Querying Amazon Reviews Using TF-IDF and Cosine-Similarity with Probabilistic Algorithm

1st Angelo Ramírez
Computer Science Department
Costa Rican Institute of Technology
San José, Costa Rica
angramirez@ic-itcr.ac.cr

2nd Guilliano D'Ambrosio
Computer Science Department
Costa Rican Institute of Technology
San José, Costa Rica
gdambrosio@ic-itcr.ac.cr

***Abstract*—This projects intends to index a huge amount of files and accessing those files upon a search like a search engine by using term frequency (TF) and inverted document frequency (IDF) to help determine how important are the words in a text and which words give more information about the text. To be able to find the most relevant texts upon a given search query. This project indexes million of archives and stores the a file of the indexed archives via the TF-IDF algorithm. It stores the information in files because of the huge amount of data it is managing. When the query is made it searches upon the file with the indexed terms to look for the document that has the highest TF-IDF for the words in the query. By managing data in files we achieve the fastest solution and avoid overcrowding the main memory that would happen if everything is kept under variables in the project. The information is written to the indexed file partially to avoid giving to much data to main memory that would slow down the indexing process. Instead of indexing all the files and then writing it to the specific file, every given amount of files all the data is written and the main memory is cleared so everything is kept as optimal as possible. The goal is to achieve a program that indexes huge amounts of data and searches for the most relevant files upon a query in the most effective way by using indexing algorithms like TF-IDF and query solving algorithms like cosine similarity.**

I. Introduction

This indexing and query solving methods have been used by the most important search engines in the world. The indexing method used in our project called TF-IDF is being used by over 83 percent of text-based recommender systems in digital libraries from around the world. This algorithm for data indexing was invented by Karen Sparck Jones. Sparck Jones was a very known computer scientist graduated from the University of Cambridge known for her large effort to encourage women into computer science and for her work on natural language and the development of the term frequency and the inverse document frequency. TF-IDF may vary in implementations when it comes to normalizing the IDF. There are many ways of calculating the TF-IDF, in this project the TF is calculated with a raw count for each word in the document, meanwhile the iTF is normalized by a logarithmic scaling function. Query solving is managed with cosine similarity. It is widely used in information retrieval and text mining. It helps tell how alike are two document by their given TF-IDF number for each word. The cosine similarity measures the similarity between two vectors in the same inner product space by using cosine two measure the angle between them. For the big data management we used pandas. It is a library that offers methods and objects to manage big amounts of data. It offers data structures that make easier working in areas like data analysis and modelling. DataFrames are a data structure from pandas, you can think of it like a SQL table or a spreadsheet. They are a 2 dimensional data structured with columns that can hold different types. DataFrames were used in the process of writing to CSV files all the data needed from the file such as paths, TF-IDF, and indexes for the words. Also they are used in the reading of the files as they provide a way of reading by chunks the data, this help us avoid the overcrowding of main memory. This good management of data by with DataFrames and pandas helps the project improve its efficiency as it works with big data groups. Numpy is a math based library that is made for scientific computing. It provides multidimensional array objects and very fast mathematical operations for these objects. Numpy helps improve efficiency when working with matrices because it provides operations for this type of object that are way faster than one provided by ourselves. Numpy is used in the previously mentioned algorithm cosine similarity, its multidimensional array operations provide a very optimal way to calculate cosine similarity.

II. Related Work

There have been a lot of projects developed in the area of data indexing and searching. Many papers highlight the high relevance of the TF-IDF algorithm to obtain an efficient answer upon a query. Also pandas is being to manage big data sets that main memory is not able to manade. Mathematical operations from Numpy are being used for multidimensional array objects and vectors because of its efficiency.

A. Cosine vs Euclidean

A project was made by the developers of Chris Emmery. It consist on the comparison of Cosine and Euclidean distances in the measuring of the similarity between vectors that with values that represented weight and length. In this project they made the comparison between a the distance given by cosine algorithm and the one given by euclidean algorithm. As a result euclidean was more accurate for this type of test but cosine works better when the magnitude of the vectors to be

compared does not matter like when we are working with text data represented by word counts. [1]

B. Page Rank

Page Rank is the algorithm used by Google since its invention to rank pages. This is a very important algorithm because it prevents the false ranking provided by search engines that ranked pages by similarity between query and words because the author of the pages could modify it to be better ranked upon the words contained in the page. The Page Rank invented by Larry Page, consists on ranking pages by the amount of hyperlinks that refer to a certain page. As more links reference the page the better the rank is. So when the rank is set a complementary algorithm similar to TF-IDF helps the search engine determine which page is the most similar to the query from the user and then gives the better ranked. [2]

C. Pandas

Pandas is a library used for data analysis on Python. The dataframes are the main data structure on Pandas. Data frames are 2-dimensional labeled data structures with columns of different types. You can think of it like a spreadsheet or SQL table. They have many tools that can help write and read CSV files, also the ability to work with chunks of files instead of reading whole files into main memory. [3]

D. Amazon Review Data

A compilation of Amazon reviews turned into a dataset of over 147 million reviews, from 1996 to 2014. Compiled by Julian McAuley from UC San Diego. [4]

III. Methodology

A. Generating TF-IDF

If TF-IDF is not stored in a file to be later used as an index to generate queries the program would require to read all the files and calculate it every time a query is made. Hence, a correct implementation is necessary, this implementation should work for all file amounts and sizes. The correct implementation includes file chunking, which means that not all files are stored in main memory at the same time, because when handling millions of files, the program would run out of memory. A problem then arises, how to calculate the IDF of terms while not having all files on memory. Since TF-IDF requires information of all the documents at the same time, one must iterate over all existing files, check what words appear in it, and then add them to the existing document count per word, if the word does not exist in the DataFrame, it is added. This way, we can now in how many documents a particular word appears, then, since we know the amount of documents, once all files are searched, the IDF is calculated, at the same time, while iterating through the files, it is possible to calculate the TF, since this does not require information from the rest of files. Every certain amount of files the different DataFrames are stored to CSV files, to prevent memory errors, this files store data from each word present in the files, and word TF per file. The DataFrames are then cleared so it is able to continue the processing of files. This part is the most important, without the clearing of the DataFrames it would be impossible with limited computer resources to manage big amounts of data. In turn, the amount of files that can be managed is potentially infinite, given that enough secondary memory is provided.

B. Storing TF-IDF

To store the TF-IDF for each word on each document a way of dividing the data was needed, this is where chunks where introduced. Handling all data in memory at the same time is not an option when it comes to optimal indexing and limited computer resources. So DataFrames from the Python Pandas library were used to create objects that contained the data needed to index correctly the files such as paths, TF-IDF, and words of each file. When the DataFrame reached a certain amount of data collected then it writes the information to a CSV file that stores the data. In the next step we clear the DataFrame and set it as a new one so it is ready to continue with the next chunk of information. This method of managing the large amount of data helps to improve the efficiency of the indexing and searching algorithm by avoiding overcrowding in the main memory. This process of writing the data and clearing the DataFrame happens until all the TF-IDF are stored in the CSV files.

C. Analysis for TF-IDF generation

In line 6 a chunck size is defined, this size is the amount of documents that will be read each iteration, it is optimal to modify the Pandas DataFrames with a few documents, rather than doing it over a single document at a time. Another problem is how to iterate over a huge set of files, this is achievable by using generators instead of lists, because of this, the object oriented library pathlib is used, this creates a generator for all files in a folder and all sub-folders. The idf in line 8 is used because information of words is constantly being updated, so iterating through it in every chunk would be too expensive. One way to save space, is to keep file paths at a different file, and use a index to reference them, otherwise, the filepath would have to be stored for every word in the TF file, taking up huge amounts of unnecessary space. Every time a chunk is processed, the DataFrames are appended to their respective CSV files. For every single file, the file is opened and it's contents are read. To calculate the TF, the lines are split into words, then the word is appended to a list, in the form of a tuple of the path index and the word. Then, the amount of words is counted, and the values are grouped by file index. Finally, the amount of times the word is counted is divided by the total amount of words in the documents, stored in a new column in the DataFrame, dubbed TF. Then, the IDF is updated, adding the new words and increasing the amount of documents a word appears in. Finally, when all documents have been iterated, the IDF column is added, and then stored. Finally, the TF-IDF is calculated for all words, using iterative pandas CSV reader, storing the index into the final CSV file.

1) Time Complexity: Let n be the amount of files inside a the folder named searchFolder. Initially, the fileamount is found by counting files inside the searchFolder, this takes n, then, for each file, the following process takes place: The file is opened, this takes constant time,reading the file itself depends on the size of the file, let the amount of words in a file be k. This means that lines 67-69 take $O(nk)$. Now, every certain amount of files the files contents are processed, let the chunksize be 1, this would mean that every single file read would be processed, this is the worst-case scenario. Firstly, the filepaths are saved to a CSV, which is used to display the results. Since this uses a third-party library, approximations using known algorithms for the operations done will be used. Writing to memory will take k, iterating over all the elements in the file and doing an operation will take k, grouping takes $klog(k)$, so does ordering, merging takes k^2 supposing k is the size of the DataFrames, and that the DataFrames are of similar size. Therefore the line that takes the longes inside the for loop, takes k^2, which will be done n times, giving us a time complexity of $O(n \cdot k^2)$, do note that there are plenty of operations which take a long time inside the loop, which will make real time slower than the theoretical.

Algorithm 1. Implementation of generation of TF-IDF.

```
1  def tfidf(searchFolder):  ##creates index
2      fileTree = list()
3      filePathDictionary = list()
4      first = True
5      contador = 1
6      chunksize = 10000
7      fileamount = sum([len(files) for r, d, files in os.walk(searchFolder)])
8      IDF = pandas.DataFrame()
9
10     for file in Path(searchFolder).rglob('*.*'):
11
12         if contador % chunksize == 0 or contador == fileamount:
13
14             fpd = pandas.concat(filePathDictionary)
15             fpd.to_CSV(filepath + "\\pathDictionary.CSV",
16             header=first,
17             mode='a',#append data to CSV
18             chunksize=chunksize)
19             df = pandas.concat(fileTree)
20             df['words'] =  getDataFrameWords(df)
21             rows = list()
22             for row in df[['path', 'words']].iterrows():
23                 r = row[1]
24                 for word in r.words:
25                     if len(word) > 0 and word.isalpha():
26                         rows.append((r.path, word.lower()))
27
28             words = pandas.DataFrame(rows,columns=['path','word'])
29             counts = words.groupby('path')\
30             .word.value_counts()\
31             .to_frame()\
32             .rename(columns={'word':'n_w'})
33
34             word_sum = counts.groupby(level=0)\
35                 .sum()\
36                 .rename(columns={'n_w': 'n_d'})
37
38             TF = counts.join(word_sum)
39
40             TF['TF'] = TF.n_w/TF.n_d
41             TF = TF.sort_values('path')
42             TF.to_CSV(filepath + "\\TF.CSV",
43             header=first,
44             mode='a',
45             chunksize=chunksize)
46
47             IDFTemp= words.groupby('word')\
48             .path\
49             .nunique()\
50             .to_frame()\
51             .rename(columns={'path':'i_d'})\
52             .sort_values('i_d')
53
54             if first :
55                 IDF = IDFTemp.copy()
56
57             else:
58                 IDF = pandas.concat([IDF, IDFTemp],sort = False)
59                 IDF['i_d'] = IDF.groupby('word')['i_d'].transform('sum')
60                 IDF = IDF[~IDF.index.duplicated(keep='first')]
61
62             fileTree = list()
63             filePathDictionary = list()
64             first = False
65
66             with open(file.resolve(), encoding='uTF-8') as f:
67                 fileTree.append(pandas.DataFrame({'path': document_amount,
68                 'lines':f.readlines()}))
69                 filePathDictionary.append(pandas.DataFrame({'path': file.resolve(),
70                 'index': document_amount},index=[document_amount]))
71         contador += 1
72
73
74     IDF['IDF'] = np.log(contador - 1/IDF.i_d.values)
75     IDF.to_CSV(filepath +  "\\IDF.CSV")
76
77     first = True
78     for chunk in pandas.read_CSV(filepath +  "\\TF.CSV", chunksize=chunksize):
79         TF_IDF = pandas.merge(chunk, IDF, on='word', sort=False)
80         TF_IDF['TF_IDF'] = TF_IDF.TF * TF_IDF.IDF
81         compact = TF_IDF.drop(TF_IDF.columns[[2,3,4,5,6]], 1).sort_values('path')
82         compact.to_CSV(filepath +  "\\TF-IDF.CSV",
83         header=first,
84         mode='a',#append data to CSV
85         chunksize=chunksize)#size of data to append for each loop
86         first = False
87
88     endTime = time.time()
89     os.remove(filepath + "\\TF.CSV")
90     timeTaken = endTime - startTime
```

D. Looking for most similar file

In order to look for the most similar file, the first step is to generate the TF-IDF for the query. This is necessary so there is a comparison point with the vectors generated by the next step of the algorithm. The TF-IDF looks for the IDF of the document index, if the words of the query do not appear on the IDF document, then they are ignored because they are not found in any document. The next step is loading in a DataFrame a chunk of data from the CSV containing all the TF-IDF files. When this DataFrame is loaded, a proyection of the queries word is done over each file of the chunk, to obtain a vector similar to that of the query, these are ordered in alphabetical order, then cosine similarity is applied between vectors from each file in the DataFrames and the vector produced from the query. The files that contain all words are stored in a list with its index path and the cosine similarity result. After all the files are evaluated the chunk is cleared and a new chunk of data begins to be evaluated, this process repeats until all the files are processed. Files that do not contain all the relevant words in the query are disposed to make the search way more efficient. After all files are processed the results stored are converted to a DataFrame and sorted in ascending order by the cosine similarity index and then inserted into a listbox to be shown to the user.

1) Cosine Similarity Query Time Complexity: Let K be the size of two arrays, dot-product multiplies each vector's value in a position with it's respective value in the second vector, therefore it takes $O(K)$, then, calculating the norm of the vectors takes K powers of two, $K-1$ sums, and 2 square roots, with one extra multiplication for the value of both results. This means that the algorithm takes linear time, $O(K)$. Since the module of numpy is written in C, these operations are extremely quick. Query searches are projections, sample selections, merges of the TF-IDF files, and comparisons using Cosine Similarity, we know that the cosine similarity takes K, where K is the size of the vectors, and to sample takes K to merge takes K^2, so the talking the $max(K, K^2)$, we get that for each file, the operation which takes the longest is merging, hence, it takes $O(n \cdot K^2)$. In reality, not all files contain all valid words in a query, and those that don't are disposed, hence it should take much less time.

Algorithm 2. Implementation of Query Search.

```
def searchQuery(querystr):
    contador = 0
    chunksize = 1000000
    words = querystr.split(' ')
    vectorized = vectorizeQuery(querystr,words)
    words = vectorized[vectorized.columns[0]].values
    wordDf = pandas.DataFrame(words, columns=['word'])
    size = wordDf.shape[0]
    queryVector = vectorized[vectorized.columns[-1]].values
    path = ''
    errorPercentage = int(parent.slider.get()) / 100
    first = True
    porcentaje = 0
    if(queryVector.all()):
        for chunk in pandas.read_csv(filepath + "\\tf-idf.csv",
        chunksize= chunksize,  index_col = 0):
            answer = list()
            sample = chunk.sample(int(chunk.shape[0] * errorPercentage))
            sample = pandas.merge(sample, wordDf, on='word', sort=False,how='inner')

            sample = sample.groupby(['path'])

            for pathNum, tfidf in sample:
                contador += tfidf.shape[0] * porcentaje
                if(tfidf.shape[0] == size):
                    column = tfidf[tfidf.columns[-1]].values
                    number = Cosine_Similarity.cosine_similarity
                    (column,queryVector).item()
                    answer.append((pathNum, number))

            answerDataFrame = pandas.DataFrame(answer,
            columns=["index","similarity"]).dropna()
            answerDataFrame = answerDataFrame
            [~answerDataFrame.similarity.duplicated(keep='first')]
            answerDataFrame.to_csv(filepath + "\\queryResult.csv",
            header=first,
            mode='a',#append data to csv
            chunksize=chunksize, index=False)
            first = False
```

Algorithm 3. Implementation of Cosine Similarity.

```
def cosine_similarity(v1,v2):
    return dot(v1, v2)/(norm(v1)*norm(v2))
```

IV. Experiments

A. Real Time Difference on Indexing With Growing Amount of Files

The time taken by the program to index the files can change depending on the amount of files. In this experiment we increased gradually the amount of files to be able to determine how the real time taken to index the files increases. At which rate it increases and how does this affect the efficiency of the program. The real time grows based on the amount of files in a polynomial way. This experiment is very useful to help determine how long it takes to index certain amount of files, this way the indexing of files can be previously begun with the necessary anticipation time.

1) Results: The results show a proportion of growth similar to the function obtained from the complexity analysis. In order to have a standard word amount for all file, an average review length of 150 words was selected to calculate the theoretical time.

Fig. 1. The X axis represents the amount of files indexed. The Y axis on the left represents the theoretical time growth. The Y axis on the right represents the real time obtained.

2) Analysis: Here the amount of files and the size of the files generates a big impact on the time, while the time does grow in a way similar to that predicted by the time complexity analysis, in reality the constant by which the time grows proportionally is big, hence when using 3200000 files, it took 14 hours to process. This shows that even when times grow at similar rates, this doesn't necessarily mean it is "quick" or "slow".

B. Comparison Between Real Time and Theoretical Time on Query Searches

1) Results: The results show a proportion of growth similar to the function obtained from the complexity analysis. In order to have a standard word amount for all file, an average review length of 150 words was selected to calculate the theoretical time. Different query sizes where tried, these will be analyzed further later.

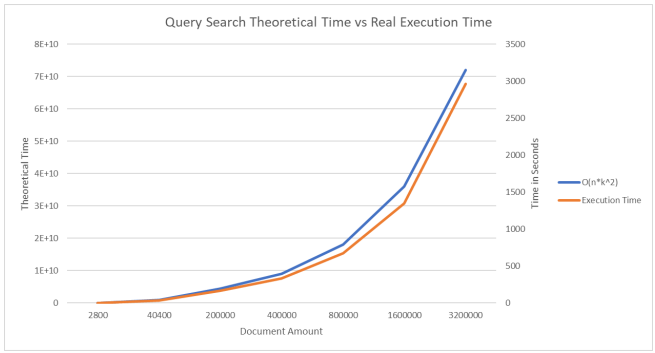

Fig. 2. The X axis represents the amount of files searched. The Y axis on the left represents the theoretical time growth. The Y axis on the right represents the real time obtained.

2) Analysis: As explained previously, the theoretical time changed in a proportion similar to the real time, but as seen, even when both functions are upper bounded by the same function ($O(n \cdot k^2)$), in reality, the query search takes up to 94.44 percent less time, when comparing the time of searching over the files, to the time it takes to index them. This is because in order to index a file, all words must be processed, and for each word, a process occurs to account it on the IDF. This ends

up showing how two functions that theoretically are equivalent can be totally different in real time.

C. Varying Search Precision Analysis

1) Probabilistic Approach: In order to search through data faster, a probabilistic approach is needed, in this case, the approach is based on a percentage between 0 and 100, the user selects the percentage of words of all documents to search through, and utilizing the sample function in the Pandas library, a percentage of arbitrary elements is selected from the TF-IDF file, which means that there are less words to iterate through, hence it is faster but the possibility of not finding a document that meets the characteristics of the search exists. Three different queries will be used, noting the time taken, and the amount of results provided, looking for an average time-precision relation.

Fig. 3. The X axis represents the percentage of precision. The Y axis on the left represents the execution time. The Y axis on the right represents the amount of results obtained.

Fig. 4. The X axis represents the percentage of precision. The Y axis on the left represents the execution time. The Y axis on the right represents the amount of results obtained.

2) Probabilistic Approach Results: In the specific search, once the precision went under 100% , no results where obtained, therefore, when doing specific searches, high percentages must be used. Using looser terms which appear in various reviews, however, resulted in an almost linear relation between document amount and precision percentage.

3) Probabilistic Approach Results Analysis: Given the average results obtained when using both loose terms and specific terms for queries, it can be seen that using looser terms implies that a higher amount of documents will be selected, therefore, some of these can be sacrificed in order to get a better search time. A sweet spot for search precision percentage was of 40% , where execution time dropped 37% , compared to the 100% precision result, while obtaining 29% of the original files, which means that a good amount of files was obtained in two thirds the time, which is desirable. Further down, search result amount percentage dropped to below 10% and execution time did not decrease considerably. Therefore, dropping below 40% precision sacrifices more than what is gained, as seen in the increasing gap between the curves in Fig. 3.

4) Varying how Specific Queries are: Given that files which do not comply with the whole query are not further processed, utilizing specific queries should make search time faster, having to process less files. Hence, different levels of detail will be used in searches and the theoretical time will be compared. To determine how specific a query is, the Document Frequency will be used, those words with a smaller Document Frequency will incur in less documents and hence produce a more specific search, resulting in less time.

5) Varying Query Specificity Results: As expected, higher specificity (defined by the amount of documents that a word appears in), yields less results, and thus a smaller processing time.

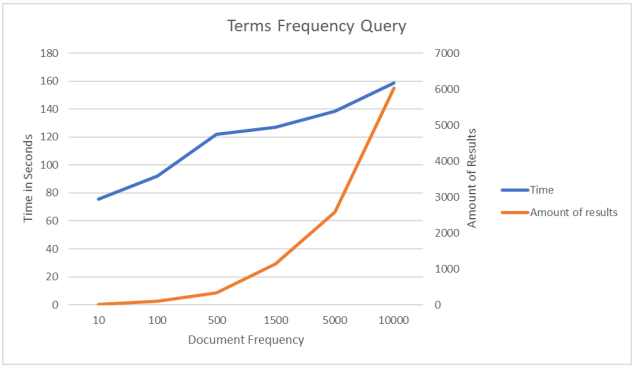

Fig. 5. The X axis represents the amount of documents a word appears in. The Y axis on the left represents the execution time. The Y axis on the right represents the amount of results obtained.

6) Varying Query Specificity Results Analysis: The queries are conducted over 200000 files, so the amount of files searched does not change, however, the time it takes to iterate over all of these files is inversely proportional to how many documents are compatible with what the query specifies. Therefore, when using words that appear in less documents, a shorter execution time will be obtained, the results of time saving where in fact similar to those obtained using the probabilistic search. Knowing that probabilistic search works the best with loose terms, and that stricter terms result in shorter execution time, a function can be programmed to weight how specific a query is, and the less specific it is then

the lesser the precision percentage used would be, this would result in overall consistent search times over a given data set, reducing the variability.

V. Conclusions

With all the data collection, experiments, and investigation several things can be concluded about this project. TF-IDF is a very useful technique for weighing words of a text in comparison with a big data set of other texts when along with DataFrames that provide the chance to process the data by chunks and storing them in secondary memory. Without the DataFrame data structures to help manage big data sets the process of indexing tends to be exponentially slower when the amount of data grows. Numpy helped a lot to improve the efficiency on algorithms like cosine similarity and IDF because of its built in operations for multidimensional array objects. When it comes to handling big amounts of data you need to be sure to have a data structure that will support the amount of data. Also you need to be careful not to have all of the data on main memory otherwise the program or application will become very slow and may be even crash for not having enough memory resources. To prevent this you need to chunk the data and work with the essential chunks at any given time to make sure you are making an efficient use of the computer resources. The last and more important conclusion is that when you are working with these amounts of data and files you need to be sure to store all calculations to make them only once like the TF-IDF that is needed every time that the user makes a query, the initial result of the TF-IDF of all the files has to be stored in secondary memory so that the program does not need to calculate the TF-IDF every time because it takes a lot of precious time and resources.

VI. Future Work

This project can be a basis for a lot of future papers or projects. It can be also improved to work faster for bigger data sets. Also we look forward to adapting this program to be able to work as a search engines for real web pages or maybe adapt it to serve as a search engine for an specific company or environments like digital libraries. This project can be used as a basis for the development of a new search engine and improved so it can compete with other established search engines. It can be compared with other type of search engines that used different algorithms than the ones used in this project to work as a comparison point on whether which algorithms work better for search engines. This would help other people know which algorithm works faster and how to take advantage on the benefits that certain libraries offer for this type of projects. The big data management used in this projects can be used to develop further applications that need to handle big amounts of data with limited memory and processor resources.

References

[1] "Euclidean vs cosine distance," Netherlands, 2017. [Online]. Available: https://cmry.github.io/notes/euclidean-v-cosine

[2] "Page rank," 2004. [Online]. Available: https://en.wikipedia.org/wiki/PageRank

[3] "Intro to data structures." [Online]. Available: https://pandas.pydata.org/pandas-docs/stable/dsintro.html

[4] "Amazon review data," San Diego, CA. [Online]. Available: http://snap.stanford.edu/data/amazon/productGraph/