Distributed System Programming

Collocation Extraction

A collocation is a sequence of words or terms that co-occur more often than would be expected by chance. The identification of collocations - such as ‘crystal clear’, ‘cosmetic surgery’, ‘איכות סביבה’ - is essential for many natural language processing and information extraction application.

Prerequisites and Installing

1. Set up an Amazon AWS Account
2. Create a new key pair
3. Create an IAM-emr, IAM-emr-ec2 roles
4. Use Eclipse Java IDE running with 1.6 JDK

Instructions:

The application is composed of a local application and instances running on the Amazon cloud. We will do so creating and exporting .jar files of java classes for all of the steps:1-4 which will run on EC2 instances and AWS class as a jar on local machine.

In order to run the AWS application use the following command:

```
java -jar AWS.jar AWS 0.5 0.2
```

The application will get as an input a sequence file: eng-us-all-2gram and 2 numeric values: min NPMI and relative NPMI. The output will be a directory with all of the 2-grams which satisfies the min and relative NPMI criteria. Those are the collocations.

Input file format

```
lorsquils_etaient 1956 77
lordoscoliosisnoun_verb 1953 69
lorentzianadj_profilenoun 1950 75
lorsquelle_parle 1944 11
lors_comment 1953 21
lorsqueverb_leadj 1957 89
lordmayornoun_ofadp 1950 26
lorriesnoun_duringadp 1904 15
lorx_donne 1976 47
loquat_tree 1940 123
lordlyadv_savage 1901 43
lorga_s 1972 34
lorsquellex_tombex 1907 63
```

Output file format

The assignment includes the following outputs:

For each decade, collocations and there npmi value, ordered by npmi

(descending).

```
1970    lorga_s 1.0
1970    lorx_donne 1.0
1950    lordoscoliosisnoun_verb 1.0
1950    lorentzianadj_profilenoun 1.0
1950    lors_comment 1.0
1950    lorsqueverb_leadj 1.0
1950    lordmayornoun_ofadp 0.99995864
1950    lorsquils_etaient 0.999957555
1940    lorsquelle_parle 1.0
1940    loquat_tree 1.0
1900    lordlyadv_savage 1.0
1900    lorriesnoun_duringadp 1.0
1900    lorsquellex_tombex 0.9885247
```

A list of 10 examples for good and 10 bad examples, we manually collected from the system output.

10 good examples:

```
0.56630026101533l7!long_day
0.6443618471975439!looked_like
0.59927332754771O8!loving_wife
0.6048883454012377!making_money
0.8917541113805639!mississippi_river
0.7387192095797998!new_jersey
0.67576608756563g9!olden_times
```

```
0.7625768940370508!passed_away
0.5773653305244367!south_west
0.5790737939548665!wild_west
```

10 bad examples:

```
0.5690055052776976!famous_lord
0.6129574481667172!general_plan
0.7601509592690138!gives_78
0.618508449653594!good_bread
0.6228676091257599!great_ends
0.6644595966815128!leave_country
0.6630069937263852!little_milk
0.599541774016909!new_courage
0.5581066196190054!old_opinion
0.7141106404645651!page_69
```

As we can tell, the collocations from the bad examples doesn't make so much sense. Furthermore, their npmi value is relatively very low.

Design & Architecture & Implementation

The system is composed of 3 elements:

- Local application(local machine)
- step 1(EC2 node)

- step 2(EC2 node)
- step 3(EC2 node)
- step 4(EC2 node)

The elements will communicate and pass parameters from the main to the mapper/reducer using the “Configuration” object.

Step 1

map:

Removing stop words by ignoring bigrams with one of it’s words is a stop word. The remaining filtered bigrams are sent to step 1 reduce in 3 ways:

1. key: decade#1firstWord value: count!secondWord
2. key: decade#2seconddWord value: count!firstWord
3. key: decade~firstWord_secondWord value: count

Sending it this way allows us to keep track on which word is the first and second and their total count in the corpus. (This information is needed in order to calculate pmi and nmpi of the bigram in the following steps.)

reduce:

Here we combine all the keys and values to the same key and mark the value with the specific kind of count: cw1,cw2,cw12
Using the special charcters markrs that was sent from map “~” “#” “!”

“_” to find out what key we received and seperate the given information. If we recieved “~” we mark the count with cw12, in case we recieved “#1” we mark the count with cw1, and in case we received “#2” we mark the count with cw2.
The information is sent in the following way:
key: decade~firstWord_secondWord value: cw1 counter/cw2 counter/cw12 counter

Step 2

map:

Combining and sending all counters of cw1, cw2 and cw12.

reduce:

In this step we calculate the nmpi according to the given equations using the different counters and send it by decade. Therefore, we set the key to be the current decade and send the the npmi result in the current value.
The information is sent in the following way:
key: decade value: npmi!firstWord_secondWord

Step 3

map:

Combining and sending all counters of cw1, cw2 and cw12.

reduce:

In this step we calculate the nmpi according to the given equations using the different counters and send it by decade. Therefore, we set the key to be the current decade and send the the npmi result in the current value.
The information is sent in the following way:
key: decade value: npmi!firstWord_secondWord

Step 4

map:

In this step we transfer a decade-npmi pair as the key, and a value in the following foramt: collocation!npmi. We use a pairs so we can later use comperable class on it and sort the collocations as neeeded.

NPMICompare:

This class is used to compare between decade-npmi pairs and order the collocations as instructed. It inherits from WritableComparator class and therefore override compareTo function. The comparision is implemnted and executed through using the following line in this current step:

```
job.setGroupingComparatorClass(NPMICompare.class);
```

reduce:

Reuce is used in order to print to a file the final output collocations.

Authors

- **Shir Chen** - *203869698*
- **Adir Ben Azarya** - *203904610*