Design of EMR application for DIRT algorithm

Hanna Hayik 207442054

Overview:

Our goal is to implement an application that can measure the similarity between two sentences according to the DIRT algorithm in the given article.

Using the Biarcs dataset and AWS ElasticMapReduce service we explain the design of our application in this document.

The whole application was made in EMR, apart from a pre-processing of the input beforehand.
My process has 3 components:

- HadoopRunner.java
- StepsRunner.java
- AWS ElasticMapReduce Steps 1-7

The EMR application contains 7 steps, that checks the valid biarcs, constructs paths and perform the necessary calculations that are mentioned in the DIRT article.

Components:

- HadoopRunner
  This .java class is built into a separate .jar that will executed locally on my machine.
  This file is crucial to my application as it initiates a connection to AWS EMR client, configures the application parameters as in which .jar to be executed, naming the application in AWS EMR dashboard, specifying instances types, passing parameters to the .jar to be ran on the cloud.
  This file is also in-charge choosing logs location on S3 and also configuring EMR & Hadoop versions to be used.

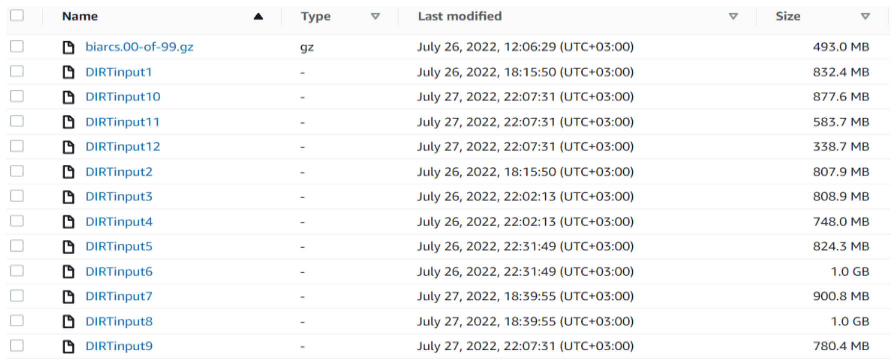
- StepsRunner
  This class is included in the .jar that is uploaded to the s3 (the same .jar that's going to be executed on the cloud).
  We can think of it as the driver class in the cloud.
  StepsRunner.java controls the steps, in terms of configuring every step that's going to be executed, for example: choosing input/output names & locations for every step, also passing output from a step to be another step's input (connecting the steps together).
  This file also specifies the JVM parameters for every step, in addition to choosing Mapper/Reducer/Partitioner and more classes for the steps, more to this class, it chooses the number of reducers to be ran.
  Last thing, this class is our personal dashboard for the running steps, which prints what steps are being configured/running currently, and it can print errors if a step fails (`job.getStatus().getFailureInfo()`).

- EMR Steps 1-7
  These steps are 7 .java files, that contain the code that calculates the needed parameters to the DIRT algorithm.
  Every steps has the basic EMR classes & functions ex: Mapper, Reducer,… also it can have custom functions that are crucial for the application, for example all steps have a printing function which I used for debugging purposes, or Step1 have filtering functions that filters the invalid biarcs from the 1st input.
  Every step's output is the next step's input, this way we can look at the steps as a pipeline.

In the following part I will explain the different steps in my application…

DIRT Implementation:

First of all, we start by locally pre-processing the biarcs we obtained (I used 68 biarc files), and we upload them to S3 to be used as input to our EMR steps. This was done because of technical complications like Java heap space usage in EMR clusters, uploading all biarcs to S3 (takes too long & not practical).



Name	Type	Last modified	Size
biarcs.00-of-99.gz	gz	July 26, 2022, 12:06:29 (UTC+03:00)	493.0 MB
DIRTinput1	-	July 26, 2022, 18:15:50 (UTC+03:00)	832.4 MB
DIRTinput10	-	July 27, 2022, 22:07:31 (UTC+03:00)	877.6 MB
DIRTinput11	-	July 27, 2022, 22:07:31 (UTC+03:00)	583.7 MB
DIRTinput12	-	July 27, 2022, 22:07:31 (UTC+03:00)	338.7 MB
DIRTinput2	-	July 26, 2022, 18:15:50 (UTC+03:00)	807.9 MB
DIRTinput3	-	July 26, 2022, 22:02:13 (UTC+03:00)	808.9 MB
DIRTinput4	-	July 26, 2022, 22:02:13 (UTC+03:00)	748.0 MB
DIRTinput5	-	July 26, 2022, 22:31:49 (UTC+03:00)	824.3 MB
DIRTinput6	-	July 26, 2022, 22:31:49 (UTC+03:00)	1.0 GB
DIRTinput7	-	July 27, 2022, 18:39:55 (UTC+03:00)	900.8 MB
DIRTinput8	-	July 27, 2022, 18:39:55 (UTC+03:00)	1.0 GB
DIRTinput9	-	July 27, 2022, 22:07:31 (UTC+03:00)	780.4 MB

Next, we run **HadoopRunner** to initiate the connection to AWS EMR service, we specify the versions, inputs and all the needed parameters, which by this triggers **StepsRunner** to run its code specifying internal parameters for every job/step in the EMR application.

First 4 steps handle the calculation of the mi(p, slot, w) values.

So now we will explain the Steps[1-7].java:

1. Step1

   This step takes the 12 files of input we created, and filters out biarcs that doesn't meet our conditions, some of these conditions are:
   Head of biarc is a form of verb, biarc has at least two nouns, filtering auxiliary verbs as suggested in the assignment and more.
   Also this step calculates |*,slot,*|.

   Mapper
   Input →
      Key: Line number   Value: original Biarc
   output →
      Key: *, slotX, *   Value: Path, slotX, X-word, |p,slot,w|, X-filler,Y-filler
      Key: *, slotY, *   Value: Path, slotY, Y-word, |p,slot,w|, X-filler, Y-filler

Reducer

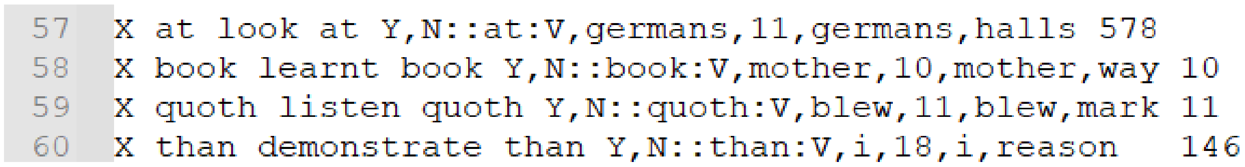
Input →

Key: *, slot, * Value: [(Path1, slotX, X-word, |p,slot,w|, X-word,Y-word), (Path2, slotX, X-word, |p,slot,w|, X-word,Y-word),…]

Output →

Key: Path1, slotX, X-word, |p,slot,w|, X-filler, Y-filler
Path2, slotX, X-word, |p,slot,w|, X-filler, Y-filler
…

Value: |*, slot, *|



```
X at look at Y,N::at:V,germans,11,germans,halls 578
X book learnt book Y,N::book:V,mother,10,mother,way 10
X quoth listen quoth Y,N::quoth:V,blew,11,blew,mark 11
X than demonstrate than Y,N::than:V,i,18,i,reason   146
```

# Map Input Records : 58096766
# Map Output Records: 116192426
Map output bytes=6958042141

Reduce input records=93591211
Reduce output records=62494785

2. Step2

now we have 2 parameters of 4 to calculate the mi() value, in Step2 we actually calculate |p, slot, *| parameter.

Mapper

Input → Key: Line number

Value: Path1, slotX, X-word, |p,slot,w|, X-filler, Y-filler
Path2, slotX, X-word, |p,slot,w|, X-filler, Y-filler
…

Output →

Key: p, slot, *

Value: path, slot, w, |p,slot,w|, X-filler, Y-filler, |*, slot, *|

Reducer

Input →

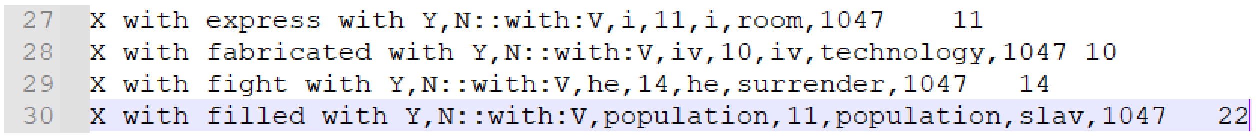
Key: p, slot, *

Value: [(path1, slot, w, |p,slot,w|, X-filler, Y-filler, |*, slot, *|),
(path2, slot, w, |p,slot,w|, X-filler, Y-filler, |*, slot, *|), …]

Output →

Key: path1, slot, w, |p,slot,w|, X-filler, Y-filler, |*, slot, *|

Value: |p, slot, *|



```
X with express with Y,N::with:V,i,11,i,room,1047    11
X with fabricated with Y,N::with:V,iv,10,iv,technology,1047 10
X with fight with Y,N::with:V,he,14,he,surrender,1047   14
X with filled with Y,N::with:V,population,11,population,slav,1047   22
```

# Map input records=62494785

# Map output records=62494785

Map output bytes=5148767273

Reduce input records=23753912

Reduce output records=62494785

3. Step3

This step calculates |*, slot, w| parameter. Very similar to the before steps.

Mapper

Input →

Key: Line Number

Value: path1, slot, w, |p,slot,w|, X-filler, Y-filler, |*, slot, *| |p, slot, *|

Output →

Key: *, slot, w

Value: path1, slot, w, |p,slot,w|, X-filler, Y-filler, |*, slot, *|, |p, slot, *|

Reducer

Input →

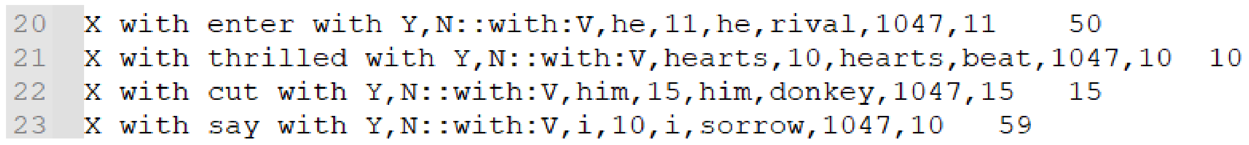
Key: *, slot, w

Value: [(path1, slot, w, |p,slot,w|, X-filler, Y-filler, |*, slot, *|, |p, slot, *|),
path2, slot, w, |p,slot,w|, X-filler, Y-filler, |*, slot, *|, |p, slot, *|,…]

Output →

Key: path1, slot, w, |p,slot,w|, X-filler, Y-filler, |*, slot, *|, |p, slot, *|

Value: |*, slot, w|



```
X with enter with Y,N::with:V,he,11,he,rival,1047,11     50
X with thrilled with Y,N::with:V,hearts,10,hearts,beat,1047,10   10
X with cut with Y,N::with:V,him,15,him,donkey,1047,15    15
X with say with Y,N::with:V,i,10,i,sorrow,1047,10    59
```

# Map input records=62494785

# Map output records=62494785

Map output bytes=5168027877

Reduce input records=36927110

Reduce output records=62399541

4. Step4

Last step related to the mi() value.

If you pay attention to the Step3 reducer output, we can notice that it already has the 4 parameters to calculate the missing mi() value.

Mapper

Input →

Key: Line Number

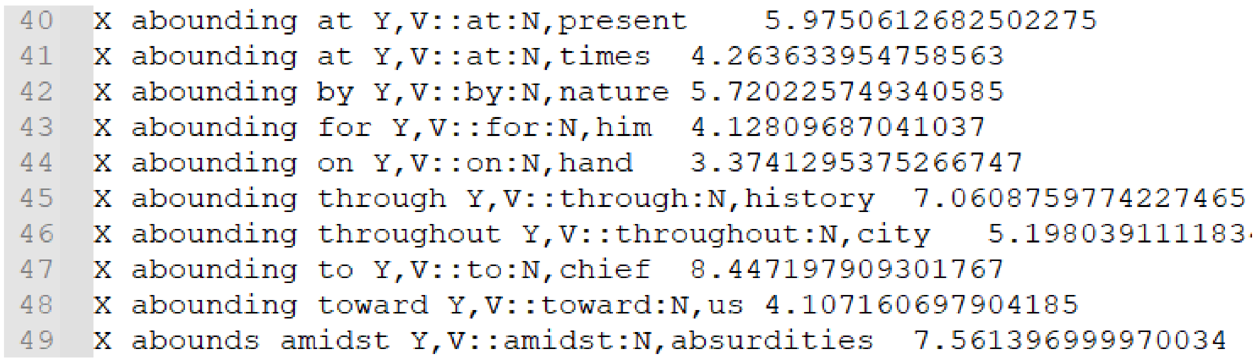
Value: path, slot, w, |p,slot,w|, w1, w2, |*,slot,*|, |p,slot,*| |*, slot, w|

Output →

Key: path, slot, word

Value: mi(path, slot, word)

Reducer → nothing to do there, it serializes the keys & values to the files only



```
X abounding at Y,V::at:N,present    5.9750612682502275
X abounding at Y,V::at:N,times  4.263633954758563
X abounding by Y,V::by:N,nature 5.720225749340585
X abounding for Y,V::for:N,him  4.12809687041037
X abounding on Y,V::on:N,hand   3.3741295375266747
X abounding through Y,V::through:N,history  7.0608759774227465
X abounding throughout Y,V::throughout:N,city   5.198039111183
X abounding to Y,V::to:N,chief  8.447197909301767
X abounding toward Y,V::toward:N,us 4.107160697904185
X abounds amidst Y,V::amidst:N,absurdities  7.561396999970034
```

# Map input records=62494785
# Map output records=62494785
Map output bytes=3044694175

Reduce input records=62494785
Reduce output records=14184690

5. Step5

Next thing we want to have a path as a key and its value is an array of the words that fill X-Y slots with their respective mi() values.
This step gathers every word to its origin path with the needed value.

Mapper

Input →

Key: Line number Value: Path, slot, word mi(p, slot, word)

Output →

Key: Path Value: word, slot, mi(p, slot, word)

Reducer

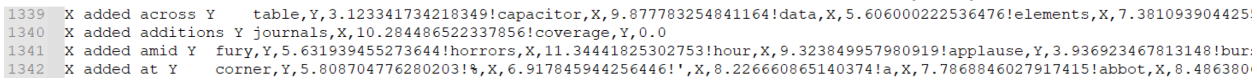
Input →

Key: Path Value:[(word1, slot, mi()), (word2, slot, mi(), …]

Output →

Key: Path Value: String(“word1, slot, mi()!word2, slot, mi(),…”)

* the character “!” was used as delimiter here to differ every triple values



```
X added across Y    table,Y,3.123341734218349!capacitor,X,9.877783254841164!data,X,5.606000222536476!elements,X,7.381093904425
X added additions Y journals,X,10.284486522337856!coverage,Y,0.0
X added amid Y  fury,Y,5.631939455273644!horrors,X,11.34441825302753!hour,X,9.323849957980919!applause,Y,3.936923467813148!bur
X added at Y    corner,Y,5.808704776280203!%,X,6.917845944256446!',X,8.226660865140374!a,X,7.7868846027917415!abbot,X,8.486380
```

```
# Map input records=14184690
# Map output records=14184690
Map output bytes=602822600

Reduce input records=14184690
Reduce output records=339547
```

6. Step6

In this step, we have 2 inputs: the output of step5 and the test set provided in the assignment (positive/negative).
Our mapper handles the 2 kind of inputs by detecting the delimiter and does each kind of input a different processing.
The goal of this step is to mark every two paths that needs comparing by adding a special value to the respective key-value.

Mapper

-Case 1: (Step5 output)

```
 Input →
   Key: Line #        Value: path        word1,slot,mi()!word2,slot,mi()…
 Output →
   Key: path          Value: word1,slot,mi()!word2,slot,mi()!word3,slot,…
```

-Case 2: (positive-preds.txt)

```
 Input →
   Key: Line #        Value: path1       path2
 Output →
   Key: path1         Value: 1@path2
   Key: path2         Value: 2@path1
```

Reducer

```
 Input →
   Key: Path1         Value: ["word1,slot,mi()!word2,slot,…"  1@Path2]
```

Output →
  Key: Path1   Value: word1,slot,mi()!1@Path2!word2,slot,mi(),...

# Map input records=342029
# Map output records=344511
Map output bytes=401376643
* note that from input to output the difference is exactly the same number of lines in the positive-preds.txt file.

Reduce input records=344511
Reduce output records=339659

7. Step7

Final step in our application, the mapper will emit two paths to compare as key, but their value will contain all the words that filled these two paths with the respective slot and mi() value for every word so we can calculate the similarity measure.

Mapper

Input →
  Key: Line #
  Value:path1   word,slot,mi()!1@path2!word2,slot,mi()!word3,...

Output →
  Key: path1@path2   Value: word1,slot,mi(),1
  Key: path1@path2   Value: word2,slot,mi(),1
  Key: path1@path2   Value: word1,slot,mi(),2
  …

Reducer

Input →

Key: path1@path2

Value: [word1,slot,mi(),1,
word2,slot,mi(),1,
word3,slot,mi(),1,
word1,slot,mi(),2,
word2,slot,mi(),2,…

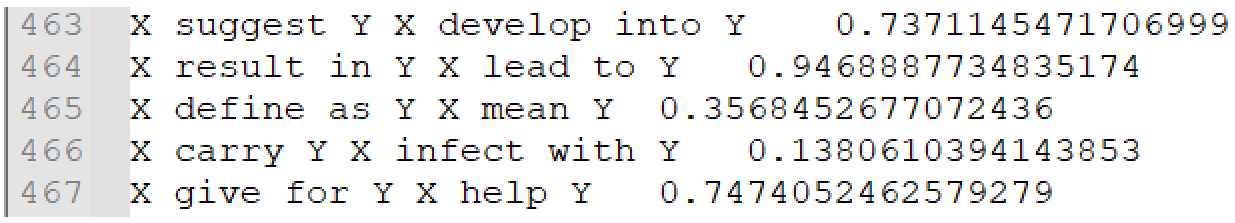
* so for every Key (two paths to compare) we have all the words that fill them with their mi() values which gives us all what we need to calculate the similarity between p1 & p2.

Output →

Key: path1 path2 Value: S(p1, p2)

# Map input records=339659
# Map output records=3646551
Map output bytes=203359647

Reduce input records=3646551
Reduce output records=569



```
X suggest Y X develop into Y    0.7371145471706999
X result in Y X lead to Y   0.9468887734835174
X define as Y X mean Y  0.3568452677072436
X carry Y X infect with Y   0.1380610394143853
X give for Y X help Y   0.7474052462579279
```