UNIVERSITÀ
DEGLI STUDI
DI MILANO

Algorithms for Massive Datasets

Project (Q1)

Professor Dario Malchiodi

Prepared by:

Chan Zhong Ping, Jeffrey (V10153)

1. Setting up Kaggle API and Dataset

The Kaggle dataset "Yelp Dataset" was accessed and downloaded on 29 May 2023.

```
import os
os.environ['KAGGLE_USERNAME'] = "XXX"
os.environ['KAGGLE_KEY'] = "XXX"
!kaggle datasets download -d yelp-dataset/yelp-dataset
!mkdir reviews #creating a new directory for the dataset
!unzip yelp-dataset.zip -d reviews #unzipping the dataset
```

Upon setting up the API key, the file is downloaded and a new directory "reviews" was created to unzip all the files into a single directory. As this project solely focuses on the review file of the dataset, the main directory that is accessed in the experiments is found at `reviews/yelp_academic_dataset_review.json`.

2. Pre-processing Methods

2.1 Data Cleaning

The data cleaning methods used in this project are as follows:

a. Changing all characters to lowercase: The reviews format is not standardised; some reviews have capitalised words after a punctuation mark, while others are all in lowercase. To ensure the data format is standardised, all characters are changed into lowercase.

b. Removing punctuations in the text body: Similarly, reviews which are longer tend to be full of punctuation marks, which affects the analysis of the dataset in the later cells. To reduce such issues, all punctuation marks are removed.

c. Lemmatize words into their base form: As some reviews tend to be quite similar, lemmatising the dataset to reduce words into their base form can help to increase the similarity between two reviews. For example, words like "eating" and "improvements" will be lemmatized to "eat" and "improve".

d. Removing all stopwords in the text body: Similar to lemmatize, the removal of stopwords helps to increase the similarity between the reviews. Stopwords are common words such as "to" or "a", which are used to make proper sentences but are not needed in analysing the data. This ensures that the context of the reviews are retained without bloat words.

In order to facilitate the data cleaning process, two helper functions were created. As this function requires the use of lemmatizing and stopwords, the relevant packages have to be retrieved first as shown:

The first helper function `remove_punctuation` creates a function to remove punctuations, which will be used in the second function. The second helper function `clean_data` performs the data cleaning methods mentioned above. In the `clean_data` function, after changing all characters to lowercase and removing punctuation, the text is split into a list of tokens by using whitespace in the text body. The next line applies lemmatization to each word in the list of words in order to reduce the words to their base or root form. The reason for lemmatization and removal of stopwords is to ensure higher accuracy of comparison during k-shingling in the next step. The function checks if the word is a stopword, which is a list created using the NLTK stopwords library, and keeps the word and lemmatizes it if there is no match found. Finally, all the tokens are joined back into `cleaned_text` and separated by a space to retain the original word text format.

2.2 K-Shingling

K-shingling in this context is breaking down the text into strings of words that are of k-length; this means that a shingle of k-length 1 depicts the entirety of words in the document. This is done before comparing the similarities of the document. There are two ways to perform k-shingling – either to perform a word shingle or a character shingle. Word shingling is faster in large datasets, but compromises on the accuracy whereas character shingling takes a longer amount of time but is often more accurate. In this project, word shingle is used with k size of 2, as the purpose is to find similar tweet/review pairs. The function first splits the text into individual words, then runs an interaction through the list of words up till the $(n-1)^{th}$ word. It then creates a string consisting of two consecutive words in the text joined by a space. Previous versions of the code stored the processed values into memory, which resulted in frequent crashes. However, using generators helps to process values on the go, which significantly reduces the memory footprint. The code is written very similar to a normal return function, except `return` is replaced with `yield` instead. The shingling process is illustrated below.

Pre-processed Text	*Cleaned Text*	*Shingles*
Wow! Yummy, different, delicious. Our favorite is the lamb curry and Korma.	wow yummy different delicious favorite lamb curry korma	['wow yummy', 'yummy different', 'different delicious', 'delicious favorite', 'favorite lamb', 'lamb curry', 'curry korma']

3. Processing Reviews & Building MinHashLSH Index

Chunking

As this is a large dataset, the processing time and resources needed is rather intensive. When dealing with large datasets, splitting the data into chunks can allow for many benefits:

a. Chunking reduces memory footprint, as the data is split into smaller chunks for the algorithm to process a more manageable portion of the data at a time. This is especially useful when the entire dataset cannot fit into the memory.

b. Multiple chunks can be processed concurrently, taking advantage of multi-core processors. This is called parallel processing and can significantly reduce the time needed to process and analyse data. However, this experiment does not make use of concurrent processing due to limited processing power provided by Google colab.

c. Data can be incrementally processed in a streaming fashion, where data is analysed as it becomes available. This speeds up processing of data as the data can be analysed in small portions without waiting for the entire dataset to be available.

In this dataset, the chunksize is reduced to 10,000. This is to break down the large file into smaller chunks for pre-processing. The chunk size can be adjusted accordingly, where smaller chunk sizes require less computational power at the expense of speed and larger chunk sizes require more computational power but are faster. The number of chunks to be processed are also limited to 20 as a proof-of-concept as to how the code works, as well as due to limited memory in Google Colab.

MinhashLSH

The number of permutations in this experiment is set as 128. While the number can be any value that is a positive integer, setting this number to be a power of 2 is recommended for efficiency purposes. This is because the MinHash signatures can be represented using bit arrays which are memory-efficient and enable fast bitwise operations like union, intersection, and similarity estimation. The larger the permutation, the more accurate the similarity search, but this comes at the cost of computing complexity.

The threshold value for this experiment is set at 0.5, and can be adjusted according to similarity pairs search requirements. This value is used to determine the minimum similarity threshold for identifying similar items. In the MinHashLSH algorithm, similarity between two items is often measured using the Jaccard similarity coefficient. The Jaccard similarity coefficient is calculated as

$$J(A, B) = \frac{A \cap B}{A \cup B}$$

It provides a measure of how similar two sets are based on the common elements they share. The Jaccard similarity coefficient is obtained by counting the number of matching elements at corresponding positions in the MinHash signatures and dividing it by the total number of permutations in the MinHash signatures. When performing similarity search using MinHashLSH, candidate pairs that have a similarity above the specified threshold are generated by the algorithm. These candidate pairs are potential matches that need to be further verified using more computationally expensive methods, such as comparing the actual Jaccard similarity between the MinHash signatures.

In this section, the chunks are passed through an iteration to process the text using the helper functions. Once the texts are shingled, a second iteration is passed to encode the shingles with UTF-8 encoding to ensure consistency and compatibility when working with text data that may contain characters from different languages and scripts. After encryption, the following details are passed into an empty list and appended: `review_id`, `user_id`, `business_id`, `stars`, `pre_cleaned_text`, `shingles` and `minhash`. To check that the text has been processed and shingled correctly, the first 5 processed reviews in the first chunk are printed in the output. Upon visual verification that the data has been properly processed after printing the first 3 chunks, the iteration continues running while skipping the print function. Once the chunks have been confirmed to be properly processed, an iteration is passed to insert the review ID and its MinHash object into the MinHashLSH index. The chunks are then accumulated onto a single list at the end of the loop.

4. Finding Similar Pairs

In this chunk, the process of cleaning the text, text shingling and shingles encoding is repeated. Upon creation of `minhash`, the code queries the LSH index to find similar review pairs using the computed MinHash signatures. The similar items retrieved from the LSH index are then iterated over, and the review IDs are checked against each other to ensure the MinHash match is not simply a match of itself before it is considered a similar pair. The pair of review IDs is then appended to the `similar_pairs` list. The illustrations below represents a simplified version of how MinHashLSH works

$$Let\ \{S_1, \ldots S_n\}\ be\ the\ set\ of\ shingles\ for\ each\ review$$

Shingles	*MinHashing*		*Minhash Signature*
$S_1 \rightarrow$	101110	$\rightarrow$	[1, 2, 1, 1, 1, 2]
.	###		
.	###		
.	###		
.	###		
$S_n \rightarrow$	101001	$\rightarrow$	[1, 2, 1, 2, 2, 1]

LSH:

Document_1:

Hash_1 -> Bucket_1
Hash_2 -> Bucket_2
Hash_3 -> Bucket_1
Hash_4 -> Bucket_2
Hash_5 -> Bucket_2
Hash_6 -> Bucket_1

Document_2:

Hash_1 -> Bucket_1
Hash_2 -> Bucket_2
Hash_3 -> Bucket_1
Hash_4 -> Bucket_1
Hash_5 -> Bucket_1
Hash_6 -> Bucket_2

In this case, Document_1 and Document_2 hashed to the same buckets for some hash functions, which indicates a potential match for similar items. LSH provides an efficient way to find similar items through focusing only on the documents that hashed to the same buckets. To calculate the similarity between two documents, their MinHash signatures are compared and the number of matching values are counted at corresponding positions. The more matching values they have, the higher the similarity between the documents.

5. Similar Pairs Dataframe

For this code chunk, a pandas dataframe is first initialised using the `similar_pairs` list which contains the pairs of review IDs. By using the left join technique against the `data_clean` dataframe, the matching rows of review IDs from `data_clean` is added to the new dataframe, along with additional details such as the user ID, business ID, review stars and pre-cleaned text. This information is useful to compare which businesses have a lot of similar reviews, or which user ID has a lot of similar reviews. These additional filters can be processed after the file is retrieved. For this experiment, due to the purpose of it being a proof of concept for finding similar items, additional information like business ID and user ID have been removed to declutter the final dataframe.

The dataframe is then renamed to indicate clearly the additional information belongs to the first set of similar reviews. The column `review_id` is dropped from the dataframe as it is a non-useful information in the presentation of the collated data. After that has been done, the dataframe is once again merged with `data_clean` using the left merge function, but is based on the rows of the second set of review IDs that are deemed as similar pairs. The review ID column is once again dropped as it is not necessary for the final dataset.

Lastly, the final dataframe has been created. For the sake of presentability and ease of reading the final data, the columns have been rearranged to ensure consistency in the format of the information which the similar pairs are presented in. This allows for the final dataframe to be organised so that subsequent manipulations or file export is possible without significant manipulation to the dataset.

6. File Export

The final dataframe is converted to a comma separated value (CSV) file and exported as a downloadable file. A copy of the results of this experiment is uploaded onto Github which is available for viewing.

7. Discussion

Scalability

This project is done on a small scale as a proof of concept for the algorithm used, with the chunk size limited to 10,000 and 60 chunks. The amount of memory used was approximately 9 GB and the processing of the chunks took 30 minutes, while the search for similar pairs of reviews took approximately 34 minutes. To find the scalability of the project, it is necessary to determine the time complexity (big “O” notation) of each section of the code.

- Reading the data: Reading of data in chunks has a complexity of O(n), with n representing the total number of reviews. The time taken to read the data will likely increase in a linear manner.
- Processing the reviews: The process of text cleaning, lemmatizing and shingling has a complexity of O(m), where m is the number of words in a review. Realistically, each review is not likely going to have so many words that it will make a huge impact, therefore the review processing can be assumed to have a constant time complexity.
- Generating MinHash signatures: The time complexity of generating MinHash signatures is O(k * m), where k is the number of hash functions and m is the number of shingles. Similarly, as the number of words do not deviate too much, the number of shingles will not deviate significantly either. Similarly for k, it is set to 128 for this project and can be assumed to be constant.
- Indexing of MinHash objects in LSH: The indexing of MinHash objects has a time complexity of O(n), where n is the number of reviews.
- LSH query for similar items: The time complexity of LSH query is O(b), where b is the number of buckets to search
- Finding similar pairs and merging final data: Both of these processes have a time complexity of O(n) each, where n is the number of reviews in the dataset.

Overall, the time complexity of this project is dependent on the number of reviews (n), number of words in the reviews (m) which affects the number of shingles, number of hash functions (k), and number of buckets to search (b). Assuming k and m to be constant, the big “O” notation of this project can be defined as:

$$Big\ "O"\ notation\ =\ O(n\,(\,n\ +\ b))$$

Based on the big “O” notation, the worst-case time complexity is a combination of linear and quadratic. The dominant term is the quadratic term, meaning that the project may not scale too well as the amount of data increases.

Comments

Ideally, for big datasets the code should scale well as the datasets become larger. The processing of the entire Yelp reviews dataset does not fit into the memory space provided by Google Colab. There are a few suggestions and alternatives to be considered:

1. Tuning the parameters of the project – The parameters like permutations number can be tuned to reduce the amount of memory required for the algorithm. While this sacrifices accuracy, it can help with reducing the memory footprint.
2. Using Apache Spark – Designed for large datasets, Spark is particularly well suited for big data analytics, and has support for multiple programming languages, especially those commonly used for data analytics like Python and R.
3. Writing intermediate results to disk – A potential solution to the problem of overconsumption of memory is to write the intermediate results from text processing and shingling into disk memory. However, a downside of this method is the time required to run the code. It is also more prone to mistakes as the intermediate results may not be properly formatted.

8. Code and Other References

Bryłkowski, H. (2017, November 14). *Locality sensitive hashing - LSH explained*. Medium. https://medium.com/@hbrylkowski/locality-sensitive-hashing-explained-304eb39291e4

Ghosh, S. (2021, October 29). *Text similarity using K-shingling, Minhashing and LSH*. Towards AI. https://towardsai.net/p/l/text-similarity-using-k-shingling-minhashing-and-lshlocality-sensitive-hashing

Kemik, H. (2021, May 11). *Explaining LSH-minhash-simhash*. Medium. https://medium.com/carbon-consulting/explaining-lsh-minhash-simhash-c3cc33040030

Stratis, K. (2023, April 1). *How to use generators and yield in python*. Real Python. https://realpython.com/introduction-to-python-generators/

Yelp. (2022, March 17). *Yelp dataset*. Kaggle. https://www.kaggle.com/datasets/yelp-dataset/yelp-dataset?resource=download

9. Declaration

I/We declare that this material, which I/We now submit for assessment, is entirely my/our own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my/our work. I/We understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me/us or any other person for assessment on this or any other course of study.