Report

Content extraction and search using Apache Tika – Employment Postings Dataset contributed via DARPA XDATA

Chen Qian	Jixin Liu	Kang Wang
1195187409	**4527496546**	**1093294783**
qianchen@usc.edu	jixinliu@usc.edu	kangwang@usc.edu

Abstract

Near-duplicates web documents are abundant. Two such documents differ from each other in a very small portion. Such differences are irrelevant for web search. So we can improve the web crawler if it can assess whether a newly crawled web page is a near-duplicate and skips crawling it.

In this article, we describe a method that uses Apache Tika to extract content from a large dataset which contains about 119 million records. Then we implemented SimHash to find the near duplicates in the records, and analyzes the result and performance of the implementation.

1.Introduction

In this assignment we are given a bunch of TSV files(Tab Separated Value) as employment postings dataset. For the dataset, each row has 19 job information columns like title, department, company and so on, but it is not well-formed and contains a lot of duplicates. Our goal is to remove duplicated data and count unique job postings. To complete this goal, we used Apache Tika to implement content extraction and search, employed customized simhash function to generate 64-bits fingerprint for each individual JSON job posting file and introduced hamming distance based on our fingerprint to detect near duplicates. It is very time-consuming for clawing huge number of individual JSON job posting files, so we implemented a multiple-threads crawler to accelerate clawing process. Furthermore, we developed a new data structure SimHashTable to optimize hashing procedure.

2.Results and Analysis

2.1 Question Answers

3.a Develop and run a simple crawler that uses Tika across the initial reduced dataset of Employment jobs to produce the individual JSON job files. How many job files were produced?

After parsing the TSV file and generated the individual job information as JSON file, there are 12,483,611 job entries in total.

4.a Develop and run a simple crawler that uses programs from the ETLLib software to output individual JSON job files from the initial reduced dataset of Employment jobs. How many job files were produced?

Using the ETLlib jsontotsv parser, 12,483,611 JSON job files are generated.

5. Develop a process for deduplicating the Employment dataset.

a.Decide what job information can be used to deduce uniqueness

The de-duplication has two steps. The first one is to eliminate the exactly same job entries. We used “FirstSeenData” and “LastSeenDate” to decide the uniqueness of job information. The second step is to identify near duplicates, we implement SimHash to realize it.

b. Integrate deduplication into your programs from #3 and #4c. How many job files were produced when deduplication is enabled, and why?

After first step of de-duplication, the elimination of same job entries.
Tika crawler generated 273,629 JSON files
The ETLLib crawler generated 276,246 JSON files.
After the second step, the elimination of near duplicates,
Tika crawler generated 268,012 JSON files
The ETLLib crawler generated 266,981 JSON files.
Time consumption for Tika crawler:
Parse TSV and generate JSON takes 96 seconds.
De-Dulplication takes 38 seconds.

6. Develop a program or a script to run Tika across the full dataset of Employment jobs to produce the individual JSON job files

a. How many job files were produced (no deduplication)?

Without the de-duplication, we got 118,766,934 JSON files.

b. Turn on deduplication – how many job files were produced?

Eliminate the exactly same job entries, we got 2,108,463 JSON files,
After removing the near duplicates, we got 2,052,067 JSON files.
Time consumption for Tika crawler:
Parse TSV and generate JSON takes 1901 seconds.
De-Dulplication takes 2150 seconds.

2.2 Analysis and Report

1) what you noticed about the dataset as you answered the questions in Part #3. Why do you think there were duplicates? Were they easy to detect?

According to our analysis of the dataset, we found out that there are two interesting columns “FirstSeenDate” & “LastSeenDate”. They have following properties

i. In each records, firstSeenDate is before or equal to the LastSeenDate.
ii. The lastSeenDate is the same with the date in the name of that TSV file.

So we made a hypothesis:

The crawler that crawled the http://www.computrabajo.com will record the firstSeenDate and the lastSeenDate of the URL,and the lastSeenDate would be same as the date that the crawler crawled the website.

This gives us a very intuitive solution to remove the same job postings.

For each record we got from the TSV file, if the FirstSeenDate is before the LastSeenDate, it must be a duplicated record. However, if the FirstSeenDate is as same as LastSeenDate, this record must be a new and unique record.

We test 60 consecutive TSV files, it shows that most of job entries appear about 40 times. This result proves the hypothesis we make.

Besides, the started data of all TSV files are 2012-11-06. So for the job postings, if their firstSeenData is before 2012-11-06, we also need to consider if it appears at the first time. Since url is uniquely identify a job, we use a hashMap is introduced to record all these jobs’ URL in order to decide whether these jobs appears before.

Overall, we can conclude the exactly same job postings can be identified by analyzing the firstSeenDate and the LastSeenDate.

For identify the near duplicates, we choose several columns for simHash, and assign each column with different weight.

We exclude “firstSeenDate”, “lastSeenDate” and “url”, since these three columns are used for the formal step to remove the exactly same job postings. “phone number” and “fax number” are removed, since most jobs do not contain these information.

For the rest of columns, we given weight based on the principle that if most jobs have similar value in that columns, we assign low weight. Otherwise assign high weight.

2)Describe your algorithm for deduplication. How did you arrive at it? What worked about it? What didn’t?

Here we only discuss the algorithm for identifying near duplicates.

We employed fingerprint with simhash algorithm for de-duplication. First we read Google paper “Detecting Near-Duplicates for Web Crawling” to understand the theory about simhash, fingerprint and hamming distance. Then we searched some references from web blogs to figure out each step of this algorithm. But simhash does not work well for short length documents, each of individual JSON job posting file is a short string, so it needs to be very careful to give each feature vector appropriate weight.

3)Describe your simple crawlers. What could they do better?

The Tika crawler read JSON files and generate finger print for it. After all files are crawled, de-duplication will be executed.

Beside, our team implemented a multiple-threads file crawler in Java. We created eight threads to claw by dividing whole file set into eight sections. In future work, we think we are able to further speed up clawing by introducing MapReduce algorithm.

4)Also include your thoughts about Apache Tika and ETLlib – what was easy about using them?
What wasn’t?

For Tika, it is a great tool to extract content as well as the metadata. When I used Tika in first time, I found it easy to understand and implement. The parse function has only four parameters, it is simple to execute. However, Tika could realize complex operations.

Customized parser can parse data in specific manner and output in standard XHTML.

Customized content handler builds the desired output content and format.

Tika is really a cool craft, we will explore more interesting features in Tika.

For the ETLLib, it did a excellent job.But for this assignment for parsing a single tsv file to JSON. However ETLLib tsvtojson parser doesn’t match our needs well. Tsvtojon has to package a JSON file with a wrapper, but in this assignment we are required to generate individual JSON file for every record. Moreover we need to parse multiple TSV files while the ETLlib only supports parsing single TSV per excution, it cannot automatically parse all TSV files in a directory. Also, we found out that the bottleneck of the performance is I/O operations. If we could remove duplicates during the crawling phase, and only generate unique JSON files it would significantly improve the program process. So we decided to re-wrote a custom tsvtojson parser based on the original one that can satisfy our needs.

5)De-duplication result analysis

After the first step of de-duplication, exactly same job postings are eliminated.

In Tika crawler, only 2.19% job entries are kept.

In ELTlib crawler, 2.21% job entries are remained.

These results are consistent with the hypothesis and observation we analyze before. Most jobs appear in about 40 consecutive TSV files.

For detection near duplicates, only 5000 - 10000 job postings are removed. This is because most duplicates are removed in the first step.

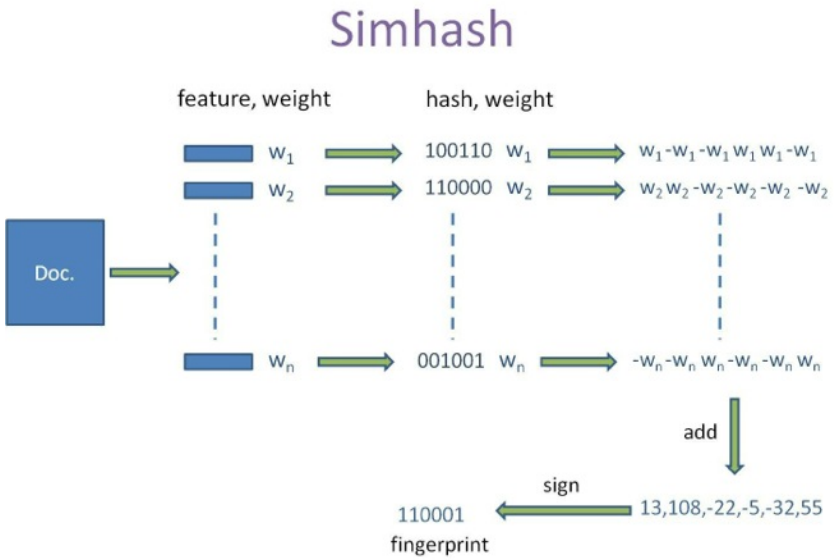
We also test the case that first step is disable. We only use simHash to detect duplicates. The result is similar, about 260,000 job postings are left.

This also proves the out first step of de-duplication is right.

3.Theory and Techniques

3.1 Fingerprint with SimHash

SimHash is a dimensionality reduction technique. It maps high-dimensional vectors to small-sized fingerprints. It is applied to web-pages as follows: we first convert a web-page into a set of features, each feature tagged with its weight. A set of weighted features constitutes a high-dimensional vector, with one dimension per unique feature in all documents taken together. With SimHash, we can transform such a high-dimensional vector into an f-bit fingerprint where f is small, say 64.



After converting documents into 64-bit SimHash fingerprints, the next challenge is we need to quickly discover two fingerprints differ in at most 3 bit-positions, so we introduced hamming distance to solve this problem.

3.2 MurmurHash

MurmurHash is a non-cryptographic hash function suitable for general hash-based lookup. It was created by Austin Appleby in 2008, and exists in a number of variants, all of which have

been released into the public domain. When compared to other popular hash functions, MurmurHash performed well in a random distribution of regular keys.

3.3 SAX

SAX (Simple API for XML) is an event sequential access parser API developed by the XML-DEV mailing list for XML documents.[1] SAX provides a mechanism for reading data from an XML document that is an alternative to that provided by the Document Object Model (DOM). Where the DOM operates on the document as a whole, SAX parsers operate on each piece of the XML document sequentially. A parser that implements SAX functions as a stream parser, with an event-driven API. The user defines a number of callback methods that will be called when events occur during parsing.

4.Implementation

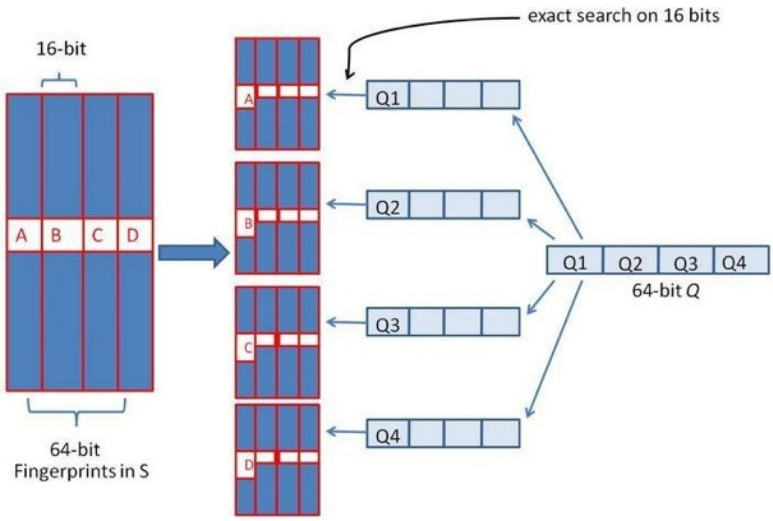
4.1 SimHash Implementation

In SimHash implementation, first we obtain value of each column by using a JSON handler to parse each job posting file, and then we combine these values into a single string, finally we developed a document spliter to map each column into feature vector. After finishing file content extraction, based on simhash theory, we calculate each vector with its weight and reduce them into final 64-bit fingerprint by employing murmurhash as key hashing function. In murmurhash function, we generate 64-bits hash code for each feature vector. In the end, we encapsulate 64-bit fingerprint into a Java object BigInteger.

4.2 SimHashTable

We developed a new java class named SimHashTable to accelerate the process of detecting duplicates. In the SimHashTable, we only use partial 16-bit out of 64-bit fingerprint as hash key and set a linked list as value. When a fingerprint comes it firstly split the fingerprint into four sections, for each section we search hashtable to see if it exists, if it exists we traverse corresponding linked list to check hamming distance between input fingerprint with all fingerprints in the list, if not, we put the section as key into hashtable and create a new linked list as value and insert intact 64-bit fingerprint into the list.

After generating all individual JSON job posting files, the number of files is more than 12 million(2^24). Without this section indexing it will have more than 12 million entries in hashtable and for each permutation within 3 hamming distance we need to do C(3, 64) = 41664 probes. In addition, with huge number of entries the efficiency of traditional hashtable will be undermined. By using this indexing, we at most have 2^16 entries in hashtable which is quite acceptable, and assume all data are uniform-distributed, then in each linked list we will have 4 * (2^24 - 2^16) = 1024, so we just need to do 1024 probes.



In duplicates detection, we search the SimHashTable to see if we can find any fingerprints in the table whose hamming distance differ 3 or less with input fingerprint. Based on pigeonhole principle, we determine two records are duplicates by check their hamming distance less than 3 or not.

4.3 Crawler

We developed a multiple-threads crawler in Java for JSON job posting file clawing. We divided whole file set into 8 sections and created 8 threads to run clawing. In our original idea, we used listFiles() method in Java to traverse all files under specific directory, but during clawing we found it is very slow because this method will load all file basic information and create a file array to store these information and this process will cost a lot of time. So, to solve this problem we design our crawler to claw file one by one via spelling file name.

We also wrote a custom jsontotsv Python parser and integrated exact deduplication while crawling the TSV file. It can do the batch processing for all tsv files.

The idea is simple, we open all tsv files in order. For each TSV file, we open it and read it by line. For each line we split the line into an array using tab as separator. We build a dictionary with column headers as key and corresponding content in array as value. Finally we use our exact deduplication method to decide if it is a duplication and use Python JSON generating API to create the unique job JSON file.

6.Conclusion

We wrote TSV parser to generate job information JSON files. With deduplication, we successfully eliminated 98.3% duplicated data from the original results . The effects of de-duplication is significant. Moreover, by using techniques like multithreading, efficient algorithm and data structure, our program achieved pretty good performance.

7.Assignment Divide:

Chen Qian: Tika TSV parser & crawler, Java Deduplication(Exact deduplication, SimHash Deduplication), python simHash Deduplication

Jixin Liu: ETLlib parser scripts, python crawler, python exact deduplication, Java SimHashTable.

Kang Wang: Multiple-threads file crawler, FileLister, Java SimHash Table.