Fall 2012

Extraction of Semantics from Primitive Concepts

Chak Li
San Jose State University

Extraction of Semantics from Primitive Concepts

A Project Report (CS298)

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Chak Li

Dec 2012

The Designated Project Committee Approves the Writing Project Titled

Extraction of Semantics from Primitive Concepts

by

Chak Li

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

Dec 2012

Dr. Tsau Young Lin — Date
Department of Computer Science

Dr. Soon Tee Teoh — Date
Department of Computer Science

Dr. Zhenzhen Kou — Date
Sharethis Inc.

Abstract

Due to the need to organize a vast amount of documents available in the Internet, the automated semantic extraction representing webpages has become a popular research topic in both industry and academia. The purpose of this project is to introduce a new method to process documents to extract the original contextual representations and yet to extend additional and connect similar representations based on the semantics underneath the extracted representations in an automatic fashion. Among the purposed steps, the core of this project is to tackle the difficulty to construct a mechanism in which machines can computationally understand the lexical meaning of the extracted semantic representations. For instance, the word "good" has the same lexical meaning as the word "well", so both should be equally treated. Furthermore, the 2-gram "wall street" should be kept as-is instead of tokenizing it into two single words, but "coffee or tea" should be tokenized into two single words "coffee" and "tea". This is important in text mining to keep but not to destruct the original semantics so one can further process documents safely, efficiently, and accurately.

In the project, I first discuss the adequate machine learning method introduced by Professor Lin to process documents to extract the original contextual representations, namely primitive concepts. Then, I introduce new methods to apply the extracted concepts to extract additional and connect similar representation based on the semantics underneath using the WordNet database. In the last section of the report, I examined the proposed data processing method with sample data and justified the empirical results with data provided by Google Search.

The project well articulates the problems of computation cost reduction and prediction enhancement in contextual extraction for documents. In general, most of the machine-learning article is well written and informative for general readers with Mathematics background, but not necessarily for readers of engineering interest. In the report, an engineering mechanism is constructed with mathematical reasoning to

persuade readers with theoretical background. Both readers from the engineering and mathematical communities are not to be left without an engineering and theoretical understanding of the methods introduced in the project.

Acknowledgements

I would like to first thank Professor Lin for his endless encouragement and project advice. The project was developed from stretch in the start of the year of 2012 and iteratively revised throughout the whole year. Professor Lin constantly motivated and supervised me to think wisely, research, and work hard for the project. I also thank my committee members Professor Teoh and Dr. Kou for their valuable comments, time and support. Day by day, and idea by idea, under the precious inputs and supervisions by the committee members, the project has grown to mature.

Contents

Abstract…….…….……..……...4-5

Acknowledgements……6

Contents…….…….7-8

List of Figures…….9

List of Tables……..10

1. Introduction

1.1 Research Background……11-12

1.2 Project Architecture……..12-13

2. Primitive Concept Extraction

2.1 Introduction to Primitive Concepts……..14

2.2 Stop Word Filtering..…….……..15-16

2.3 Algorithm to Extract Primitive Concepts…….……....16-18

2.4 Inside the Data Processor…..……..…….18-19

2.5 Summary..…….19

3. Analyzing and Extending Additional Representations

3.1 Introduction to WordNet……..20-21

3.2 Concept Tokenizer and its Algorithms……..21-24

3.3 Concept Extender using WordNet……24-25

3.4 Verification on Extended Topics using Google Search API……..25-26

3.5 Summary…….……...26-27

4. Analyzing and Connecting Primitive Concepts

4.1 Connecting Primitive Concepts……...…….…….28-34

4.2 Concept Graph and its Clusters.…….……34-35

4.3 Summary……...35

5. Implementation

5.1 Programming Languages used for Implementation……...36

6. Empirical results

6.1 Output by Data Processor….…….37-38

6.2 Output by Semantic Analyzer……38-40

7. Conclusion……...41

List of References…….……...42

List of Figures

Figure 1: Project architecture……12

Figure 2: Inside the data processor……19

Figure 3: Graphs of primitive concepts……33

List of Tables

Table 1: Top 10 words with highest TFIDF and their associated document number…...37

Table 2: Primitive concept samples……...38

Table 3: Top and last 10 extended topics based on number of search results……...39

Table 4. Semantic analysis on concepts in length from 1 to 6……..40

Introduction

1.1 Research Background

The automated semantic extraction for web pages has become a popular research topic in both industry and academia. Yet, the extracted web data has not achieved the level of sufficiency and remains noisy. The purpose of this project is to explain a new data processing mechanism and its method on how to efficiently extract valuable concepts namely primitive concepts from documents from both engineering and theoretical points of view.

Professor Lin introduced a document frequency based algorithm to extract primitive concepts to represent documents. His project development has evolved to the sentence and paragraph levels. However, the extracted concepts need a mechanism to analyze their semantics to further extract additional representations and combine them together to form clusters with unique and/or similar semantics. In addition, connecting concepts with similar semantics remains an interesting topic. I recently applied the outsourced knowledgebase Wordnet to implement a concept parser, which extracts n-grams from primitive concepts and several matchers to construct edges between extracted n-grams in the same synsets. The tools are used to extend additional and connect similar representations by analyzing syntax types such as verb and noun and extracting antonymys, hypernyms, hyponyms, and synonyms of the n-grams identified.

In experiment, I extended 24k additional topics from 2000 concepts. 64 percent of the extended topics have over ten million research results found from Google. Besides, I matched 2600 pairs from 2000 concepts with similarity greater than 80% in synonyms

only. To further increase the number of pairs, in chapter 3, I will introduce a quantitative measure for hyponyms and hypernyms to further combine concepts with similar semantics.

1.2 Project architecture

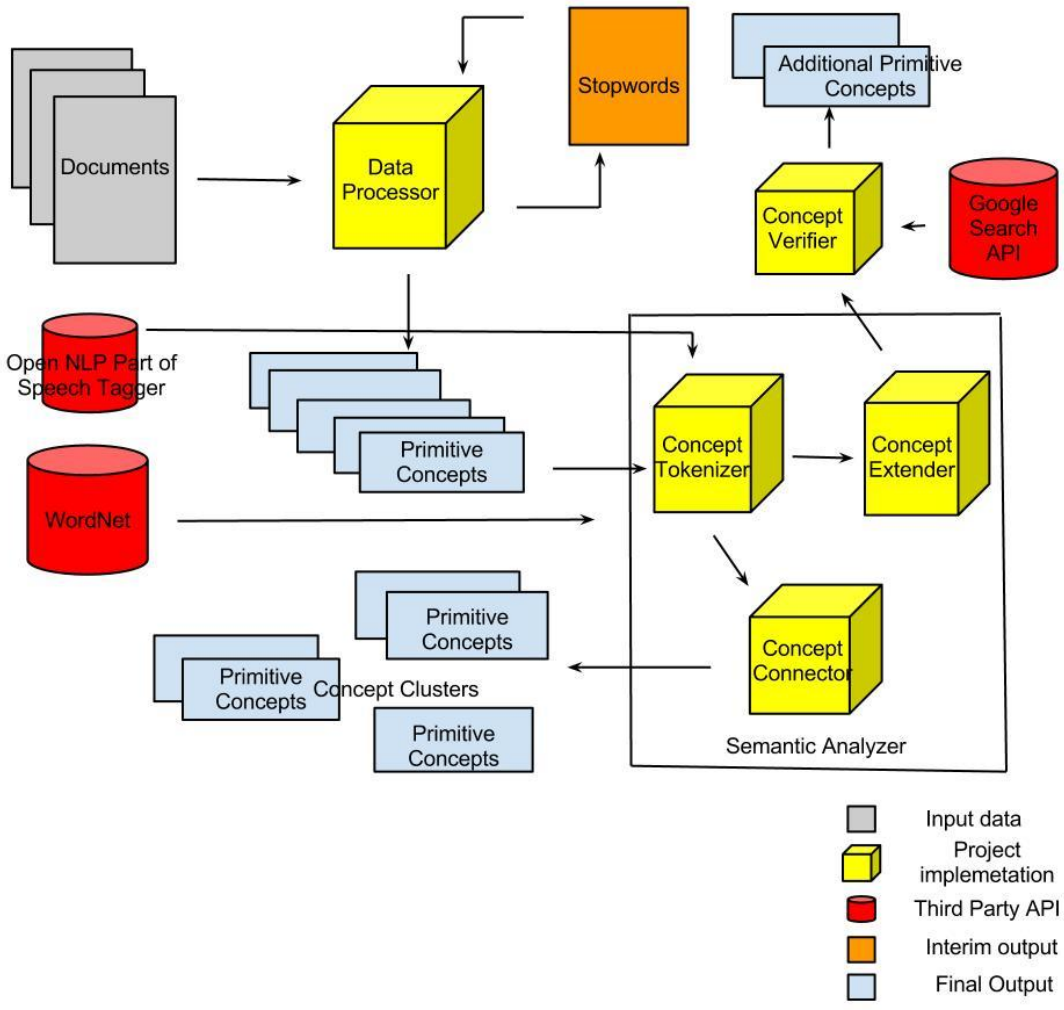

Figure 1. Project architecture

The outline of the project is shown in figure 1. The data processor takes document as input and generates primitive concepts as output. The primitive concepts are then sent to the semantic analyzer that composes of concept tokenizer, concept extender and concept connector. Using Open NLP Part of Speech Tagger and WordNet, the semantic analyzer parses the extracted concepts, tokenizes n-grams, and provides additional concepts with similar meanings using concept extender. The newly found topics are then verified their significance by concept verifier with Google Search API. Lastly, the concept connector forms concept graphs and clusters based on the tokenized concepts generated by the concept tokenizer.

Primitive Concept Extraction

2.1 Introduction to Primitive Concepts

To extract concepts from documents in automatic fashion, Professor Lin introduced a document frequency based algorithm to extract primitive concepts to represent documents. The proposal suggests concepts that should be extracted within finite rolling windows across a given document after filtering stop words. Given a finite rolling window, concepts are defined as any combinations of single words inside the window. In mathematics,

primitive concepts := n**C**r where n <= r, n>0, and r is the length of rolling window

For example if we have a document d = w1 w2 w3 w4 where none of the word w is stop word and assume r = 3, we will have primitive concepts in tuples as follows.

(w1), (w2), (w3), (w1, w2), (w1, w3), (w2, w3), (w3, w4), (w1, w2, w3), and (w2, w3, w4)

In fact, word ordering is important in Linguistics to maintain the precision in lexical meaning in documents. Therefore, we take combinations instead of permutations to keep the importance of word ordering and ignore any cases of reverting orders.

2.2 Stop Word Filtering

Filtering stop word is the preliminary process before extracting primitive concepts. Stop words in general do not contain important lexical meaning, so they should be filtered. In the project, I used TFIDF methodology to filter unimportant words. TFIDF is a score measurement for each word w in document d and defined as the follows.

$$TFIDF(w, d) := tf(w, d) * idf(w, D)$$

where

$tf(w, d)$:= frequency of word w in document d / total number of words in document d,

$idf(w, D)$:= log (total number of documents in corpus D / number of documents in corpus D that contain word w),

and

D is the corpus in interest that contains document d

In the project, the data processor processed and loaded documents to Oracle database into three tables. The tables' schemas are as follows.

```
CREATE TABLE DocWord( docnum NUMBER(7), word VARCHAR2(100), wordpos VARCHAR2(3000))

CREATE TABLE DocTable( docnum NUMBER(7), wordtotal NUMBER(7))

CREATE TABLE WordTable( word VARCHAR2(100), wordcount NUMBER(7))
```

In words, table DocWord contains the positions of word w in each document d. Table DocTable contains the total number of words in each document d, whereas table WordTable contains the number of first appearance of word w in each document in corpus D. The above tables provided all the numbers needed to compute TFIDF of each word in each document. The data processer then loaded the resulted TFIDF measure to the following table.

```
CREATE TABLE WordTFIDF( docnum NUMBER(7), word VARCHAR2(100), TFIDF FLOAT(20))
```

Table WordTFIDF stores the TFIDF score for each word in each document. The scores are used to determine the importance of each word to its document. Words with low score will be discarded and words with high scores will be further sent to extract primitive concepts.

2.3 Algorithm to extract Primitive Concepts

In section 2.1 we mentioned primitive concepts are extracted in sliding windows using combinations. To find all combinations, one can use the algorithm introduced by Rosen.

```
public int[] getNextCombination () {

  if (numLeft.compareTo(BigInteger.ZERO) == 1)
    return null;

  if (numLeft.equals (total)) {
    numLeft = numLeft.subtract (BigInteger.ONE);
    return a;
   }

   int i = r - 1;
   while (a[i] == n - r + i) {
     i--;
   }
   a[i] = a[i] + 1;
   for (int j = i + 1; j < r; j++) {
     a[j] = a[i] + j - i;
   }

   numLeft = numLeft.subtract (BigInteger.ONE);
   return a;

 }
```

,where total is the total number of combinations given n and r. When initiation, numLeft is set as total and a is set as a sequence of ascending integers starting with 0. For instance if n = 5 and r =3, a = [0,1,2] initially. The function above takes the current value of a, returns the next combination, and reiterates until numLeft = 0. The

algorithm increases the values in a from right to left (or large to small in array index) until a = [n-r, n-r+1,…, n-1]. The algorithm determines where in array a is the starting point i to be incremented thereafter. When the algorithm knows the right value of i then all the subsequent values j=i+1,…, r are incremented by j-i. For instance if our input is n=5 and r=3, the algorithm returns [0,1,2] in the first iteration, [0,1,3] in the second iteration, [0,1,4], in the third iteration, [0,2,3] in the fourth iteration,…, [2,3,4] in the tenth iteration, and null thereafter.

Primitive concepts are extracted within sliding windows. Computation will be expensive if we restart to find combinations again when the given window slides to the next one. In fact, one does not need to restart to find combinations if the old word which appears in the given window does not appear in the next window is replaced by the word which does not appear in the given window but appear in the next window. For example if our input is n=5, r=3, and the current window is ranged from 0 to 4, one can replace [0,1,2], [0,1,3], [0,1,4], [0,2,3], [0,2,4], and [0,3,4] by [1,2,5], [1,3,5], [1,4,5], [2,3,5], [2,4,5], and [3,4,5]. The existed combinations from 1 to 4 [1,2,3], [1,2,4], [1,3,4], and [2,3,4] are also included in the new combinations of the next sliding window. Therefore from range 1 to 5, we have [1,2,5], [1,3,5], [1,4,5], [2,3,5], and [2,4,5] (replacement), and [1,2,3], [1,2,4], [1,3,4], and [2,3,4] (copy from the existed combinations). After putting in ascending order, the new combinations are [1,2,3], [1,2,4], [1,2,5], [1,3,4], [1,3,5], [1,4,5], [2,3,5], [2,3,4], [2,4,5], and [3,4,5].

2.4 Inside the Data Processor

The implementation of data processor is composed of four Java classes, Document Word Count Db Loader, TFIDF Db Loader, Primitive Concept Generator and Primitive Concept Combiner. The Document Word Count Db Loader and TFIDF Db Loader are responsible to compute the TFIDF measurement for each word in each

document. The result is loaded to Oracle Database. On the other hand, the Primitive Concept Generator uses the words with high TFIDF score to generate primitive concepts. It then sends the result to Primitive Concept Combiner to further process and combine data into complete concepts. The processed primitive concepts by Primitive Concept Combiner are the final output.

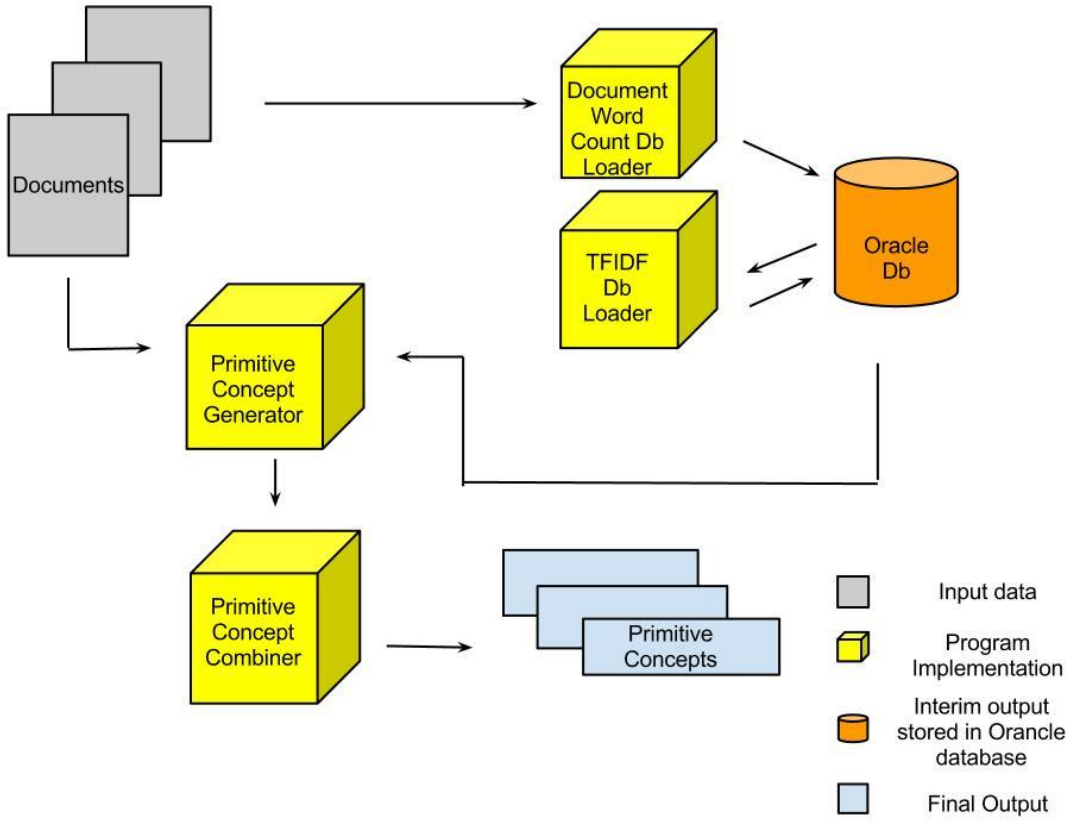

Figure 2. Inside the data processor

2.5 Summary

Context Extraction from documents may sound easy, but is actually hard to implement to yield results with concrete meanings. The method mentioned in chapter

two might not be the optimal answer to some specific corpus but it is a very good solution to any corpus in general in terms of the balance of complexity, accuracy, and resources control.

Analyzing and Extending Additional Representations

3.1 Introduction to WordNet

WordNet is a knowledge database for English dictionary. It groups words with same meaning into synonym sets called synsets. It also provides general definitions and various semantic relations between synonym sets. In text mining, WordNet plays an important role to support automatic text analysis and artificial intelligence applications. In Java, one can provide an n-gram and the type of the n-gram to get the corresponding synsets. For instance,

```
WordNetDatabase database = WordNetDatabase.getFileInstance();
Synset[] synsets = database.getSynsets("neural network",
SynsetType.NOUN);
```

In the example above, the instance of WordNetDatabase gets the corresponding synsets for the n-gram of “neural network” and the type of noun.

As shown in the above example, because n-grams follow different grammatical rules, WordNet distinguishes between nouns, verbs, adjectives and adverbs for input of n-grams. In Java implementation, SynsetType is an enum of NOUN, VERB, ADVERB and ADJECTIVE.

WordNet not only groups words with same meaning but also defines various semantic relations such as antonym, hypernym, and hyponym between synsets. The major semantic relations used in the projects are as follows.

Antonyms: Y is an antonym of X if X is opposite to Y (unhappy is an antonym of happy)

Hypernyms: Y is a hypernym of X if X is a (kind of) Y (engineering is a hypernym of computer science)

Hyponyms: Y is a hyponym of X if Y is a (kind of) X (computer science is a hyponym of engineering)

3.2 Concept Tokenizer and its Algorithms

The concept tokenizer acts a crucial role in the project to automatically identify the optimal combination of given n-grams. For instance, when one parses a concept of " Computer Science Neural Networks branch" to the semantic analyzer, the semantic analyzer should analyze n-grams of "Computer Science", "Neural Networks", and "branch" rather n-grams of 5 single words. In the project, the concept tokenizer continuously fetches information from WordNet, matches n-grams with the fetched information, and computes the optimal combination of parsed n-grams.

The concept tokenizer considers all possible combinations of n-grams parsed from input of primitive concept. It considers the largest n-gram of length of input as well as

input of all individual single words. For example, if the length of input is 3, then the tokenizer computes [3] (largest n-gram), [2,1], [1,2], and [1,1,1] (all individual single words). The algorithm to computes all possibility is as follows.

```
public static HashSet<Combination> getCombinationSet(int size) {

HashSet<Combination>  hs_not_finished = new HashSet<Combination>
();
HashSet<Combination>  hs_finished = new HashSet<Combination> ();
HashSet<Combination>  hs_temp_not_finished = new
HashSet<Combination> ();

hs_not_finished.add(new Combination(size));
LoopingIterator it =  new LoopingIterator(hs_not_finished);
while(it.hasNext()){

        Combination j = (Combination) it.next();
        if (j.getSizeleft() > 0){
                for (int i =1; i<= j.getSizeleft() ;i++){

                        ArrayList<Integer> a = (ArrayList<Integer>)
                        j.getArray().clone();
                        a.add(i);
                        int s = j.getSizeleft()-i;
                        Combination newC = new Combination(s, a);
                        if (s > 0)
                                hs_temp_not_finished.add(newC);
                        else
                                hs_finished.add(newC);
```

```
                }
        }
        hs_not_finished.remove(j);

        if (!it.hasNext()){
                for (Combination k:hs_temp_not_finished){
                        hs_not_finished.add(k);
                        hs_temp_not_finished.remove(k);
                        break;
                }
        }
}
return hs_finished;
}
```

, where the combination class is defined as

```
class Combination{
        ArrayList<Integer> a = new ArrayList<Integer> ();
        int sizeleft;

        Combination(int sl, ArrayList<Integer> al){
                a=al;
                sizeleft =  sl;
        };
        Combination(int sl){
                a= new ArrayList<Integer> ();
                sizeleft =  sl;
        };
        ArrayList<Integer> getArray(){
```

```
            return a;
        }
        void setSizeleft(int sl){
            sizeleft = sl;
        }
        int getSizeleft(){
            return sizeleft;
        }
        public String toString(){
            return "sizeleft=" + sizeleft + " a=" + a.toString();
        }
}
```

The algorithm takes the original size of input as initiation. It progressively discounts combinations by range from 1 to sizeleft until sizeleft = 0. During the progression, it adds discount numbers to array list a.

The tokenizer takes all combinations generated by the algorithm above, fetch n-gram to n-gram and combination to another combination from WordNet to confirm their existence in lexical meaning. Matching ratio is defined as number of existence / number of fetching trials. The program chooses the optimal combination with largest matching ratio and smallest number of fetching trials among tie cases. For instance if ["machine", "learning"] and ["machine learning"] all have matching ratio =1, the latter will be the optimal because its number of fetching trials is the smallest.

Lastly, the tokenizer uses Open NLP Part of Speech tagger to tag the type of optimal n-grams extracted. It also translates antonym by looking for word of contrary such as "not" to combine wording. For instance "not happy" is combined as "unhappy" by the tokenizer.

3.3 Concept Extender using WordNet

The concept extender in the semantic analyzer takes the optimal primitive concept computed by the concept tokenizer to find additional concepts with same synsets. Since the output from concept tokenizer is simply an array of tuples (n-gram, type of synset), the extender uses the tuples, fetch the corresponding synsets from WordNet. The extender then uses the additional synsets from WordNet to form additional primitive concepts. The additional primitive concepts are

[n1 in N1, n2 in N2, …, nm in Nm] – the optimal primitve concept computed by concept tokenizer,

where N1 is the synset fetched using the first tuple t1, N2 is the synset fetched using the second tuple t2,…, and Nm is the synset fetched using the last tuple tm (the array of t1, t2, …, tm is the optimal concept computed by concept tokenizer). Therefore the total number of the additional primitive concepts is N1 x N2 x … x Nm - 1.

3.4 Verification on Extended Topics Using Google Search API

The concept verifier that uses Google Search API further verifies the additional primitive concepts generated by the concept extender. The purpose of this verification is to filter uncommon concepts and keep common concepts. For each concept, I use its

number of search result as its term frequency provided by the web to determine the commonness of the given concept.

I parse each concept as a search query to Google API to fetch the search result in Json format (see the following code). From the fetched Json object, I get the "resultCount" to determine the commonness of the given concept. If the "resultCount" is smaller than a threshold then the concept is uncommon, else the concept is common.

```
URL url = new
URL("http://ajax.googleapis.com/ajax/services/search/web?v=1.0&q=" +
query);
InputStream response = url.openStream();
BufferedReader reader = new BufferedReader(new
InputStreamReader(response));
String line = reader.readLine();
JSONTokener tk = new JSONTokener(line);
JSONObject json = new JSONObject(tk);
String resultCount =
json.getJSONObject("responseData").getJSONObject("cursor").getString("r
esultCount").replaceAll(",", "");
```

Google Search API is a great tool for verifying the usage of primitive concepts in other representations. One may wonder for the primitive concept “social web”, what is the usage of its other representation “social network”? The answer is 368,000,000 search results. One may also wonder for the primitive concept “collaboration networks”, what is the usage of its other representation “coaction networks”? On the contrary, “coaction networks” only have 8910 search results. Obviously, we should keep the common representation “social network” and discard the uncommon representation “coaction

networks". Using search results to filter primitive concept representations increase the accuracy of the final outputs of the project.

3.5 Summary

The concept tokenizer and extender and the concept verifier play important roles to extract concepts in other representations. They not only examine the semantics of primitive concept using WordNet and Open NLP Part of Speech Tagger but also uses Google search API to justify the commonness of the extracted representations. In this way, the tools process primitive concepts and generate final results that are as accurate as possible in an automatic fashion.

Analyzing and Connecting Primitive Concepts

4.1 Connecting Primitive Concepts

In the previous section, we used the data processor and the concept tokenizer to extract primitive concepts and their underlying meanings. One may be interested in knowing the connectivity between the extracted primitive concepts with similar meanings. A simple example is to fetch the synsets underneath the extracted ngrams and use the resulted synsets to match the extracted ngrams in order. Below is the implementation to determine whether two primitive concepts have a common semantic link.

```
public static boolean ifMatch (ArrayList<ngramSynsetType>
ngramSynsetTypeList1,ArrayList<ngramSynsetType>
ngramSynsetTypeList2,WordNet wn, boolean ifConsiderHyperHypo ){
```

```
ArrayList<NgrmaHashcode> ngramHashcode1 =
toNgrmaHashcode(ngramSynsetTypeList1,wn,ifConsiderHyperHypo);
ArrayList<NgrmaHashcode> ngramHashcode2 =
toNgrmaHashcode(ngramSynsetTypeList2,wn,ifConsiderHyperHypo);

if (ngramHashcode1.size() != ngramHashcode2.size()){
        return false;
}

for (int i = 0; i < ngramHashcode1.size();i ++){
        HashSet<Integer> hashCode1 =
ngramHashcode1.get(i).getHashCode();
        HashSet<Integer> hashCode2 =
ngramHashcode2.get(i).getHashCode();
        boolean ifMatchHashCode = false;
        for (int hashCode : hashCode1){
                if (hashCode !=-1){
                        if (hashCode2.contains(hashCode)  ){
                                ifMatchHashCode = true;
                                break;
                        }
                }else{
                        if (hashCode2.contains(-1)){
                                if (ngramHashcode1.get(i).getNgram().
                                equals((ngramHashcode2.get(i).getNgram()))){
                                        ifMatchHashCode = true;
                                        break;
                                        }
                                }
                        }
```

```
            }
            if (ifMatchHashCode == false)
                return false;

        return true;
}
```

The output generated by the concept tokenizer is simply tuples of (n-gram, synset type). The static method ifMatch takes the tuples in the format of ArrayList as input then gets the corresponding synsets (in format of semantic indices), n-gram by n-gram, for each concept. The synsets extracted are also stored in ArrayList to keep the n-grams in order. Then the method iterates both ArrayLists simultaneously to determine whether the corresponding synsets from the ArrayLists have a common synonym for each n-gram pair. If every n-gram pairs find at least one common synonym then the method returns true otherwise false. In addition, the method takes the boolean input ifConsiderHyperHypo to indicate whether hypernyms and hyponyms should be fetched from WordNet. For instance, the code below prints true and then false.

```
String concept1 = "computer science students";
String concept2 = "engineering people";

//consider hypernyms nor hyponyms
System.out.println(ifMatch(tokenizer.getTokens(phrase1),tokenizer.getTok
ens(phrase2),wn, true));

//does not consider hypernyms nor hyponyms
System.out.println(ifMatch(tokenizer.getTokens(phrase1),tokenizer.getTok
ens(phrase2),wn, false));
```

Similarly, given two primitive concepts, each extracted tuple (n-gram, synset type) that has converted to the format of semantic indices can be matched by another tuple that provides the most number of common semantic indices. This forms matched tuple pairs. One can then compute cosine similarity using the matched tuple pairs and unmatched tuple pairs together with the number of each tuple for each given concept.

In the project, there are two similarity matchers each takes two tokenized primitive concepts and WordNet database as input and compute their cosine similarity. They are static methods in the SemanticSimilarityMatcher and SemanticSimilarityMatcher2 classes.

```
SemanticSimilarityMatcher.findSimularity(tokenizer.getTokens(concept1),
tokenizer.getTokens(concept2),wn)
```

and

```
SemanticSimilarityMatcher2.findSimularity(tokenizer.getTokens(concept1
),tokenizer.getTokens(concept2),wn)
```

The SemanticSimilarityMatcher computes similarity described in the second paragraph in section 4.1. Its algorithm in Java as follows.

```
public static double findSimularity(ArrayList<ngramSynsetType>
ngramSynsetTypeList1,ArrayList<ngramSynsetType>
ngramSynsetTypeList2, WordNet wn){
```

```
//input conversion
HashMap<String, Integer> ngramCount1 =
toNgrmaCount(ngramSynsetTypeList1);
HashMap<String, Integer> ngramCount2 =
toNgrmaCount(ngramSynsetTypeList2);
HashMap<String, ArrayList<Integer>> ngramHashcode1 =
toNgrmaHashcode(ngramSynsetTypeList1,wn);
HashMap<String, ArrayList<Integer>> ngramHashcode2 =
toNgrmaHashcode(ngramSynsetTypeList2,wn);
HashMap<Integer, ArrayList<String>> hashcodeNgram1 =
revertNgrmaHashcode(ngramHashcode1);
HashMap<Integer, ArrayList<String>> hashcodeNgram2 =
revertNgrmaHashcode(ngramHashcode2);

//matched map
HashMap<String, Integer> _ngramCount1 = new HashMap<String,
Integer>();
HashMap<String, Integer> _ngramCount2 = new HashMap<String,
Integer>();

for (String  ngram:ngramHashcode1.keySet() ){
  int count = ngramCount1.get(ngram);
  for (int i =0; i < count; i++ ){
    ArrayList<String> matchedNgram = new ArrayList<String>();
    for (Integer hashCode : ngramHashcode1.get(ngram)){
      if (hashcodeNgram2.containsKey(hashCode)){
        if (hashCode != -1){
          matchedNgram.addAll(hashcodeNgram2.get(hashCode));
        }else {
          for (String negNgram :hashcodeNgram2.get(hashCode)){
```

```
            if (negNgram.equals(ngram)){
              matchedNgram.add(negNgram);
            }
          }
        }
      }
    }

    String optimalNgram = getMaxFreqNgram(matchedNgram);

    if (optimalNgram != null){
      ngramCount1.put(ngram, ngramCount1.get(ngram)-1);
      ngramCount2.put(optimalNgram, ngramCount2.get(optimalNgram)-
1);
      if( _ngramCount1.containsKey(ngram)){
        _ngramCount1.put(ngram, _ngramCount1.get(ngram) + 1);
      } else
        _ngramCount1.put(ngram, 1);
      if( _ngramCount2.containsKey(ngram)){
        _ngramCount2.put(ngram, _ngramCount2.get(ngram) + 1);
      } else{
        _ngramCount2.put(ngram, 1);
      }
      if (ngramCount2.get(optimalNgram) ==0 ){
        removeNgram(optimalNgram, hashcodeNgram2);
        ngramHashcode2.remove(optimalNgram);
      }
    }
  }
}
```

```
merge(_ngramCount2, ngramCount2);
merge(_ngramCount1, ngramCount1);

return ComparorUtils.findSim(_ngramCount1, _ngramCount2);
}
```

The algorithm first converts input into few HashMaps. Then, it progressively finds optimal n-gram for each token and updates the existing HashMaps and put matched words into _ngramCount1 and _ngramCount2. Finally, both _ngramCount1 and _ngramCount2 merges with the unmatched n-grams and are used as input to compute cosine similarity.

The SemanticSimilarityMatcher2 uses the same logic in SemanticSimilarityMatcher but extends hypernyms and hyponyms from the unmatched n-grams. When one n-gram finds a suitable concept to match using its hypernym or hyponym, its count is incremented by the reciprocal of the distance between its original position, i.e. 0 and its hypernym or hyponym position plus 1. For instance, if "engineering" is a hypernym of "computer science" in one step, then the number of "engineering" is incremented by 1/ (1 + 1) = 0.5.

4.2 Concept Graph and its Clusters

In the project, the concept connector used the SemanticSimilarityMatcher2 to find graph and clusters of primitive concepts with similar semantics. The connector computes similarity for every pairs and connects the pairs with similarity >= 0.75. With a sample of 250 primitive concepts, the connector constructed a graph of 70 matched concepts with high similarities and found 29 clusters with sizes equal to or greater than

two. The graph is disconnected and undirected. The largest cluster has 20 nodes whereas the smallest ones have 2 nodes. The data visualization of 70 matched concepts is as follows.

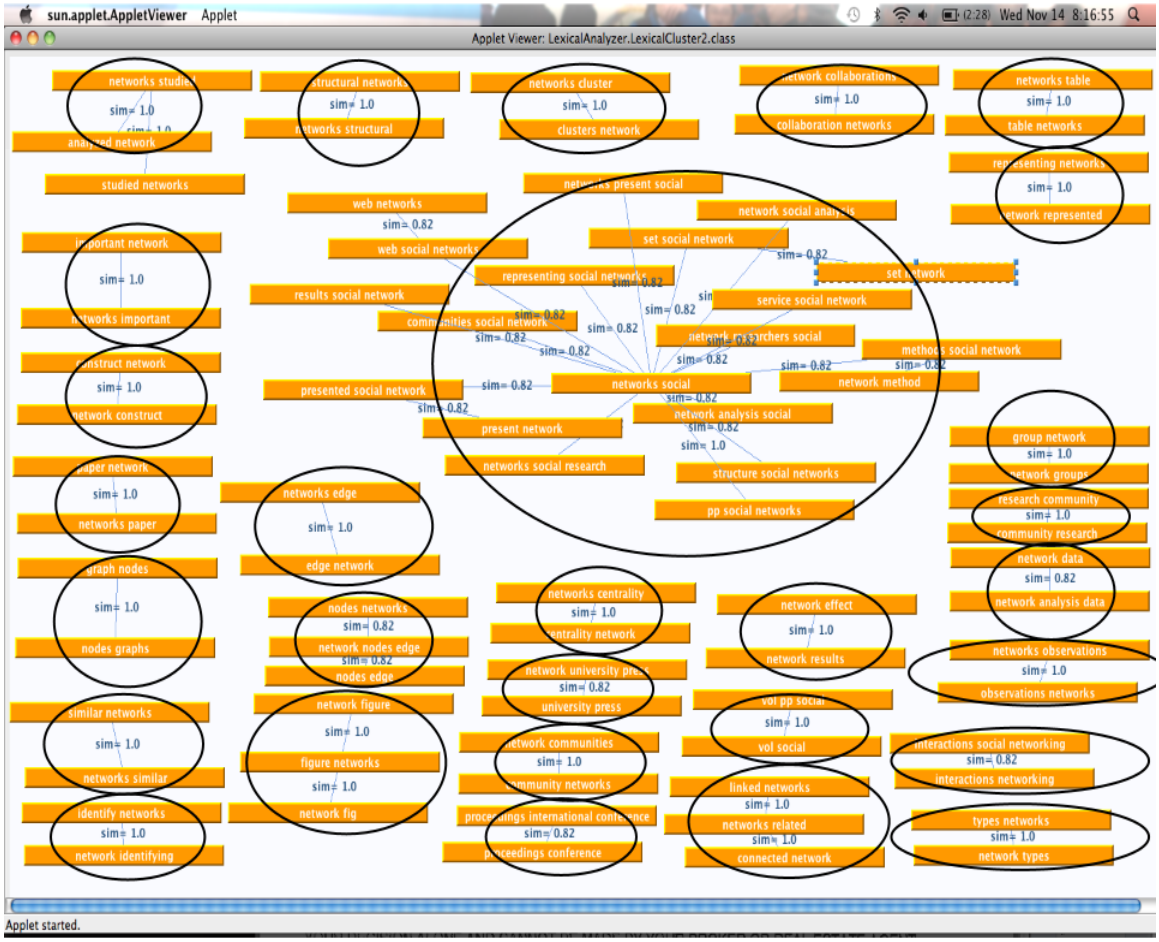

Figure 3. Disconnected and Undirected Graph of primitive concepts

4.3 Summary

The concept connector groups primitive concepts with similar meaning. To group concepts with same (or very close to same meaning), one can increase the threshold to a

value of 1 in the lexical clusterer. In this way, concepts are grouped without the interference of concepts not stemmed or expressed in different representations. Our concept connector allows user to configure his threshold based on his use cases.

Implementation

5.1 Programming Languages used for implementation

Java is the major programming language to implement the project. In the project, I used Java APIs, including Oracle Java Database Connectivity, MySQL Database Connector, Open NLP Tools, WordNet Searching, JGraphT, and JavaScript Object Notation(JSON). Java was chosen to implement the project because it had lots of external packages needed for data processing and text mining.

The Oracle and MySQL database tools are used in the data processor and the lexical extender for TFIDF processing and primitive concept storage. In the concept tokenizer, the Open NLP Tools is used to tag the part of speech of n-grams extracted from primitive concepts. The Java API WordNet Searching is used in the concept tokenizer, the concept extender, and the concept connector. They fetch information from WordNet needed to determine the underlying meaning of given primitive concepts for further tokenization, additional concepts extension, and construct concept graph. Lastly, the JGraphT is used for visualization of concept graph, whereas the concept verifier uses the JSON API to get the count result for each additional primitive concept from the web.

In additional to Java, I also used Ant and XML to build application, and Linux commands and SQL to process and analyze data for section 6.

Empirical results

6.1 Output from Data Processor

The project examined a set of 21 journal papers in social computing, privacy and security. Each paper has around seven pages. The data processor in the project extracted around 2000 primitive concepts. In addition to the primitive concepts, it also computes TFIDF for around 1000 single words for each document. Top 10 single words are as follows.

DOCNUM	WORD	TFIDF
5	mentoring	0.03888713
11	Oss	0.0334261
12	Blog	0.01895713
6	mobile	0.01867141
14	mission	0.0175417
5	mentor	0.01623443
6	Logs	0.01560093
19	Tags	0.01511765
2	Lbd	0.01485421
4	otasizzle	0.01440626

Table 1. Top 10 words with highest TFIDF and their associated document number

Below are some samples of primitive concept extracted from the documents.

SAMPLE PRIMITIVE CONCEPTS
analyzed network
networks studied
connected network
networks related
edge network
networks edge
network nodes edge
important network
graph nodes
linked networks

Table 2. Primitive concept samples

The TFIDF method works great to filter stop words for each document, so primitive concepts are extracted without the interference by the stop words. The primitive concepts extracted carry conceptual meaning. This is done by the method of extraction using combination within sliding windows introduced in section 2.

6.2 Output from Semantic Analyzer

In experiment, I extended 24k additional topics from 2000 concepts. 64% of the extended topics have over ten million research results found from Google. Top and last 10 extended topics are as follows.

REPRESENTATION	COUNT
link web	2530000000
work web	2340000000
link up web	2220000000
make web	1970000000
web number	1920000000
web social	1870000000
place societal web	1840000000
set societal web	1840000000
web back up	1790000000
web place	1720000000

REPRESENTATION	COUNT
mensuration number	45100
web tabular array	42600
network tabular array	34000
futurity worked	26300
ensue execute	20500
mensuration figure	17400
coaction web	9010
coaction network	8930
network colligate	5480
web colligate	3630

Table 3. Top and last 10 extended topics based on number of search results

Using SemanticSimilarityMatcher2, I constructed a graph of 7000 matched pairs from 2000 concepts, mixture of various lengths, provided by my classmate Bieu Do. Setting similarity greater than 80%, the resulted pairs formed 142 clusters. In other words, each cluster contained around 7000/142=49 pairs on average.

In addition, I constructed another six graphs of matched pairs from primitive concepts with similarity greater than 80%. Instead of analyzing on mixture of concepts with various lengths, each graph contains concepts with same length. The resulted statistics are as follows.

Concept length (in number of keywords)	Number of concepts	Number of matched pairs	Number of clusters
1	3935	815	363
2	31048	9746	5340
3	1643	925	118
4	992	383	69
5	794	380	85
6	1194	8566	20

Table 4. Semantic analysis on concepts in length from 1 to 6

The concept verifier using Google API is a great tool to further filter newly generated concepts with low frequency in usage. Learning from the experiment, I suggest setting the threshold to be 10 million to keep the top 64% of the newly generated concepts. On the other hand, using the concept connector, each concept matched around 7000/2000 -1 = 2.5 other concepts on average which did not lead to scenarios of over nor lacking matching concepts. Therefore, I believe setting the threshold of 80% is reasonable to connect concepts into graph and clusters.

Conclusion

Throughout the flow of the project, documents are processed in such a way that original concepts are extracted and kept their original semantics without human interference. Computing TFIDF, fetching search result from Google, and connecting concepts with cosine similarity and WordNet are efficient to reduce noisiness and increase accuracy. This data pipeline uses a set of third party APIs to organize documents available in the Internet and can be further appended using additional third party sources such as Wikipedia and Dmoz. All in all, a perfect or close-to-perfect data analyzer should fully understand and parse data lexically and functions semi-supervised (learn from sources, apply, and reiterate) rather than blindly relying on fancy mathematical clustering algorithms.

List of References

Ken A., James G., David H., 2005. Java Programming Language, Fourth Edition. Prentice Hall.

Tan P., Steinbach M., Kumar V., 2006. Introduction to Data Mining. Addison Wesley

Christiane F., George M., 1998. WordNet: An Electronic Lexical Database (Language, Speech, and Communication). A Bradford Book

George M., Richard B., Christiane F., Derek G., and Katherine M, 1993. Introduction to WordNet: An On-line Lexical Database. International Journal of Lexicography, Volume 3 Issue 4, Pages. 235-244.

Eugene C., 1996. Statistical Language Learning (Language, Speech, and Communication). A Bradford Book

Christiane F., Derek G., and Katherine M., 1993. Adjectives in WordNet. International Journal of Lexicography, Volume 3 Issue 4, Pages 265-277

Christiane F., 1993. English Verbs as a Semantic Net. International Journal of Lexicography ,Volume 3 Issue 4, Pages 278-301

Michael B., Jacob K., 2010. Text Mining: Applications and Theory. Wiley.

George M., 1993. Nouns in WordNet: A Lexical Inheritance System. International Journal of Lexicography, Volume 3 Issue 4, Pages 245-264.