

# Relation Extraction using MultiR

15th August 2014

## Abstract

# Chapter 1

## Sg

### 1.1 Bird's eye view of things

There are 2 separate repositories : MultirFramework and MultirExperiments. Latter is the latest version and the one you should be using for development.

#### 1.1.1 Training

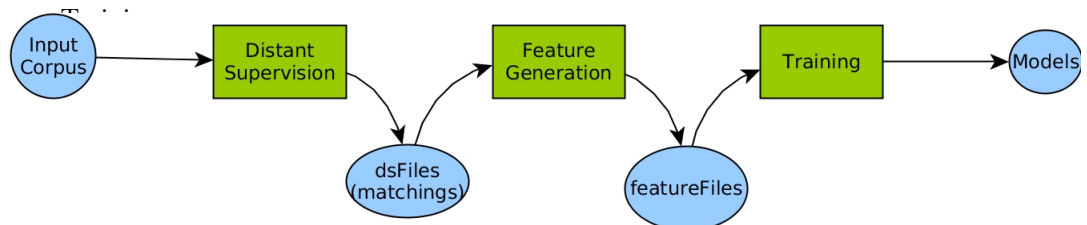


Figure 1.1: The Distant supervision training process

`knowitall/MultirExperiments/blob/master/src/main/java/edu/washington/multir/experiment/Experiment.java` is a good place to start. It covers basically the whole of the Distant supervision pipeline.

#### 1.1.2 Extraction

TODO

### 1.2 Setting the stage

There are several ways of implementing each of the blackboxes shown in the figure above. Thus, before starting, we need to decide on which implementations

to use for each of them. Once we have zeroed in on the choices, we need to supply this information to the orchestrator. The way we do that in the current setup is via a configuration file, which is essentially a json file consisting of key value pairs. Keys are the black boxes and the values are the particular implementation to use.

As the diagram shows, the Distant supervision process (ds process henceforth) also needs to know where is the corpus, the kb and what to call the intermediate files. All of this is also supplied by the configuration file. A standard configuration file will look as follows :

```

1 {
2   "corpusPath" :
3   "jdbc:derby:/mnt/a99/d0/aman/multir-exp-second/MultirExperiments-master/data/FullCorpus-Fi
4   "train" : "true",
5   "testDocumentsFile" :
6   "data/emptyFile",
7   "fg" :
8   "edu.washington.multirframework.featuregeneration.DefaultFeatureGenerator",
9   "ai" :
10  "edu.washington.multirframework.argumentidentification.NERArgumentIdentification",
11  "rm" :
12  "edu.washington.multirframework.argumentidentification.NERRelationMatching",
13  "nec" :
14  "edu.washington.multirframework.distant supervision.NegativeExampleCollectionByRatio",
15  "necRatio" : "0.25",
16  "kbRelFile" : "data/kb/kb-facts-train.tsv.gz",
17  "kbEntityFile" : "data/kb/entity-names-train.tsv.gz",
18  "targetRelFile" : "data/kb/target-relations.tsv",
19  "typeRelMap" : null,
20  "sigs" : [
21  "edu.washington.multirframework.argumentidentification.DefaultSententialInstanceGeneration
22  ],
23  "dsFiles" : [
24  "data/training-instances.tsv.gz"
25  ],
26  "featureFiles" : [
27  "data/training-instances-features.tsv"
28  ],
29  "models" : [
30  "data/multir-extractor" ],
31  "cis" :
32  "edu.washington.multirframework.corpus.DefaultCorpusInformationSpecification",
33  "si" : [
34  "edu.washington.multirframework.corpus.SentNamedEntityLinkingInformation",
35  "edu.washington.multirframework.corpus.SentFreebaseNotableTypeInformation"
36  ],
37  "ti" : [
      "edu.washington.multirframework.corpus.TokenChunkInformation"
    ],

```

```
38 "di" : [ ],
39 "useFiger" : "false"
40 }
```

We will be referring to the configuration file as and when required.

## Chapter 2

# The Distant supervision process

The first block of the pipeline, Distant supervision is responsible for reading the corpus and generating matching the sentence with the KB. In this chapter, we look at how MultiR handles the whole process.

## 2.1 Distant supervision process

### 2.1.1 Responsibilities

We start with a quick enumeration of steps involved in the process.

1. Read the corpus
2. Iterate over it sentence by sentence
3. Find named entities pairs in each sentence
4. Match the pairs found in step 4 to the facts that you know about entities (a.k.a the knowledge base)
5. Dump matched relations for each sentence in a separate file <sup>1</sup> (to be used in training)

We now read through the code and match the steps listed above with actual implementation.

### 2.1.2 Code

It will be easier to follow the code if you take the following at the moment as gospel's truth : **Annotation** is a document with attached information,

---

<sup>1</sup>This file is specified in the configuration file by the property "dsFiles". The Distant supervision process runs only if this file does not exists.

**CoreMap** is a sentence with attached information, **CoreLabel** is a word with attached information. We plan to define these in detail in a separate chapter. We do not follow a strict dfs order while explaining the code; some method calls are skipped for brevity.

---

```
DistantSupervision ds = new DistantSupervision(ai, sigs.get(0), rm, nec);
ds.run(DSFiles.get(0), kb, corpus)
```

---

The above lines in `experiment.java` kickstart the whole process. Once inside the `run` method of `DistantSupervision`, things proceed as follows :

### Iterating over documents

First of all, we go to the corpus to get an iterator of the documents.

---

```
Iterator<Annotation> di = c.getDocumentIterator();
while(di.hasNext()) {
    Annotation d = di.next(); //get the next document
    List<CoreMap> sentences =
        d.get(CoreAnnotations.SentencesAnnotation.class); //get the
        sentences in the document
    List<NegativeAnnotation> documentNegativeExamples = new ArrayList<>();
        //create space for negative annon in this doc
    List<Pair<Triple<KBArgument,KBArgument,String>,Integer>>
        documentPositiveExamples = new ArrayList<>();
```

---

Iterate over the corpus document by document. The document iterator is returned with the help of `Corpus.java` and `CorpusHandler.java`. These classes abstract the Apache derby database from the user.

### 2.1.3 Identifying entities of interest in text

---

```
for(CoreMap sentence : sentences) {
    int sentGlobalID = sentence.get(SentGlobalID.class);
    //argument identification
    List<Argument> arguments = ai.identifyArguments(d,sentence);
    //sentential instance generation
    List<Pair<Argument,Argument>> sententialInstances =
        sig.generateSententialInstances(arguments, sentence);
    //relation matching
```

---

There are two crucial functions being used here.

- **identifyArguments** : NER based argument identifier returns a list of named entity arguments by scanning the annotated `CoreMap` (sentences) for `CoreLabels` (Words), the property `CoreAnnotations.NamedEntityAnnotation` of each of the labels and only returning those words that have NER tags among those listed in `relevantNER`.

- **SententialInstanceGeneration** : Takes a list of arguments and a sentence (CoreMap, annotated sentence in the sfu nlp lib parlance) and returns a list of pairs that are non overlapping along with the sentences. Also makes sure that two arguments that are actually the same are not added as a pair. This is done by making sure that they don't have the same id.

#### 2.1.4 Matching relations

---

```
List<Triple<KBArgument,KBArgument,String>> distantSupervisionAnnotations
=
    rm.matchRelations(sententialInstances, kb, sentence, d);
```

---

Once we have identified the entities of interest in each of the sentences, the next step is to create the distantly supervised training data. This involves taking the entity pairs of interest (obtained by SententialInstanceGeneration) and using the kb to see if there are any matches. The NERRelationMatching has pretty simple implementation. For each pair of entities, it first maps the entities to freebase ids and then queries the knowledge base for all possible relations that exist between the entity pair.

#### 2.1.5 Adding sentence ids and finding negative examples

---

```
//adding sentence IDs
List<Pair<Triple<KBArgument,KBArgument,String>,Integer>>
    dsAnnotationWithSentIDs = new ArrayList<>();
for(Triple<KBArgument,KBArgument,String> trip :
    distantSupervisionAnnotations){
    Integer i = new Integer(sentGlobalID);
    Pair<Triple<KBArgument,KBArgument,String>,Integer> p = new
        Pair<>(trip,i);
    dsAnnotationWithSentIDs.add(p);
}
//negative example annotations
List<NegativeAnnotation> negativeExampleAnnotations =
    findNegativeExampleAnnotations(sententialInstances,distantSupervisionAnnotations,kb,sentGlobalID);
documentNegativeExamples.addAll(negativeExampleAnnotations);
documentPositiveExamples.addAll(dsAnnotationWithSentIDs);
```

---

The negative examples are formed by the entity pairs that appear together but are not there in the knowledge base.

#### 2.1.6 Writing the matches to the disk

---

```
writeDistantSupervisionAnnotations(documentPositiveExamples,dsWriter);
```

---



```
writeNegativeExampleAnnotations(nec.filter(documentNegativeExamples,  
    documentPositiveExamples,kb,sentences),dsWriter);
```

---

The matches are stored on the disk in the a tab separated file (tsv) which is compressed to a gz file. The file can be specified using dsFiles property in the configuration. For example, if the sentence is **"Pravda is a popular newspaper of Moscow"**, the match would be saved as

```
/m/013v50 0 0 Pravda /m/04swd 5 5 Moscow 102 newsPaperIn
```

We will use this example to again go through the activities that take place in the Distant supervision blackbox to reach this state: this sentence would have been fetched from the corpus, Pravda and Moscow would have been identified as Named entities (ORG and LOC respectively), relation matching would have found an entry with these 2 entities in the KB and finally, it will be written down in the file specified by dsFiles.

## Chapter 3

# Feature Generation

### 3.0.7 Starting

The process of feature generation starts from

---

```
public void run(List<String> dsFileNames, List<String> featureFileNames,  
                Corpus c, CorpusInformationSpecification cis)
```

---

The first argument contains the list of matchings. You will expect that being here means that dsFileNames have been populated. The second argument is the list of files where the corresponding features have to be written. Next we have objects that help in getting corpus information.

Once in run, the major step is

---

```
List<SententialArgumentPair> saps =  
    getSaps(dsFileNames, featureFileNames);
```

---

This is familiar sentential argument pair from distance supervision. Argument pairs again are an abstraction that contains types, offsets, the sentence id and other similar information.

### 3.0.8 Storing entity pairs in each sentence

We next populate the following data structure :

---

```
Map<Integer, List<SententialArgumentPair>> sapMap = new HashMap<>();
```

---

This is a mapping which given a sentence number, returns all the arguments that are there in that sentence.

We then start iterating over the corpus. Recall that this is the same corpus that was used during distance supervision.

---

```

//iterate over corpus
Iterator<Annotation> di = c.getDocumentIterator();
int docCount =0;
start = System.currentTimeMillis();
while(di.hasNext()){
    Annotation doc = di.next();

    List<CoreMap> sentences =
        doc.get(CoreAnnotations.SentencesAnnotation.class);
    for(CoreMap sentence: sentences){

        Integer currSentID =
            sentence.get(SentGlobalID.class);
        if(sapMap.containsKey(currSentID)){
            System.out.println(sentence);
            List<SententialArgumentPair> sentenceSaps =
                sapMap.get(currSentID);
            writeFeatures(sentenceSaps,doc,sentence,writerMap);
        }
    }
    docCount++;
}

```

---

The above piece of code iterates over the corpus sentence by sentence. For a given sentence, the map created in the above step is used to extract all the entity pairs that exists in a particular sentence. if a sentence has some entity pairs, we extract all of them and then call writeFeatures, supplying the list of entity pairs just extracted. Lets now see how things look in writeFeatures()

### 3.0.9 Writing features to files : writeFeatures()

---

```

for(SententialArgumentPair sap : currentSaps){
    BufferedWriter bw = writerMap.get(sap.partitionID);

    List<String> features =
        fg.generateFeatures(sap.arg1offsets.first,sap.arg1offsets.second
            ,sap.arg2offsets.first,sap.arg2offsets.second,sap.arg1ID,sap.arg2ID);
    bw.write(makeFeatureString(sap,features)+"\n");
}

```

---

The features are based upon the entity pairs that exists in a sentence. Thus, MultiR iterates over the list of entity pairs and iterates over them one by one, calling generateFeatures() for each of these entities.

### 3.0.10 Generating Features

We will be discussing the generateFeatures() function of the default feature generator here. The signature of the function looks as follows :

---

```
public List<String> generateFeatures(Integer arg1StartOffset,
    Integer arg1EndOffset, Integer arg2StartOffset,
    Integer arg2EndOffset, String arg1ID, String
        arg2ID, CoreMap sentence, Annotation document)
{
```

---

The generate features function is supplied with information like offset and IDs of the entities that form the pair, the sentence and the document involved. Also supplied are the annotated sentence. Then we iterate over the sentence token by token, and it is checked whether the current token is one of the argument. Next up, we iterate over the dependency parse information and record it into an array.

Finally, a call is made to the OriginalMultirFeatureGenerator()

---

```
originalMultirFeatures(tokenStrings, posTags, depParents, depTypes,
    arg1Pos, arg2Pos, arg1ner, arg2ner);
```

---