Bilkent University

CS425

Project Final Report

Group 17

SIMILYRIC

Emre Gürçay- 21401645

Hüseyin Emre Başar- 21503907

Erdem Adaçal- 21400938

Çağatay Küpeli- 21402290

Table Of Contents

1. Introduction

Music is the source of life. Many artists produce their music to reach to people around the world to tell their feelings. Almost all artists try to be unique while producing their music but in the reality that is not the case. However, when we look at to the lyrics of the music's we do not see much of a difference between different songs. Artists share resemblance in their lyrics. For example, when we look at two different love songs the difference is very less. Also in the music world having similar lyrics creates a huge problem. Recently an artist named Jessi Graham sued Taylor swift for $42 million, for using similar lyrics in her song [1]. Taylor Swift had *"Cause the players gonna play, play, play, play, play/ And the haters gonna hate, hate, hate, hate, hate… And the fakers gonna fake, fake, fake, fake, fake"* chorus in her song while Graham had *"Haters gone hater, playas gone play/ Watch out for them fakers, they'll fake you everyday."* part in his song. They are both very similar. We were affected by this topic and thus chose to implement a project on this topic.
Our aim in this project is to find artist that have nearest duplicate song lyrics.

2. Datasets

In terms of the data, in our project we worked with 3 different datasets in this project. We found 2 different datasets from Kaggle which was provided on the course site. Kaggle is platform where people share datasets for free for community usage. The reason why we used three datasets is we had some problems with first dataset which will be explained in the following sections. The last dataset was crawled from MetroLyrics to satisfy our interests.

2.1 380,000+ Lyrics Dataset

This dataset was found from Kaggle and this dataset is the dataset that we provided in our proposal [2]. It has enough song lyrics. This dataset is ~324MB. After starting our implementation, we found out that this dataset has some non-UTF8 characters and also non-English words, although we wanted to find similar lyrics for English

songs. Also this dataset had some lyrics from unknown artist which makes no sense when thinking the aim of the project. Below is the screenshot of some lyrics found on the dataset and caused program to fail. Failing our attempts with dataset led us to pursue for searching different datasets.

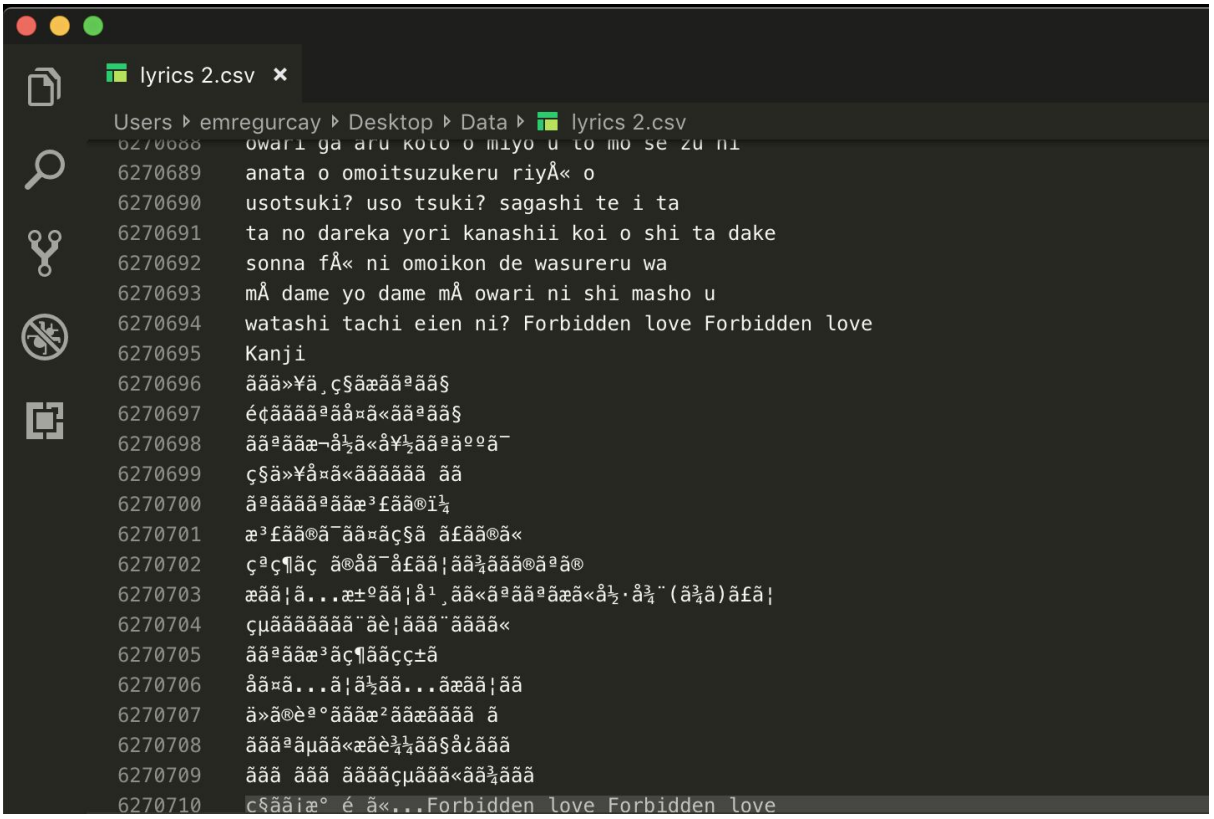


Figure 1. 380,000+ Corrupted Data

2.2 55,000+ Lyrics Dataset

This dataset was found on Kaggle. It contains only English songs found from LycricsFreak [3]. After failing the attempt with the first dataset we looked for other datasets which do not have non-English words or characters. This dataset provided us some reliable data and it contains lyrics from known artists. This is relatively good because it helped us to find some interesting results. However, this dataset is slightly small because it only contains 55,000+ songs. This dataset is ~72MB.

	artist artits	song	link	text
	643 unique values	44824 unique values	57650 unique values	57494 unique values
1	ABBA	Ahe's My Kind Of Girl	/a/abba/ahes+my+kind+of+girl_20598417.html	Look at her face, it's a wonderful face And it means something special to me Look at the way that she smiles when she sees me How lucky can one fellow be? She's just my kind of girl, she ma...
2	ABBA	Andante, Andante	/a/abba/andante+andante_20002708.html	Take it easy with me, please Touch me gently like a summer evening breeze Take your time, make it slow Andante, Andante Just let the feeling grow Make your fingers soft and light Let yo...

Figure 2. LyricsFreak Dataset From Kaggle [3]

2.3 Depth-First Crawler

As it was stated earlier, both 55,000+ and 380,000+ dataset had some problems. 55,000+ data was not large enough to get real results. Likewise, 380,000+ dataset was two years old and included many non-UTF-8 characters in both lyrics and artist name. Therefore, it was essential for us to gather our own data. We looked for many different websites in terms of how their navigation paths works, and the number of lyrics.

Our initial intention was to use either AZLyrics or LyricsFreak; however, we end up using MetroLyrics to gather data. All these website can be access from the following links:

- AZLyrics: https://www.azlyrics.com
- LyricsFreak: https://www.lyricsfreak.com
- MetroLyrics: http://www.metrolyrics.com

AZLyrics had a good navigation path. Unfortunately, it lacked in terms of number of lyrics. It was a small improvement to 55,000+ dataset which we found on Kaggle. LyricsFreak had the reverse problem. It had enough number of data; but, they did not put a navigation path for every lyric and only way to find some lyrics is to use search bar which requires user to know the name of the song or band/artist. Therefore, we decided to use MetroLyrics.

We used BeautifulSoup library to implement crawler. It worked with single thread to decrease the likelihood of ip banning from the website. To compensate this situation, we run this code using multiple ip addresses with different machine at the sametime. There were some problems with navigation path. For some reason, MetroLyrics stated using a new HTML tag for some lyrics. It caused some problem and we need to restart over and over again to realize that was the problem. The following images will display how different these tags are and why they gave us trouble.

Figure 3. Tag Problem During Crawling

As it can be seen from the image one of the <a> elements goes with class name “title hasvidtable” and other one uses only “title”. We did not anticipate this problem while gathering the data; however, one of the team members realized that some lyrics are missing. Then, we were able to realized this problem.

Similarly another problem was that some songs were instrumental songs; but for some reason there are lyrics for these songs which as follows.

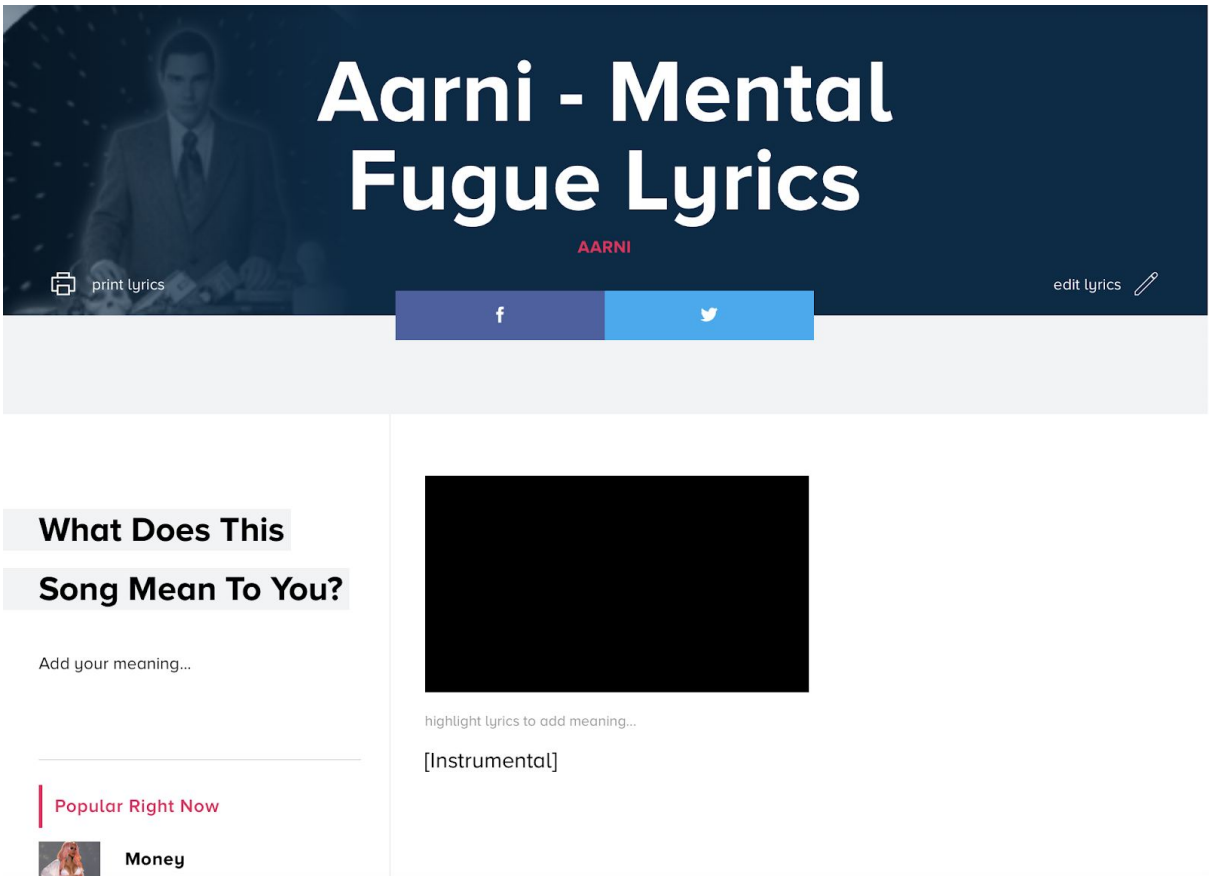


Figure 4. Instrumental Lyric Problem MetroLyrics [4]

We only get lyrics which are longer than 20 characters just to remove these datas from out dataset.

Lastly, we had to remove non-UTF-8 characters and non-english lyrics. We used LangDetect Library to solve both of these issues. As in the name, it detects which language the text is written and provide a percentage [5]. As a result, non-UTF-8 characters and non-english lyrics are eliminated.

We used MongoDB, a NoSQL Database, to store all the data. We were curious about how NoSQL Databases work, and wanted to experience it; however, the main reason is that NoSQL Databases are better for big data projects.

Scraping the data was a success and we gathered 427,999 lyrics; however, this dataset was problematic as much as others which will be discussed under Result for Crawled Dataset section.

```
> db.webscale.count()
427999
> db.webscale.findOne()
{
        "_id" : ObjectId("5c1cecfbd9193f0370d84bac"),
        "artist" : "a",
        "song" : "black hole",
        "text" : "some days go on forever last long into the night some days fee
l like december although its warm on the insideforget the middle of summer i bet
 you i am alive well wipe the floors together nice trysave it all you have all t
he answers thank you for the memories and hope you go with it crying out you hav
e all the anger blame it on yourself again theres nothing wrong with it youre a
black holesome days are complicated some ways ill take the blame some days i tak
e it all to heart again this place is made of panic they check you every mile th
ank god youre going out for a whilesave it all you have all the answers thank yo
u for the memories and hope you go with it crying out you have all the anger bla
me it on yourself again theres nothing wrong with it youre a black holenobody no
ticed what youre talking about nobodys got a clue what youre all about save it a
ll save it all rocksave it all you have all the answers thank you for the memori
es and hope you go with it crying out you have all the anger blame it on yoursel
f again theres nothing wrong with it youre a black holeyoure a black hole youre
a black hole"
}
```

Figure 5. MongoDB Demonstration

3. Implementation Methods

3.1 Preprocessing Dataset

At first while we were implementing the project we first tried our first proposed dataset which is explained in section 2.1. While dividing documents into set of 4 shingles we noticed that our dataset was corrupted. So we learned that it is important to know your data before processing it.

3.2 One-Pass Implementation

In our project we find near duplicate items by looking the size of their intersection in order terms their similarity. Basically we use shingling, minhash and locality sensitive hashing to find the candidate pairs and then for each candidate pair we calculate the cosine distance between two possibly similar documents.

As we have a lot of data, storing all the information needed for the shingling and minhash would require lots of storage space. For example, for 50,000 songs and k = 4 for shingling we require a shingling matrix 26^4 x 50,000 and for each element 4

bytes required so this would require, 456,976 x 50,000 x 4 bytes = 91395200000 bytes which is equivalent to ~ 85 GB.

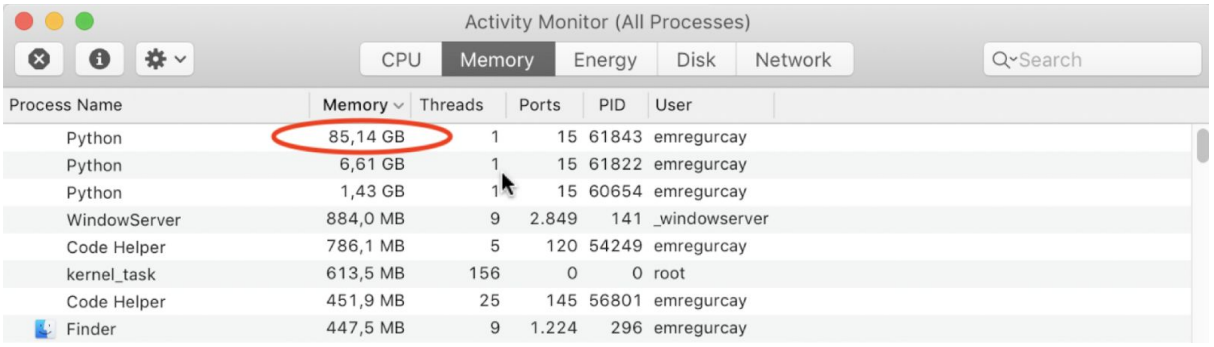


Figure 6. Inefficient Memory Usage

Thus we decided to shingle and minhash document by document and fill the signature matrix respectively by this way for each document we need only ~1,7MB and after filling the signature matrix for this document we do not need this array anymore so deallocate it.

We used k = 4 for shingling as the music lyrics are not very large documents. Below is the representation of the shingling for one document.

	D0	D1	D2	...Dn
	0			
S	1			
h	1			
i	0			
n	0			
g	0			
l	0			
e	0			
S	0			
	1			
	0			

Figure 7. Shingle Array Representation

We have an array which is size 26^4 this is enough to store all possible combinations of 4 shingles in English alphabet. Also we stored 0's but later we noticed that we do not need to store 0's so it will be explained in the section 4.

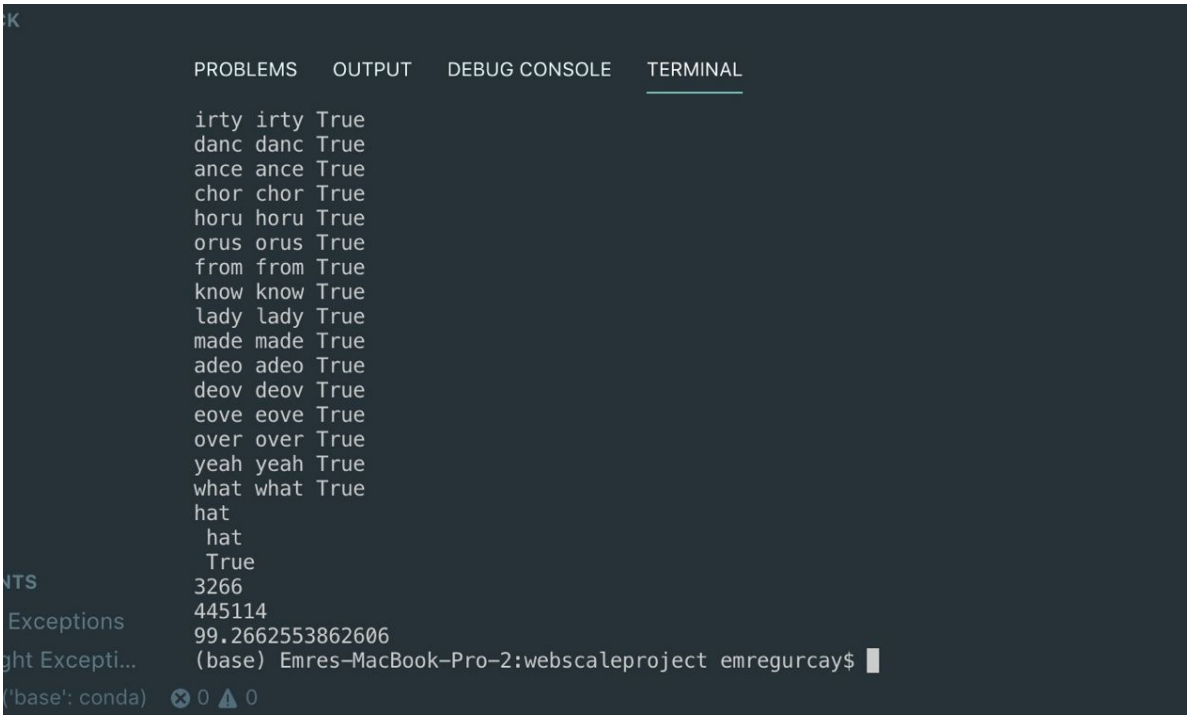


Figure 8. Shingles

This is an example of our k = shingles for one document.
We also implemented a word based shingle method. For each word in document we look for the stop word set and look if this word exists in this set if not hash the word. However, we changed this section to better solution, it will be explained in the improvements section as well.
We also have a regular expression to check the shingles because we did not want punctuations or blanks to exist in our shingles.

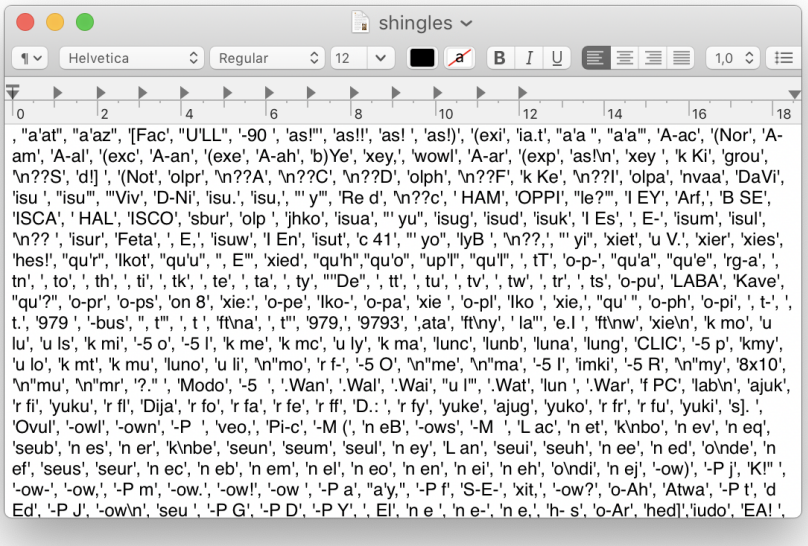


Figure 9. Shingles Before Regular Expression

In order to fill the signature matrix in the beginning of the code, we prepare a hash function set. This is because we shingle each document individually and hash functions are randomly generated. For example, 100 hash functions by using the formula $h_{a,b}(x)=((a \cdot x+b) \ mod \ c)$ we require an array of size 200 and index k is the random number a and index 2k is the random number b. C is a prime number and same for all the hash functions.

324	948	746	…	HASHn

Figure 10. Hash Array

Later we prepared a signature matrix like below

	D0	D1	D2	Dn
HASH0	123	234	345	483
HASH1	425	565	432	567
HASH2	689	467	275	358
HASH3	986	0	725	247
...				
HASHn				

Figure 11. Signature Matrix

In the locality sensitive hashing we divide rows in to bands where each band has 5 rows and hash these into the buckets. If they hash to the same buckets, we represent them as the candidate pairs and compute their cosine distance.

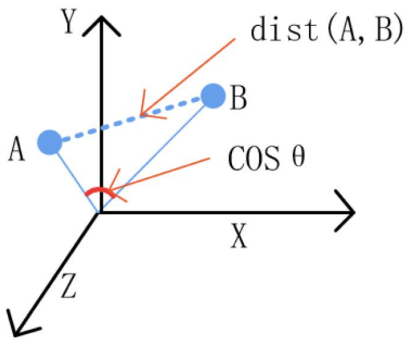


Figure 12. Cosine Distance Representation

We have a threshold of 0.3,

Possibility of D0 and D1 are identical in one band = (0.3)5 = 0,0243

For 200 hash functions and 5 rows in each band;

Possibility of D0 and D1 different in all 40 bands(1 – 0.0243)^40 = 0.3738111

So in total 62.618889 % pairs are similar.

Although this implementation works correctly, it was fairly slow. For the 55,000+ dataset we were able to get results in an hour but for the data we crawled it required much more time. So we used Microsoft Azure to compute it but after 3 days it was still running and we decided shut it down as it was costly.

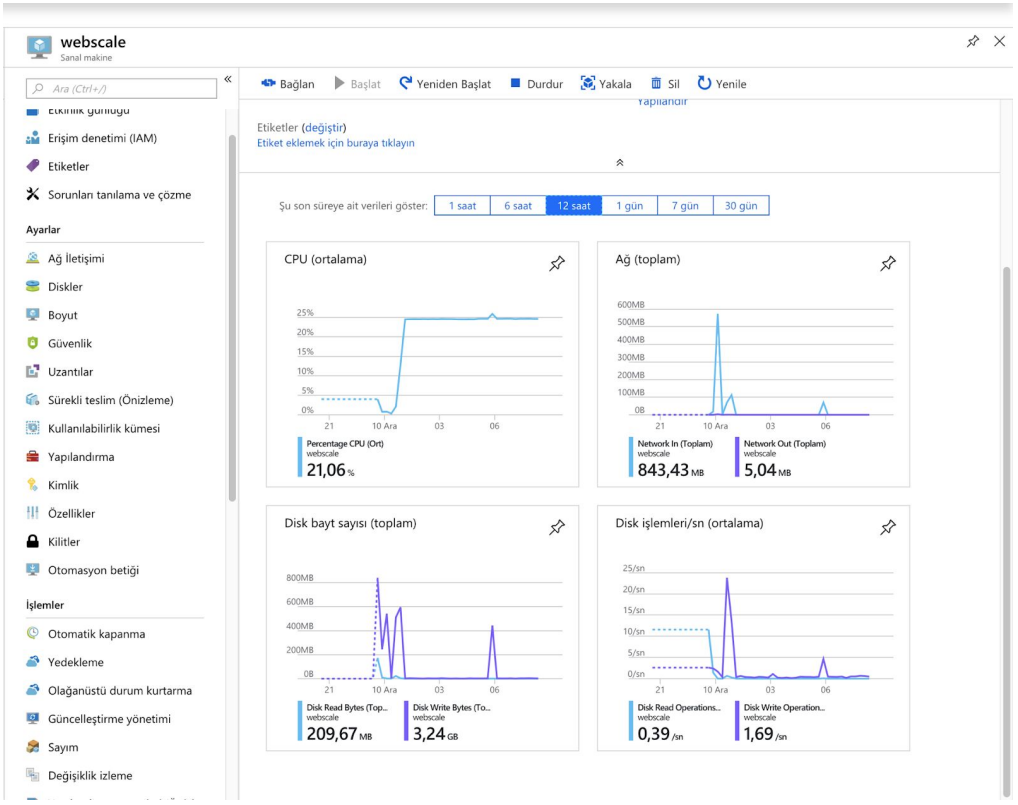


Figure 13. Microsoft Azure CPU Usage While Minhash

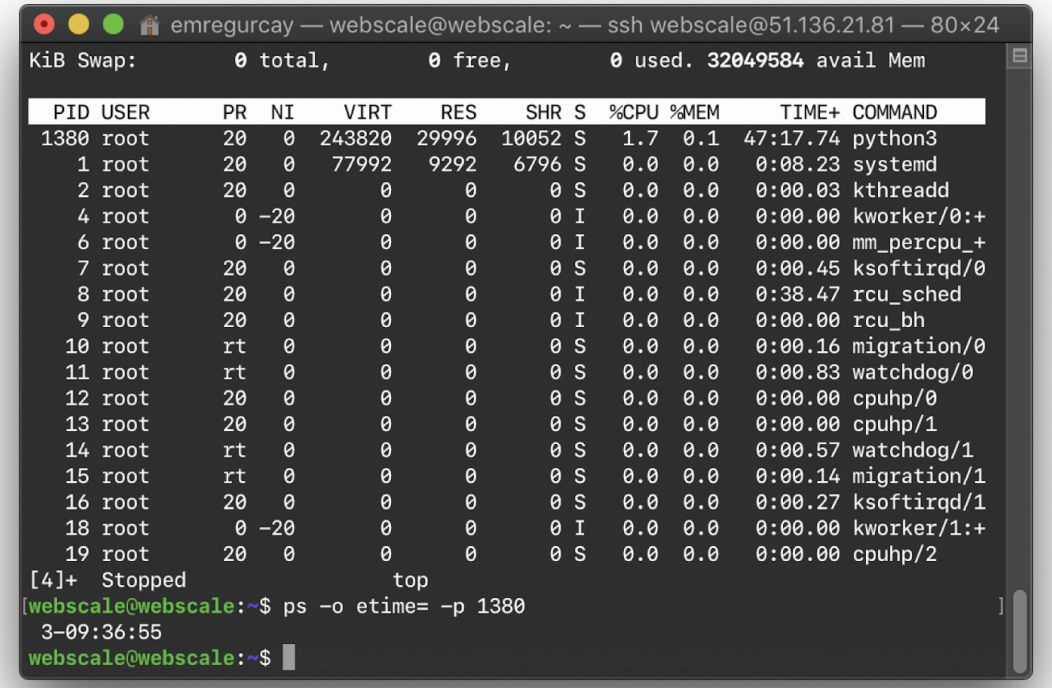


Figure 14. Microsoft Azure SSH Terminal Time Spent on Process

3.3 Map/Reduce Implementations

Our first implementation with our largest dataset were taking days to finish executing. As we are messing up with big data, we thought that we should do a parallel implementation to make our code run fast. After researches, we found that with using Apache Beam, we can implement map reduce. Apache Beam is an open source, unified model for defining both batch and streaming data-parallel processing pipelines [14]. On apache beam, data is collected as PCollections and these PCollections can be both input or output. Data is transmitted through pipelines.

3.3.1 Algorithm

Pardo:

- Input : A line of our dataset.txt file
- Output: < key = Song Name, value = shingle of lyrics>

Explanation: **DoFn** is a **Beam** SDK class that defines a distributed processing function. By passing DoFn argument to ParDo function, our program starts to run parallel. As it can be seen from the screenshot, it processes 1527 element per second, where element means each line of our txt file. Each line on our txt file represents a different song. Data format is like Artist\Song Name\ Lyrics.

MAP1:

- Input: <key = Song Name, Value = Shingle of lyrics >
- Output: < key = bucket of one band, Value= Song Name>

Algorithm:

-create SignatureMatrix

-for each shingle

-apply hash function and fill signature array

for each band in SignatureMatrix

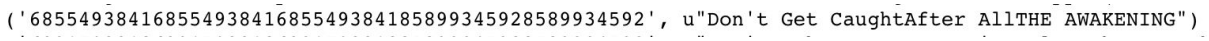
-bucketNo ← hash(band)

-generate << key = bucketNo value = song name>>

Explanation: It takes shingle and songs name, apply minhash and lsh and then generates buckets with containing a songs bands.

Reduce:

After reducing we collect all songs which fall into same bucket. As it can be seen from screen shot, bucketNo starting with 65855 has 3 songs which are Don't Get Caught, After All and The Awakening. But these are not whole song, they are just their one band.



Filter Buckets:

As we are interested in finding near duplicates on our dataset, if a bucket contains only one song, we should ignore it so we use apache beam filter. After filtering, buckets which contain 2 or more songs remain.

Example Output After Filtering:

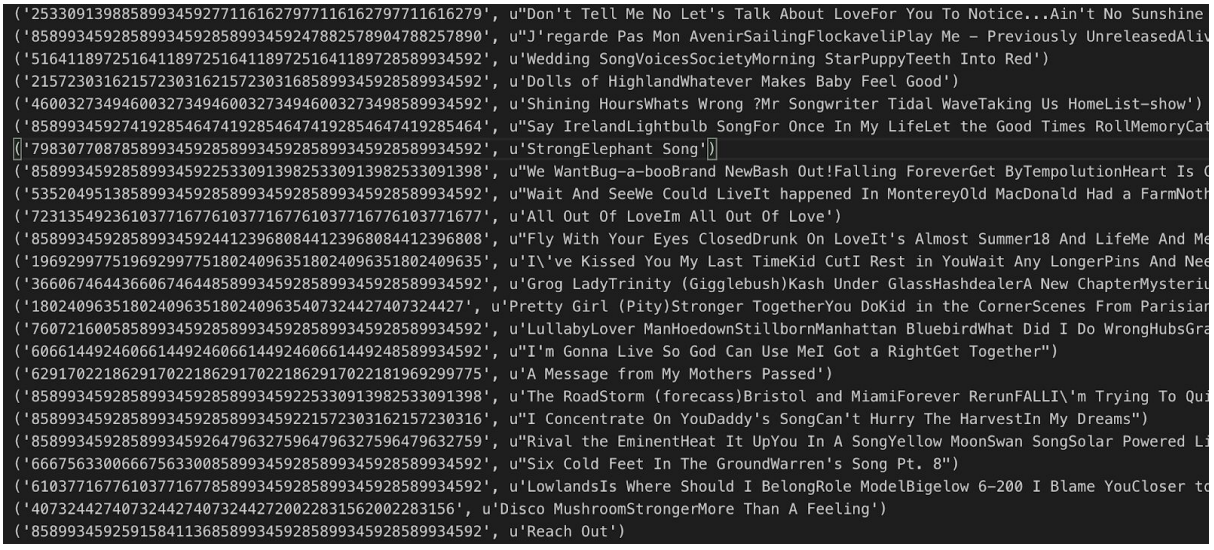


Figure 15. Example Output After Filtering

MAP 2:

- Input: <<key = bucket of one band, Value = Song Name >>
- Output: << key = song1, song2, Value = 1 >>

Algorithm:

For every subset in combination of 2 songs in bucket

generate << key = subset[0], subset[1], Value = 1 >>

Explanation: As there are songs bands in buckets, we select 2 of them, as they are in same bucket, we map them with 1. This means they fell into same bucket once.

Reduce:

After reducing, we obtain an output like <<key = song1, song2, value = [1,1,1,1,1] >> etc.

Example Output After Map2:

```
('IFoundLove(w/VinceGillandRosanneCash) IFoundLove', '11111111111')
("Eveyone'sHomeforChristmas Everyone'sHomeForChristmas", '1111111111111111111')
('HappyBirthday(Malay) HappyBirthday(French)', '1111111111111111111')
('HappyBirthday(Romanian) HappyBirthday(Danish)', '1111111111111111111')
('Satellite/StealingTime Satellite/StealingTime(Acoustic)', '1111111111111111111')
('St.LouisBlues St.LouisBlues', '1111111111111111111')
('HappyBirthday(Tongan) HappyBirthday', '1111111111111111111')
('Insomnia2.0 Insomnia2.0[AviciiRemix[RadioEdit]]', '111')
('AsTheDeer AsTheDeer', '11111111111111111111')
("Fireworks PleaseDon'tHurryYourHeart", '1')
('ImagineMeWithoutYou ImagineWithoutYou', '1111111111111111111')
('ThePope Secret', '11')
("Lost ChapterV:Lost(robin'sSong)", '1111')
('Megawacko2.1 Megawacko2.0', '1')
('GloriousAgainTheNorthlandShallBecome GloriousAgainTheNorthlandShallBecome', '1111111111111111111')
('CorruptingMorality CorruptionMorality', '1111111111111111111')
('LeaveMeAlone LeaveMeAlone[interlude]', '1')
('Hurricane Hurricane(ArtyRemix)', '1')
('Spaceman Spaceman(radioEdit)', '11')
('HappyBirthday(Tongan) HappyBirthday(Czech)', '1111111111111111111')
```

Figure 16. Example Output After Map2

Calculating Similarity Percentage:

- Input: < key = song1, song2, Value = 1,1,1,1,1 >
- Output <key = song1, song2, Value = “%x” >

Algorithm:

For occurences in Value

sum = sum + value

return key, sum

Explanation: After our last reduce step, we obtain how many times songs at key occur in same bucket. We sum these 1’s and make percentage calculation by multiplying it with 100 and dividing it to number of bands. For example if both songs bands fell into same bucket for 4 time, this means that 4/20, they are same so their similarity is %25. We could do this step on reduce but on apache beam, we cannot modify GroupByKey operation so we had to add this step to find percentage.

Format Results:

As we process our data in form of PCollection, before writing, we have to change encoding of characters to utf-8. In order to do that, we connected output to another pipeline which is called format results. Function basically changes all outputs to utf-8 format.

Writing To Text:

Lastly, we use apache beams WriteToText function to write our results on txt file.

Example Output after Writing To Text

```
('IFoundLove(w/VinceGillandRosanneCash) IFoundLove', 55.00000000000001)
("Eveyone'sHomeforChristmas Everyone'sHomeForChristmas", 100.0)
('HappyBirthday(Malay) HappyBirthday(French)', 100.0)
('HappyBirthday(Romanian) HappyBirthday(Danish)', 100.0)
('Satellite/StealingTime Satellite/StealingTime(Acoustic)', 100.0)
('St.LouisBlues St.LouisBlues', 100.0)
('HappyBirthday(Tongan) HappyBirthday', 100.0)
('Insomnia2.0 Insomnia2.0[AviciiRemix[RadioEdit]]', 25.0)
('AsTheDeer AsTheDeer', 100.0)
('ImagineMeWithoutYou ImagineWithoutYou', 100.0)
('ThePope Secret', 15.0)
("Lost ChapterV:Lost(robin'sSong)", 10.0)
('GloriousAgainTheNorthlandShallBecome GloriousAgainTheNorthlandShallBecome', 100.0)
('CorruptingMorality CorruptionMorality', 100.0)
('HappyBirthday(Tongan) HappyBirthday(Czech)', 100.0)
```

Figure 17. Example Output

This is just an example output to show how outputs will be seen. Left column shows a pair of 2 songs and right column shows the similarity percentage.

Pipeline Structure:

Figure 18. Example Pipeline Structure

Example Steps Of Map Reduce Implementation:

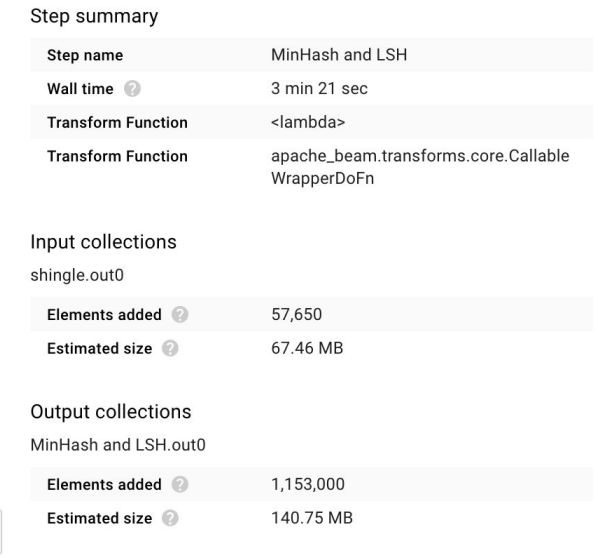


Figure 19. Example Map Reduce Step

Above figure shows that 57.650 element is pipelined to MinHash and LSH function, and after all songs are processed, it shows that there are 1.153.000 buckets.

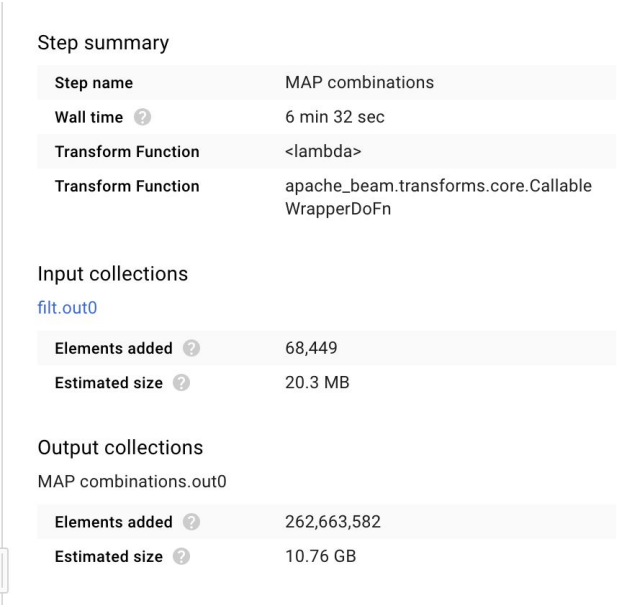


Figure 20. Example Map Reduce Step After Filtering

Above figure shows that after filtering, there are 68.449 buckets are remained, containing more than 1 song. Then it starts to calculate combinations on buckets. For 57k data there are 262.663.582 combinations found.

4. Improvements

There are two different implementation of our project. We named them as follows: “one-pass implementation” and “map implementation”. “one-pass implementation” uses the implementation thought in course, and “map implementation” uses Map-Reduce and combination to find similarity.

4.1 One-Pass Implementation

4.1.1 Removing 0’s From Shingle Array

We implemented this project just as it was introduced in the course; however, during presentation, Dr. Ozdal pointed out that we do not need to store 0’s in shingle array. He stated that this will improve the performance of the program.
As it was stated earlier, we were using Python for this project. Shingle array was actually a list. We convertes it into a set. This way, we can only store the required data and the implementation was more easy to understand. During our last CS 473 (Algorithms I), we were introduced with sets and in our lecture, Dr. Cevdet Aykanat, stated that sets are better if you want to determine an item exist in that container. Therefore we adjusted shingling and minhashing part. The performance increase was noticeable.

```
Cagatays-MacBook-Pro:Desktop cagatay$ time python3 webscale_no_map.py
Initialized permutation array
All the required preparations has been completed!
Shingling and Minhashing has finished!

real    53m25.791s
user    50m24.247s
sys     2m19.092s
```

Figure 21. Previous Iteration with 55,000+

```
Cagatays-MacBook-Pro:Desktop cagatay$ time python3 webscale_no_map.py
All the required preparations has been completed!
Shingling and Minhashing has finished!

real    5m22.596s
user    5m12.981s
sys     0m8.101s
Cagatays-MacBook-Pro:Desktop cagatay$
```

Figure 22. Current Iteration with 55,000+

The figure above is modified to finish after minhashing over. We used 55,000+ dataset for this performance test. The results gets better with number of entries in the dataset.

4.1.2 Finding Word Counts and Preprocessing Data According to Results

We were using a stop word list which allow us to not include words like ‘I’, “you”, “be”, ect. While we were gathering information about content-based similarity finding, we realized that we can count the number of words in each song. Then use this data to improve stop word array according to scraped dataset. We put all the words that are meaningless without another word. For example, “love” is 28th word and “like” is 24th word that is used mostly; however, they are not in stop word list because it has a meaning alone which aims to help us to understand how similar each lyric in terms of word base implementation.

Word count list is as follows:

- 'the' : 4042221
- ‘i’ : 3383417
- you : 3335446
- 'to' : 2318843
- 'and' : 2194928
- 'a' : 1956830
- 'me' : 1558520
- 'my' : 1462836
- 'in' : 1330157
- 'it' : 1287427
- 'of' : 1115889
- 'your' : 1008895

- 'that' : 953032
- 'on' : 910286
- 'im' : 899549
- 'is' : 781809
- 'all' : 770503
- 'for' : 732377
- 'be' : 720687
- 'we' : 708362
- 'dont' : 651287
- 'know' : 594944
- 'so' : 591408
- 'like' : 588477
- 'with' : 580976
- 'its' : 573829
- 'but' : 573308
- 'love' : 565193

4.1.3 Changing List Definitions

The following stack overflow topic covers shows that even list definition has an impact on how fast program works. It was concluded integer multiplication method is best one in terms of speed [6].

4.2 Map Implementation

4.2.1 Running on Google Cloud

On apache beam, there are 2 running options. First one is called DirectRunner, which allows you to run your code on local machine. Direct runner executes pipelines on local machine. It does not focus on efficiency. As we could not gain any performance on DirectRunner, we've decided to run it with DataflowRunner on Cloud. Using Google Cloud, we've changed our code to run it with DataflowRunner. As we have free trial account, it gives us 8 instance quota for one dataflow job so when we run our code on cloud, we've created 8 compute engine instance. As it can

be seen from Figure 24 and Figure 25, our job tries to scale worker count to 25 but since we have 8 quota available, it works with 8 workers.

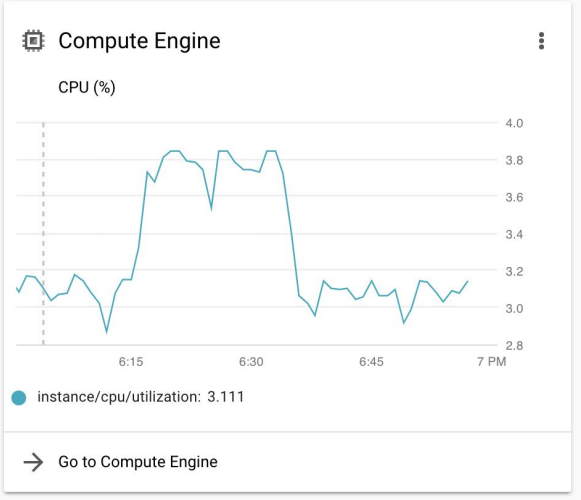


Figure 23. CPU

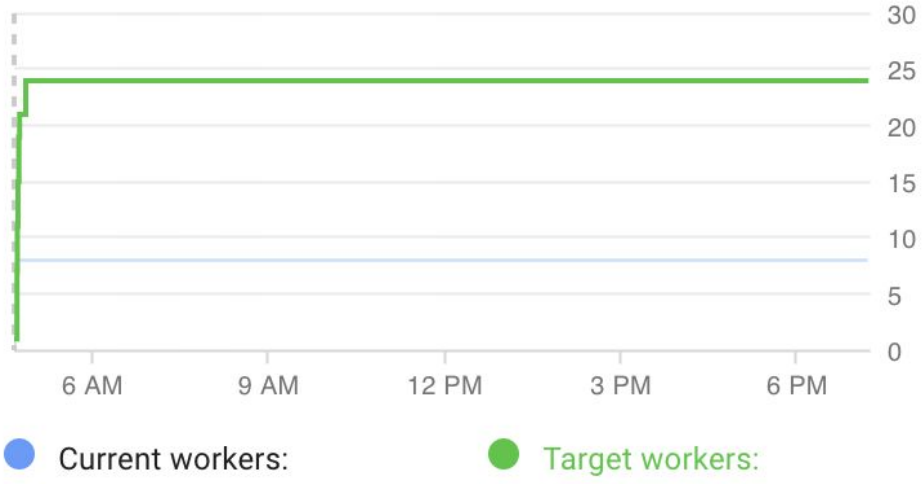
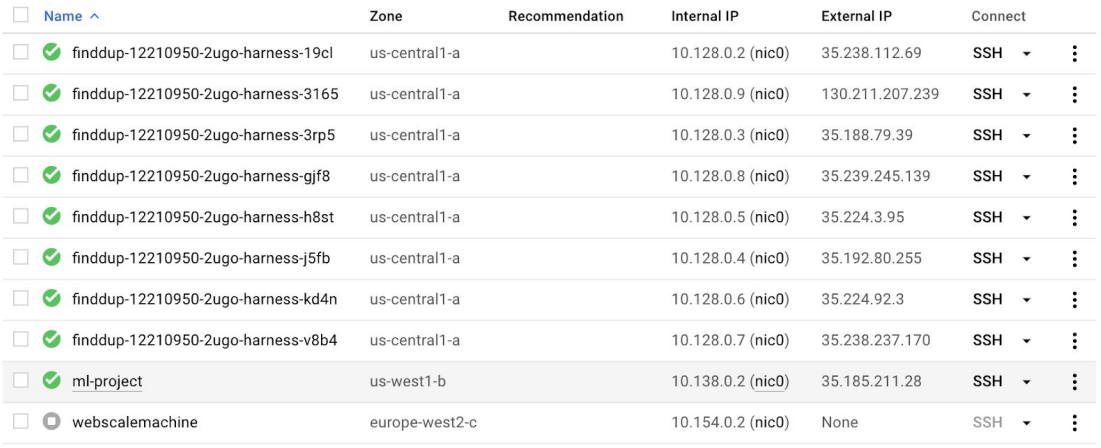


Figure 24. Worker Count



Name	Zone	Recommendation	Internal IP	External IP	Connect
finddup-12210950-2ugo-harness-19cl	us-central1-a		10.128.0.2 (nic0)	35.238.112.69	SSH
finddup-12210950-2ugo-harness-3165	us-central1-a		10.128.0.9 (nic0)	130.211.207.239	SSH
finddup-12210950-2ugo-harness-3rp5	us-central1-a		10.128.0.3 (nic0)	35.188.79.39	SSH
finddup-12210950-2ugo-harness-gjf8	us-central1-a		10.128.0.8 (nic0)	35.239.245.139	SSH
finddup-12210950-2ugo-harness-h8st	us-central1-a		10.128.0.5 (nic0)	35.224.3.95	SSH
finddup-12210950-2ugo-harness-j5fb	us-central1-a		10.128.0.4 (nic0)	35.192.80.255	SSH
finddup-12210950-2ugo-harness-kd4n	us-central1-a		10.128.0.6 (nic0)	35.224.92.3	SSH
finddup-12210950-2ugo-harness-v8b4	us-central1-a		10.128.0.7 (nic0)	35.238.237.170	SSH
ml-project	us-west1-b		10.138.0.2 (nic0)	35.185.211.28	SSH
webscalemachine	europe-west2-c		10.154.0.2 (nic0)	None	SSH

Figure 25. Google Cloud VM Instances

4.2.2 Word Count

On the previous sections, we mentioned about how we found word counts and preprocessed our data. This brought an another example to our minds. We thought that we can use Map Reduce implementation to find word counts faster in our dataset which consists of 427.999 songs, but it did not execute faster. But if dataset had more lyrics which means more words, it will definitely be a faster implementation.

5. Results

We have tried our project with different implementations, parameters and two different datasets.

5.1 Results for 55,000+ Lyrics Dataset

Before improvements made in our code for this dataset it was running in 54 minutes and 34 seconds, after the improvements for word based shingles it was reduced to 16 minutes and 28 seconds. This is a huge improvement because code is running almost 3 times faster compared to first implementation with 200 hash functions.
When we look at the results we see that Hillsong and Hillsong United have nearest duplicate lyrics. This is not a shocking result because when we searched on the web

we learned that they are the same group but they share their albums and songs under different names. Also many people on the web searched about this similarity.

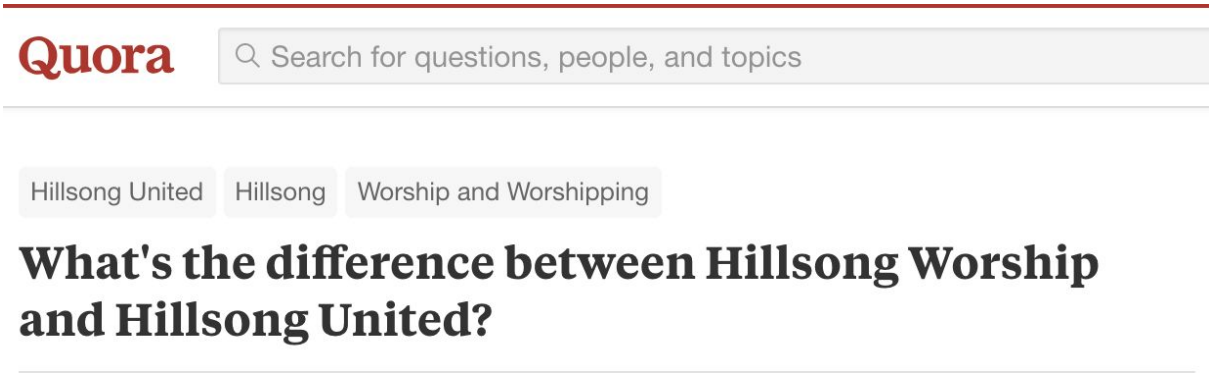


Figure 26. A Discussion about Hillsong Name Confusion [7]

For example, Paul McCartney is the songwriter of The Beatles so it is expected for them to share the same songs. But Paul McCartney have additional songs as well. Also in the results it was seen Irving Berlin and Ella Fitzgerald have same songs although they do not share the same band. We searched for this result and we found out that, in 1958 Ella Fitzgerald performed and album which only focuses on the songs Irving Berlin. It is possible to find the album under the *Ella Fitzgerald Sings the Irving Berlin Song Book* name [8].

Figure 27. Album Representation [9]

When we look the other results they all high percentages because they were in the same band or they sang the same songs. For instance, Rod Stewart is the singer of Faces, and Ozzy Osbourne is the singer of Black Sabbath.

Those results mean that our dataset have the same songs for different artist. The reasons why it does find %100 similarity for the same songs are randomized hash functions and also same songs are written in different forms. They have small differences like *going* and *goin'* or for example *in the river* and *in a river.* But at the end we find the similarity which are greater than the threshold.

When we run the code with k = 4 based shingles the required time increases about 3 minutes in the improved version this is due to the shingle count with k = 4 there more shingles than the word based shingles.

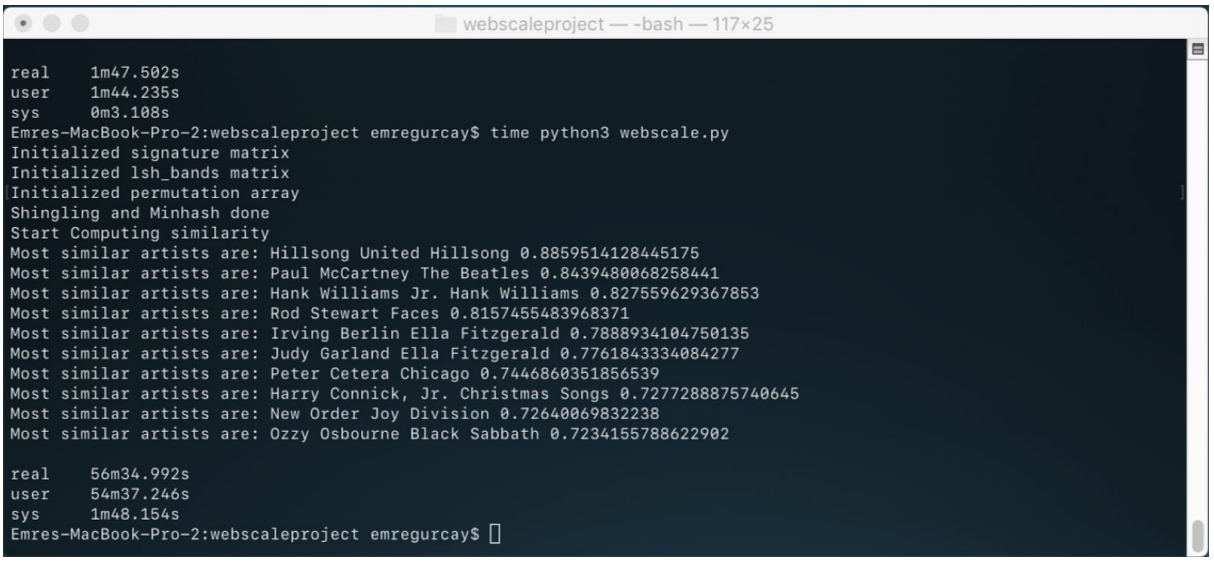


Figure 28. First Iteration

```
Most similar artists are  Hillsong United Hillsong 0.913517445072393
Most similar artists are  Wham! George Michael 0.8799263016011818
Most similar artists are  Hank Williams Hank Williams Jr. 0.7816750042365183
Most similar artists are  Michael Buble Westlife 0.7718086167452
Most similar artists are  Irving Berlin Ella Fitzgerald 0.7627695645891394
Most similar artists are  Paul McCartney The Beatles 0.7604434866501687
Most similar artists are  Glee The Beatles 0.7454172506216
Most similar artists are  Chuck Berry Rolling Stones 0.7166540311807447
Most similar artists are  Frank Sinatra Ella Fitzgerald 0.7115119355725542
Most similar artists are  Ozzy Osbourne Black Sabbath 0.7098897890356034
Most similar artists are  Ella Fitzgerald Frank Sinatra 0.7021095387116235
Most similar artists are  Bing Crosby Perry Como 0.6858499771557621
Most similar artists are  Frank Sinatra Perry Como 0.676804705871918
Most similar artists are  Nat King Cole Frank Sinatra 0.6574426248939429
Most similar artists are  Judy Garland Irving Berlin 0.633277134255514
Most similar artists are  George Michael Stevie Wonder 0.6325438760551781
Most similar artists are  Luther Vandross Diana Ross 0.6316401919882447
Most similar artists are  Diana Ross The Beatles 0.6226508056230562
Most similar artists are  Harry Connick, Jr. Barbra Streisand 0.6130666168584666
Most similar artists are  Michael Buble Diana Ross 0.6120776970079067

real    16m28.845s
user    16m26.260s
sys     0m2.600s
```

Figure 29. Improved Version

5.2 Results for Crawled Dataset

For the crawled dataset we were unable get the results in the normal version. We terminated the cloud computation after 3 days due to the high cost. However, in the improved version it required again much time but this time we were able get the results. We ran the code on the local and it took 502 minutes 49 seconds for the char based k = 4 and for the word based it took 417 minutes and 57 seconds. Both of them have 200 hash functions when computing the signature matrix.

For the results of this dataset we were a bit unsatisfied because the site that crawled our data from have artist two times. For instance, when we look at the figure XX we see that nearest duplicate songs have the same artist but they have written differently on the site. So in the artists section of the site under the names begin with ‘C’ *Carter Deana* exists but again site under the names begin with ‘D’ the same artist exist as *Deana Carter*.

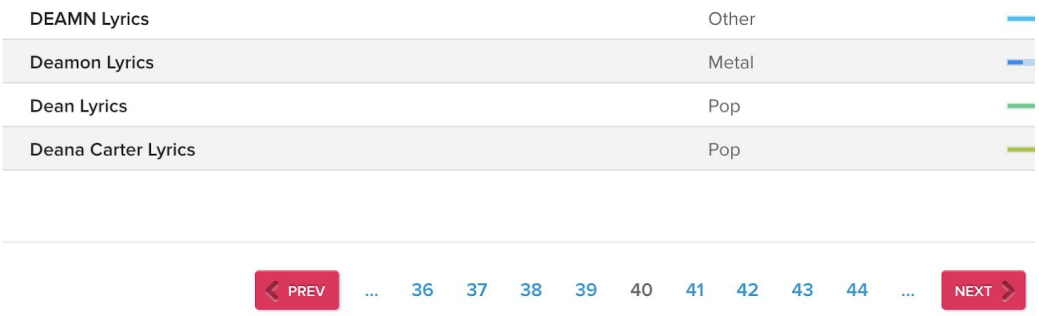


Figure 30. Deana Carter under D Section [10]

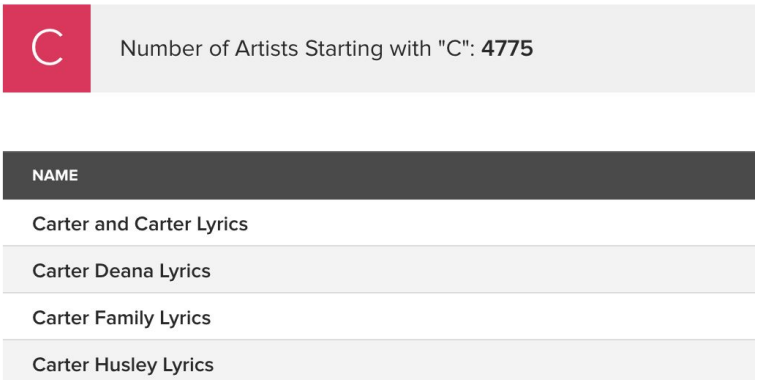


Figure 31. Carter Deana under C Section [11]

Even though they are the same artist in the web page they have different number of lyrics under their names. For example, Deana Carter has 67 different song in her page but Carter Deana has only 50 different songs. So, even though they are the same there is a difference [12, 13]. Thus in our dataset they are shown as two different artists so they have nearest duplicate songs. Even though the results are a bit unexcited for this dataset we saw that our implementation works very good.

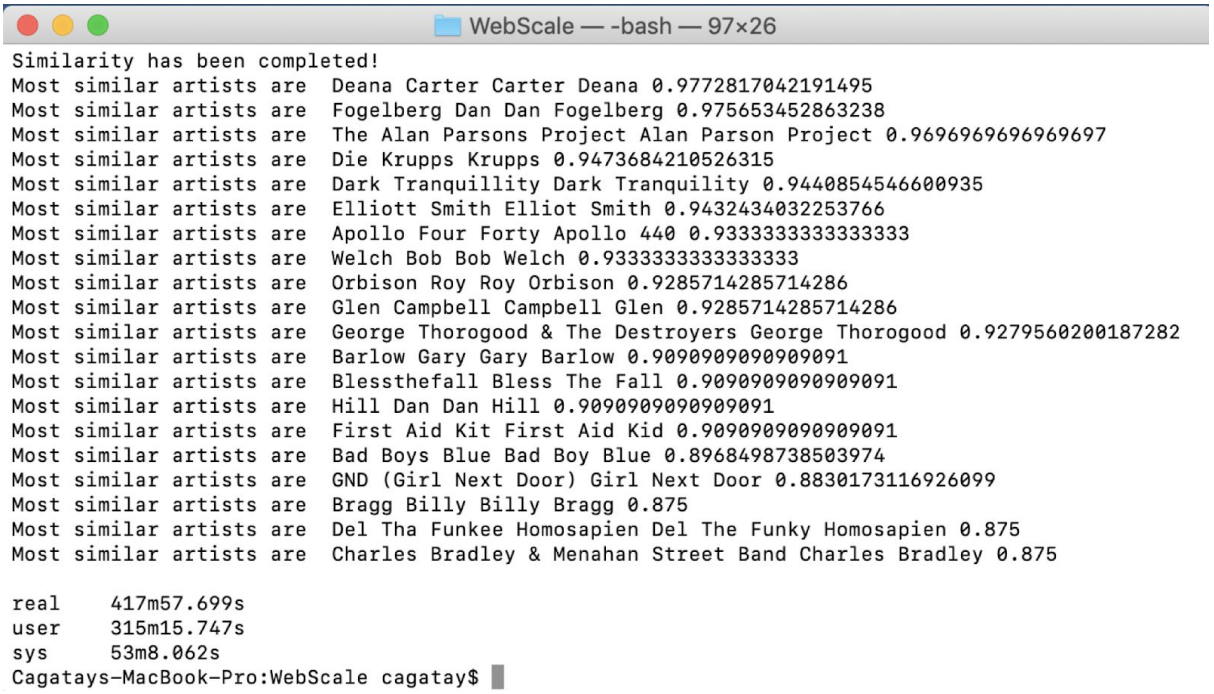


Figure 32. Results

5.3 Running Map Reduce on Datasets

When we run our code, nearly 1-2 minutes is spent for initializing and scaling worker count. When we tested 55k dataset, shingling and applying LSH parts took 3 minute, for 427k songs dataset, it took 8 minutes. So Map Reduce implementation handled this part very efficiently. On our code, we specified number of hash functions as 100, number of bands as 20 (5 rows per band). We continued our experiments with our crawled dataset.

Algorithm which is described in the previous sections spent much time on combinations part, especially when we did not have good shingle and lsh functions. If there are too many song bands which fall into same bucket (because of bad shingle and hashing), combination size increases a lot. Since combination run time is nearly to n^2, with bad hash function, we even found billions of pairs which can be seen on Figure 33.

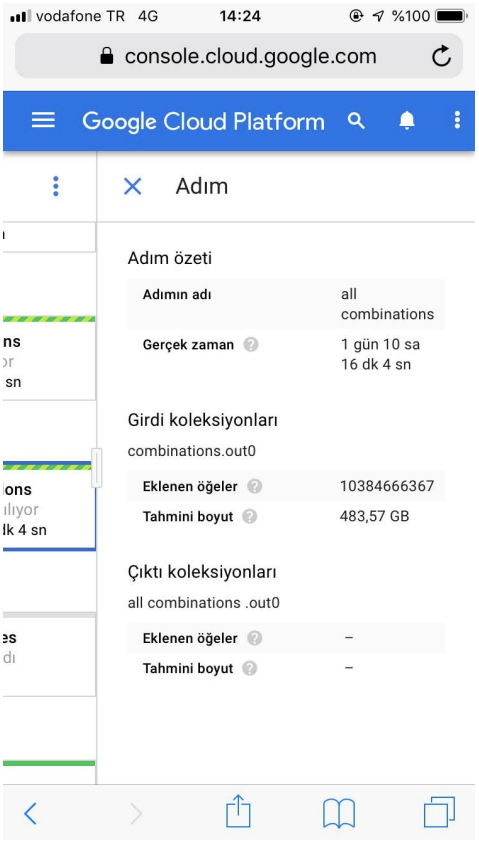


Figure 33. Combination Collections

As it can be seen from above figure, calculation of combination wasted 483 GB of space which is huge.

After implementing a good shingle and lsh functions, we tried it on our crawled dataset and execution took 4 hours to finish executing which is slightly better than previous algorithm. Moreover, if we had more available worker quota, execution would be faster.

Running 57k data on cloud with this algorithm took 16 minutes which is nearly same with one pass implementation.

6. Discussion

In the project we tried to find near duplicate items using LSH. But we also wanted to learn how much efficiency does LSH provide us. So in order to test that after creating the signature matrix we tried to find near duplicate items using Jaccard Similarity

instead of LSH. This resulted as a much slower implementation and even for the 55,000+ lyrics dataset after 71 minutes it was still calculating Jaccard Similarity for the documents so we terminated the process. It makes sense because it works in O(n^2) however in our code we use an much more efficient way and calculate near duplicate documents in O(n.m) time.

On the other hand, map reduce algorithm performance on smaller datasets is not really more efficient than one pass implementation. It is because of communication between workers. We cannot benefit from parallelism on smaller datasets. However on bigger datasets, map reduce implementation is really fast. The bigger dataset we have, the more efficient map reduce algorithm is.

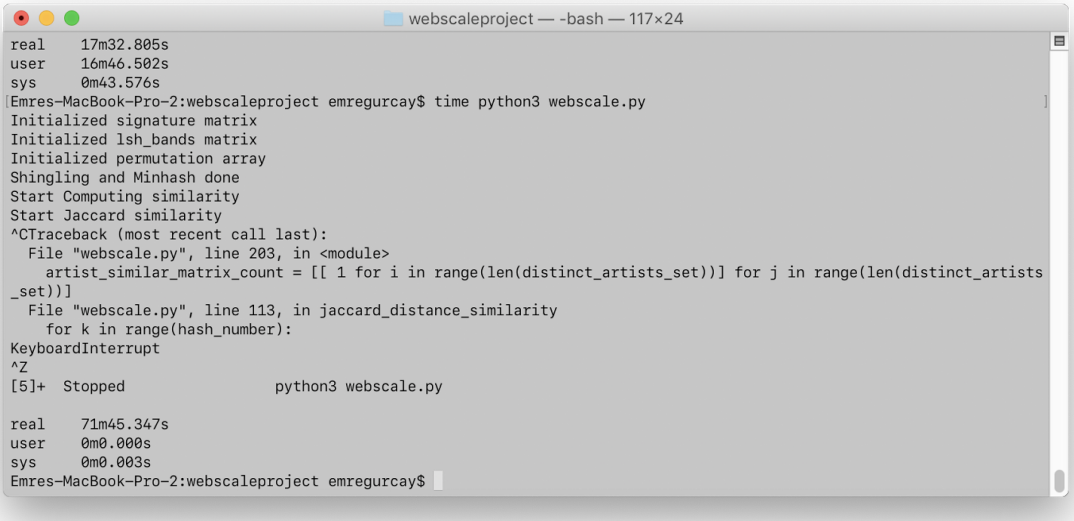


Figure 34. Terminated Jaccard Similarity Process

7. Future Work

In our project we find near duplicate items; however, we can add an another layer to combine current system with a content based similarity checking to find duplicate items. Maybe one day, if content similarity checker will be implemented, first thing to do is to improve our dataset so that there will be no duplicate artist or cover songs.

8. Workload

In this project all team members fulfilled their duties equally and on time. At first we divided work load as two part as Dataset Crawling and One-Pass implementation. Emre Gürçay and Emre Başar worked on the One-Pass implementation while Erdem Adaçal and Çağatay Küpeli was working on Dataset Crawling. After the first implementation Erdem Adaçal and Çağatay Küpeli worked on improving the One-Pass implementation and Emre Gürçay and Emre Başar worked on the Map - Reduce model implementation.

References

[1] “Unknown artist sues Taylor Swift for $42 million, takes advantage of U.S. copyright laws,” *The State Press.* [Online]. Available: http://www.statepress.com/article/2015/11/copyright-laws-are-too-strict-for-artists.

[2] “380,000+ lyrics from MetroLyrics,” *Kaggle*. [Online]. Available: https://www.kaggle.com/gyani95/380000-lyrics-from-metrolyrics

[3] “55,000+ Song Lyrics,” *Kaggle*. [Online]. Available: https://www.kaggle.com/mousehead/songlyrics

[4] “Aarni - Mental Fungue Lyrics,” *MetroLyrics*. [Online]. Available: http://www.metrolyrics.com/mental-fugue-lyrics-aarni.html

[5] “Port of Google's language-detection library to Python,” *Github*. [Online]. Available: https://github.com/Mimino666/langdetect

[6] “Best and/or fastest way to create lists in python,” *Stack Overflow*. [Online]. Available: https://stackoverflow.com/questions/20816600/best-and-or-fastest-way-to-create-lists-in-python

[7] “What's the difference between Hillsong Worship and Hillsong United?,” *Quora*. [Online]. Available: https://www.quora.com/Whats-the-difference-between-Hillsong-Worship-and-Hillsong-United

[8] “Ella Fitzgerald Sings the Irving Berlin Song Book,” *Wikipedia*. [Online]. Available: https://en.wikipedia.org/wiki/Ella_Fitzgerald_Sings_the_Irving_Berlin_Song_Book

[9] “ella fitzgerald sings the irving berlin songbook,” *Google*. [Online]. Available: https://www.google.com/search?q=ella+fitzgerald+sings+the+irving+berlin+songbook

[10] “Searh For Music Artists and Song Lyrics d Page 40,” *MetroLyrics*. [Online]. Available: http://www.metrolyrics.com/artists-d-40.html

[11] “Searh For Music Artists and Song Lyrics c Page 23,” *MetroLyrics*. [Online]. Available: http://www.metrolyrics.com/artists-c-23.html

[12] “Deana Carter Song Lyrics,” *MetroLyrics*. [Online]. Available: http://www.metrolyrics.com/deana-carter-lyrics.html

[13] “Carter Deana Song Lyrics,” *MetroLyrics*. [Online]. Available: http://www.metrolyrics.com/carter-deana-lyrics.html

[14] “Beam Overview”, *Apache Beam* [Online]. Available: https://beam.apache.org/get-started/beam-overview/