we get from the training, then do tagging. The default CpeDescriptor will do tagging only, with the parameters we have trained before.

2: Tips for running my program

Generally there are 3 phases in my program, training, tagging, evaluating.

If you run my program using grader.sh, the default mode will be going through tagging and evaluating only, with the parameters I already trained. Note that default training data set and data for evaluation all comes from sample.out. I've adjusted some parameters to prevent overfit. If you replace the training set with your own tagged data set, I believe the performance will be much better.

For running under eclipse, if you only want to do training, please use the CPE descriptor src/main/resources/training/TrainCPEDescriptor.xml. It will first read contents from src/main/resources/data/hw2.in and start tagging. Then it will compare the tagging result with src/main/resources/dataForTraining . Then a Logistic Regression Algorithm will be run to adjust the weight of each annotator, to reduce the difference between our tagging and dataForTraining, which is actually a part of sample.out. However, the evaluation is based on the assumption that phrases in sample.out are all correct answers. So if you have another tagged dataset which you consider to be the right tagging, please go to src/main/resources/descriptors/training/TrainerDescriptor.xml. Under the Resources tag, you will find the external resource named dataForTraining. Replace that file with your own dataset. Please make sure the format of the file content remains the same.

After training completed, you will get 4 parameter, that is, weight for Lingpipe, StanfordNLP, Abner, and bias. With these parameters, we can combine the results from Lingpipe, StanfordNLP and Abner to generate a reasonable answer.

3: Type System

Sentence The type sentence has two attributes, String id and String text. Collection reader reads raw document and pass it to SplitToSentence.java. Then it splits the raw document and generates the Sentence type to store the ID and text of each sentence.

GeneName This type inherits from edu.cmu.deiis.Annotation. The type gene 4 four attributes. Two of them come from Annotation type, that is, Confidence and casProcessorID. Confidence records the probability that this is a real string name. casProcessorID records which annotator this information comes from. Id stores the id of the sentence it belongs to. GeneName stores the name of genes.

TaggedGenes After merging all the gene names coming from different annotators, we can generate the TaggedGenes type. In this type there are features including Text, Name, Id, Begin and End. We first remove all the white spaces from Text and Name, so that we can calculate the non-whitespace offset of each gene name, and store it in Begin and End feature.

LRPrarmeter Store the weights generated from the trainer, and pass it to the tagger.

3: NLP Tools Applied

I tried Lingpipe, Stanford NLP and Abner.

This part is pretty much the same as hw1. The only different thing is that this time we need to care about the confidence. For the Lingpipe, if you use the ConfidenceChunker it will generate a confidence for each phrase it extract from the sentence and you can just use it. For the other two, no confidence is given. So I just assign a constant number as their confidence for each phrase.

4: The Idea of How to Merge the Result

The three NLP tools I applied have very different performances regarding the precision. If will make no sense if I just treat their results equally and select the final result by voting. So I want to assign different weights to each annotators. To find the proper weights, I used a Logistic Regression algorithm to train on the sample.out, as the golden standard.

First we define a cost function to denote the difference between our output and the gold standard output.

$$x^{(i)} = \vec{\theta} \cdot \vec{conf}_i$$

$$h_\theta(x^{(i)}) = sigmoid(x^{(i)})$$

$$sigmoid(x) = \frac{1}{1+e^{-x}}$$

$$Cost(\theta, x) = -\frac{1}{m}(\sum_1^m y^{(i)} log(h_\theta(x^{(i)})) + (1-y^{(i)}) log(1-h_\theta(x^{(i)})))$$

m is the number of gene name candidates. θ are the weights for each feature.

Now we need to minimize the Cost Function. Here we use Gradient Descent to decrease the Cost function iteratively. Consider the partial derivative on θ_j,

$$\theta_j := \theta_j - \alpha \frac{\partial J(\theta)}{\partial \theta_j}, j = 1, ..., n$$

n is the number of features, α indicates how much we will change the θ in one iteration. And

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m} \sum (h_\theta(x^{(i)}) - y^{(i)} x_j^{(i)})$$

With a properly initialized θ, α,we can begin our training on the sample.out, and hopefully we can get a set of parameters, which are the weights for each annotator components. With these weights, we can decide how to combine and merge the results from the three annotators. A very interesting fact is that the weight for StanfordNLP is negative, which means if the stanford NLP thinks a phrase is a gene name, it probably is not. It's not surprising since the Stanford NLP works in a very naive way.

A set of parameters which I have trained was stored under src/main/resources/defaulParameter. These parameters will be read and used if you do the tagging directly using grader.sh.

6: Conclusion

Actually the F-1 score isn't as good as I have expected though I have tried adjusting the parameters many times. Even it's worse than using Lingpipe only. But it's not too surprising for me because the sample.out is produced by Lingpipe. Using other annotators may have negative effects on the output, especially the Stanford NLP, which obviously is a bottle neck because it works in a naive way. More work needs to be done to improve the accuracy of it. Also, we need to ensure the data set used for training is resonable, and be careful not to overfit the training set.