## Introduction

This component is implemented as a part of the OpenNLP similarity component for easy integration with search engines written in Java, especially those that are Lucene/SOLR based. Aside from taxonomy learning, the similarity component performs text relevance assessment, accepting two portions of the text (phrases, sentences, paragraphs) and returning a similarity score. The similarity component can be used on top of the search to improve the relevance; it computes a similarity score between a question and all of the search results, computing a taxonomy match and syntactic generalization.

The Apache OpenNLP library is a machine learning-based toolkit for the processing of natural language text. It supports the most common NLP tasks, such as tokenization, sentence segmentation, part-of-speech tagging, named entity extraction, chunking, parsing, and coreference resolution. These tasks are usually required to build more advanced text processing services. OpenNLP also includes maximum entropy and perceptron-based machine learning. The similarity component is an extension of OpenNLP that is intended for search and machine learning engineers to allow easy integration into Java-based production environments to perform tasks that are involved in testing the relevance of the responses.

Additionally, this component is useful for web mining images, videos, forums, blogs, and other media that have textual descriptions. Applications such as content generation and filtering meaningless speech recognition results are included in the sample applications of this component. Relevance assessment is based on machine learning of syntactic parse trees (constituency trees). The similarity score is calculated as the size of all of the maximal common sub-trees for sentences from a pair of texts.

The objective of the similarity component is to give an applications engineer a tool for determining text relevance that can be used as a black box. As a result, there is no need to understand computational linguistics or machine learning.

The code is available at <https://svn.apache.org/repos/asf/incubator/opennlp> and also at <https://code.google.com/p/relevance-based-on-parse-trees>. For a quick start, a user needs to download the latest opennlp-similarity…jar and look at a few tests in the source code showing the main functionality.

## Constructing Parse Thicket

The entry point to main functionality of parse thickets is the Matcher class in package opennlp.tools.parse\_thicket.matching. To build the parse thicket from text, use

function ParseThicket buildParseThicketFromTextWithRST(String para)

To compute similarity between two paragraphs of text, use

**public** List<List<ParseTreeChunk>> assessRelevance(String para1, String para2) {

// first build PTs for each text

ParseThicket pt1 = ptBuilder.buildParseThicket(para1);

ParseThicket pt2 = ptBuilder.buildParseThicket(para2);

// then build phrases and rst arcs

List<List<ParseTreeNode>> phrs1 = phraseBuilder.buildPT2ptPhrases(pt1);

List<List<ParseTreeNode>> phrs2 = phraseBuilder.buildPT2ptPhrases(pt2);

// group phrases by type

List<List<ParseTreeChunk>> sent1GrpLst = formGroupedPhrasesFromChunksForPara(phrs1),

sent2GrpLst = formGroupedPhrasesFromChunksForPara(phrs2);

List<List<ParseTreeChunk>> res =

phraseMatcher .matchTwoSentencesGroupedChunksDeterministic(sent1GrpLst, sent2GrpLst);

**return** res;

}

The construction of parse thicket from text occurs in a singleton class ParseCorefsBuilder which does parsing and coreference resolution via Stanford NLP and then adds Speech Act arcs using the package opennlp.tools.parse\_thicket.communicative\_actions. The following code snippets shows the data staructures used to form the additional arcs to the ones obtained for anaphore:

**private** List<WordWordInterSentenceRelationArc> buildCAarcs(

List<List<ParseTreeNode>> nodesThicket) {

List<WordWordInterSentenceRelationArc> arcs = **new** ArrayList<WordWordInterSentenceRelationArc>();

**for**(**int** sentI=0; sentI<nodesThicket.size(); sentI++){

**for**(**int** sentJ=sentI+1; sentJ<nodesThicket.size(); sentJ++){

List<ParseTreeNode> sentenceI = nodesThicket.get(sentI),

sentenceJ = nodesThicket.get(sentJ);

Pair<String, Integer[]> caI = caFinder.findCAInSentence(sentenceI);

Pair<String, Integer[]> caJ = caFinder.findCAInSentence(sentenceJ);

**int** indexCA1 = caFinder.findCAIndexInSentence(sentenceI);

**int** indexCA2 = caFinder.findCAIndexInSentence(sentenceJ);

**if** (caI==**null** || caJ==**null**)

**continue**;

Pair<String, Integer[]> caGen = caFinder.generalize(caI, caJ).get(0);

ArcType arcType = **new** ArcType("ca",

caGen.getFirst().toString()+printNumArray(caGen.getSecond()), 0, 0);

WordWordInterSentenceRelationArc arc =

**new** WordWordInterSentenceRelationArc(**new** Pair<Integer, Integer>(sentI,indexCA1),

**new** Pair<Integer, Integer>(sentJ,indexCA2),

caI.getFirst(), caJ.getFirst(), arcType);

arcs.add(arc);

}

}

**return** arcs;

}

The reader is advised to consult tests which show the results of generalization of two paragraphs of text at PairwiseMatcherTest and PT2ThicketPhraseBuilderTest to observe the difference between the pair-wise sentence generalization without use of PT, and PT-generalization respectively.

## Running PT-supported search

To start with this component, please refer to SearchResultsProcessorTest.java in package opennlp.tools.similarity.apps to observe pairwise sentence match for passage re-ranking

public void testSearchOrder() runs web search using Bing API and improves search relevance.

Look at the code of public List<HitBase> runSearch(String query)

and then at BingResponse calculateMatchScoreResortHits(BingResponse resp, String searchQuery)

which gets search results from Bing and re-ranks them based on computed similarity score.

The main entry to Similarity component is

SentencePairMatchResult matchRes = sm.assessRelevance(snapshot, searchQuery);

where we pass the search query and the snapshot and obtain the similarity assessment structure which includes the similarity score.

To run this test you need to obtain search API key from Bing at http://datamarket.azure.com/and specify it in BingQueryRunner inAPP\_ID.

To proceed to PT-supported search, the reader can consult MultiSentenceSearchResultsProcessorTest.

The algorithm for search results re-ranking is as follows:

List<HitBase> calculateMatchScoreResortHits(List<HitBase> hits,

String searchQuery) {

List<HitBase> newHitList = **new** ArrayList<HitBase>();

**int** count = 0;

**for** (HitBase hit : hits) { // get a webpage and a snippet for url

String[] pageSentsAndSnippet = formTextForReRankingFromHit(hit);

Double score = 0.0;

List<List<ParseTreeChunk>> match = **null**;

match = matcher.assessRelevanceCache(pageSentsAndSnippet[0] ,

searchQuery);

score = parseTreeChunkListScorer.getParseTreeChunkListScore(match);

**if** (score < 2){ // attempt to match with snippet, if failed with original text

match = matcher.assessRelevanceCache(pageSentsAndSnippet[1],

searchQuery);

score = parseTreeChunkListScorer.getParseTreeChunkListScore(match);

}

hit.setGenerWithQueryScore(score);

newHitList.add(hit);

}

Collections.*sort*(newHitList, **new** HitBaseComparable());

**return** newHitList;

}

## Hierarchy of generalization operation

Text generalizer **Matcher.java**

Matcher

Mentions & same entity generalizer

Paragraph generalizer

**Matcher.java**

Communicative actions generalizer

Pairs of communicative actions generalizer

Mentions & same entity generalizer

Rhetoric structure generalizer

Sentence generalizer

**ParseTreeMatcher.java**

Noun phrase generalizer

Phrase group generalizer

**PhraseGroupGneralizer.java**

Verb phrase generalizer

Phrase generalizer

**PhraseGeneralizer.java**

Verb generalizer

Word generalizer

**PartseTreeNodeGeneralizer.j**ava

NER generalizer

Part-of-speech generalizer

**PartOfSpeechGeneralizer.jav**a

Lemma generalizer

**LemmaGeneralizer.jav**a

Fig.1: Hierarchy of generalizations: from paragraphs to word features

Hierarchy of syntactic generalization (SG) is shown in Fig. 8. On the left, the levels of SG are enumerated from top to bottom. On the top, the level of an arbitrary text including multiple paragraphs is shown. Since we do not develop a model for the whole text, based on parse trees of sentences, text (document) similarity is implemented as a pair-wise SG of its paragraphs. The lowest level of hierarchy is SG of lemmas (words) , according to Section 2.5, is shown on the bottom. On the right, additional SG components are depicted, which rely on additional information to generalize specific features, from discourse structures on the top-right to the features of individual words on the right-bottom.

### First use case of Similarity component: search

To start with this component, please refer to SearchResultsProcessorTest.java in package opennlp.tools.similarity.apps

public void testSearchOrder() runs web search using Bing API and improves search relevance.

Look at the code of

public List<HitBase> runSearch(String query)

and then at

private BingResponse calculateMatchScoreResortHits(BingResponse resp, String searchQuery)

which gets search results from Bing and re-ranks them based on computed similarity score.

The main entry to Similarity component is

SentencePairMatchResult matchRes = sm.assessRelevance(snapshot, searchQuery);

where we pass the search query and the snapshot and obtain the similarity assessment structure which includes the similarity score.

To run this test you need to obtain search API key from Bing at www.bing.com/developers/s/APIBasics.html and specify it in

public class BingQueryRunner in

protected static final String APP\_ID.

### Solving a unique problem: content generation

To demonstrate the usability of Similarity component to tackle a problem which is hard to solve without a linguistic-based technology, we introduce a content generation component:

RelatedSentenceFinder.java

The entry point here is the function call

hits = f.generateContentAbout("Albert Einstein");

which writes a biography of Albert Einstein by finding sentences on the web about various kinds of his activities (such as 'born', 'graduate', 'invented' etc.).

The key here is to compute similarity between the seed expression like "Albert Einstein invented relativity theory" and search result like

"Albert Einstein College of Medicine | Medical Education | Biomedical ...

www.einstein.yu.edu/Albert Einstein College of Medicine is one of the nation's premier institutions for medical education, ..."

and filter out irrelevant search results.

This is done in function

public HitBase augmentWithMinedSentencesAndVerifyRelevance(HitBase item, String originalSentence,

List<String> sentsAll)

SentencePairMatchResult matchRes = sm.assessRelevance(pageSentence + " " + title, originalSentence);

You can consult the results in gen.txt, where an essay on Einstein bio is written.

These are examples of generated articles, given the article title

[www.allvoices.com/contributed-news/9423860/content/81937916](http://www.allvoices.com/contributed-news/9423860/content/81937916) and

[www.allvoices.com/contributed-news/9415063](http://www.allvoices.com/contributed-news/9415063)

### Solving a high-importance problem: filtering out meaningless speech recognition results.

Speech recognitions SDKs usually produce a number of phrases as results, such as

"remember to buy milk tomorrow from trader joes",

"remember to buy milk tomorrow from 3 to jones"

One can see that the former is meaningful, and the latter is meaningless (although similar in terms of how it is pronounced). We use web mining and Similarity component to detect a meaningful option (a mistake caused by trying to interpret meaningless request by a query understanding system such as Siri for iPhone can be costly).

SpeechRecognitionResultsProcessor.java does the job:

public List<SentenceMeaningfullnessScore> runSearchAndScoreMeaningfulness(List<String> sents)

re-ranks the phrases in the order of decrease of meaningfulness.

Similarity component internals are in the package opennlp.tools.textsimilarity.chunker2matcher

ParserChunker2MatcherProcessor.java does parsing of two portions of text and matching the resultant parse trees to assess similarity between

these portions of text.

To run ParserChunker2MatcherProcessor

private static String MODEL\_DIR = "resources/models"; needs to be specified

The key function

public SentencePairMatchResult assessRelevance(String para1, String para2)

takes two portions of text and does similarity assessment by finding the set of all maximum common subtrees

of the set of parse trees for each portion of text

It splits paragraphs into sentences, parses them, obtained chunking information and produces grouped phrases (noun, evrn, prepositional etc.):

public synchronized List<List<ParseTreeChunk>> formGroupedPhrasesFromChunksForPara(String para)

and then attempts to find common subtrees:

in ParseTreeMatcherDeterministic.java

List<List<ParseTreeChunk>> res = md.matchTwoSentencesGroupedChunksDeterministic(sent1GrpLst, sent2GrpLst)

Phrase matching functionality is in package opennlp.tools.textsimilarity;

ParseTreeMatcherDeterministic.java:

Here's the key matching function which takes two phrases, aligns them and finds a set of maximum common sub-phrase

public List<ParseTreeChunk> generalizeTwoGroupedPhrasesDeterministic

Package structure is as follows:

opennlp.tools.similarity.apps : 3 main applications

opennlp.tools.similarity.apps.utils: utilities for above applications

opennlp.tools.textsimilarity.chunker2matcher: parser which converts text into a form for matching parse trees

opennlp.tools.textsimilarity: parse tree matching functionality.

### Comparison with bag-of-words approach

// we first demonstrate how similarity expression for DIFFERENT cases have

// too high score for bagOfWords

String phrase1 = "How to deduct rental expense from income ";

String phrase2 = "How to deduct repair expense from rental income.";

List<List<ParseTreeChunk>> matchResult = parser.assessRelevance(phrase1,

phrase2).getMatchResult();

*assertEquals*(

matchResult.toString(),

"[[ [NN-expense IN-from NN-income ], [JJ-rental NN-\* ], [NN-income ]], [ [TO-to VB-deduct JJ-rental NN-\* ], [VB-deduct NN-expense IN-from NN-income ]]]");

System.*out*.println(matchResult);

**double** matchScore = parseTreeChunkListScorer

.getParseTreeChunkListScore(matchResult);

**double** bagOfWordsScore = parserBOW.assessRelevanceAndGetScore(phrase1,

phrase2);

*assertTrue*(matchScore + 2 < bagOfWordsScore);

System.*out*.println("MatchScore is adequate ( = " + matchScore

+ ") and bagOfWordsScore = " + bagOfWordsScore + " is too high");

// we now demonstrate how similarity can be captured by POS and cannot be

// captured by bagOfWords

phrase1 = "Way to minimize medical expense for my daughter";

phrase2 = "Means to deduct educational expense for my son";

matchResult = parser.assessRelevance(phrase1, phrase2).getMatchResult();

*assertEquals*(

matchResult.toString(),

"[[ [JJ-\* NN-expense IN-for PRP$-my NN-\* ], [PRP$-my NN-\* ]], [ [TO-to VB-\* JJ-\* NN-expense IN-for PRP$-my NN-\* ]]]");

System.*out*.println(matchResult);

matchScore = parseTreeChunkListScorer

.getParseTreeChunkListScore(matchResult);

bagOfWordsScore = parserBOW.assessRelevanceAndGetScore(phrase1, phrase2);

*assertTrue*(matchScore > 2 \* bagOfWordsScore);

System.*out*.println("MatchScore is adequate ( = " + matchScore

+ ") and bagOfWordsScore = " + bagOfWordsScore + " is too low