DI SS 2023
Task 1

Vladimir Panin (12238686) Cani Besnik (12243760)
Andrei Florescu (12242987)

April 2023

1 Introduction

The purpose of this report is to present the results of applying basic MapReduce programming techniques to process large text data. In particular, the goal of this exercise is to select the most discriminative terms in the Amazon Review Dataset 2014, which contains 142.8 million reviews from 24 product categories. To achieve this goal, we will use MapReduce jobs to calculate chi-square values for the terms in the review dataset after performing tokenization, case folding, and stopword filtering. The output will be a file containing the top 75 most discriminative terms for each category according to their chi-square values.

2 Problem Overview

Since the Amazon Review Dataset 2014 is comparably large the file, a subset of this dataset called *reviews_devset.json* was used for local development. Each line in *reviews_devset.json* represents a review and contains the following entries:

```
reviewerID - ID of the author of the review
asin - string - unique product identifier
reviewerName - string - name of the reviewer
helpful - array of two integers [a,b] - helpfulness rating of the review:
                                        a out of b customers found the review helpful
reviewText - string - the content of the review; this is the text to be processed
overall - float - rating given to product asin by reviewer reviewerID
summary - string - the title of the review
unixReviewTime - integer - timestamp of when review was created in UNIX format
reviewTime - string - date when review was created in human readable format
category - string - the category that the product belongs to
```

An exemplary product review is the following:


```
{"reviewerID": "A3D8QKK1Z19XMM", "asin": "B00004R9VV", "reviewerName": "Craig S Fry",
"helpful": [1, 1], "reviewText": "Good price. Good product. I had read
some poor reviews of competitive products before purchasing this one,
so I was a little concerned. But the Flowtron turned out to be exactly what I was looking for.",
"overall": 5.0, "summary": "Satisfied", "unixReviewTime": 1383696000,
"reviewTime": "11 6, 2013", "category": "Patio_Lawn_and_Garde"}
```


In the preprocessing stage three steps had to be carried out. First of all the tokenization had to be be performed using a set of given delimeters like white spaces, tabs, digits, and special characters such as ()[].!? and many others. Next, the tokens must undergo case folding to ensure consistency. Stopword filtering must also be applied using the predefined stop word list *stopwords.txt*. in addition all tokens consisting of only one character had to be removed,

Once the data is preprocessed, chi-square values can be calculated for all unigram terms for each category. After calculating these values, the terms can be ordered according to their value per category, and the top 75 terms per category should be preserved. Finally, the lists should be merged over all categories to obtain the final result.

3 Methodology and Approach

The implementation used consists of two different MapReduce jobs. First, a MapReduce job was implemented to yield the count of a token in each category. In the second step the output is sorted on the basis of the calculated chi-value in a descending way for each category, so that the output is in the format:

```
<category name> term_1st:chi^2_value term_2nd:chi^2_value ... term_75th:chi^2_value
```

For that two dictionaries *categories_counts*, for storing the counts of the category, and *categories_tokens* , for storing the most discriminative tokens, were defined.

3.1 1. MRjob

The **map_words_categories** function takes in two parameters, the first parameter is ignored, and the second parameter is line, which is a string. The line parameter contains the full review dataset provided.

Within the function, each review is first extracted from the line parameter using the *splitlines()* method. Then, the review text is extracted from the JSON object and tokenized using a regular expression. The resulting list of tokens is filtered to remove stopwords, tokens with only one character and non-alphabetic characters such as !?,;:. The resulting set of tokens is then iterated over, and each token is yielded as a key-value pair. The key is a tuple consisting of the review's category, the token and the value 1 as a count.

After all tokens for a given review have been processed, the function yields the category together with the string 'category_count ' and the value 1, indicating that a new review has been found for that category.

The **combiner_count_words(self, category, token)** method is the combiner of the first MRjob and the only combiner used in the project. It output of the **map_words_categories(self, _, line)** mapper function and sums the counts for each (category, token) pair and then yields it.

The **reduce_count_categories(self, category, token)** method is the first reducer for the MRJob. It takes a category and token pair as input and sums the count for each category and token pair. It stores the counts to the variable *totalCount* and then yields the token and the *totalCount* variable

An exemplary row of the output of the first MRjob is the following.

```
["Patio_Lawn_and_Garde", "nice"] 1
```

This represents that the token "nice" appears one time in the category "Patio_Lawn_and_Garde"

3.2 2. MRjob

The task of the second MRjob is to calculated the chi-values for each category and give out the 75 most discriminative terms for each category. This MRjob consists of a mapper **second_mapper** and a reducer **second_reducer**.

The **second_mapper** method is responsible for processing the input data and generating key-value pairs for the second round of MapReduce. It takes a key-values pair as an input and the yields

again a key-value pair, where the key is None and the value is a tuple containing the category, the token and the respective count.

The **second_reducer** method takes the intermediate key-value pairs generated by the **second_mapper** and adds the respective count for each token in each category to the dictionary *categories_tokens*. After that, a string with the tokens and their chi-squared values for the corresponding category using the **calculate** method is yielded.

The **calculate()** function is responsible for calculating the chi-squared statistic for each token in each category, sorting the tokens by their chi-squared value, and yielding the top 75 tokens per category along with their respective chi-squared values. For that the methods **calculateChi()** and **sortTokens()** are used and then the category along with the aggregated string with the 75 most discrimative terms is yielded.

The **calculateChi()** method calculates the chi-squared value for each category and token pair. The chi-squared value is a statistical measure of the independence between two variables, and in this case, it is used to determine the importance of a token for each category. The method first sums up the number of documents in each category to get the total number of documents N. Then, for each category and token, it calculates A, the number of documents in the category that contain the token, B, the number of documents not in the category that contain the token, C, the number of documents in the category that do not contain the token, and D, the number of documents not in the category that do not contain the token. Using these values, it then calculates the chi-squared value using the formula $\frac{N*((A*D)-(B*C))^2}{((A+B)*(A+C)*(B+D)*(C+D))}$. Finally, it stores the chi-squared value for each category and token pair in a dictionary called categories_chi.

The **sortTokens()**: This method sorts the tokens for each category based on their chi-squared value in descending order. It first sorts the tokens for each category based on their chi-squared value using the sort() method with a custom lambda function. It then truncates the list to 75 tokens if it has more than that, and sorts the tokens alphabetically using another lambda function.

An exemplary output of the second MRjob is:

```
"Patio_Lawn_and_Garde" "" acre:351.2030 ants:182.9011 bait:313.3229
bbq:449.2898 bilt:313.2352 bird:194.3341 ... "
```

3.3 Pipeline

The pipeline diagram for both first MRjob and the second MRjob is given in the figure 1.

4 Conclusion

In conclusion, the code ran successfully on the Hadoop system with a run time of 24 minutes. The process of developing the code and implementing it on Hadoop also provided valuable insights into some technicalities of distributed computing and the importance of optimizing performance.

Especially, a appreciation was gained towards the need for optimizing performance. Multiple previous versions of the code, were working for local local testing, but were failing on the Hadoop Cluster by throwing a java.lang.RuntimeException during the mapping stage. After encountering these multiple Exceptions, multiple solution were explored. Finally, the solution turned out to be to define a global variable *categories_tokens* and accessing it inside the **calculateChi()** method.

All in all, the projected was completed successfully and some basic principles of MapReduce were implemented and were able to be executed on the cluster provided.

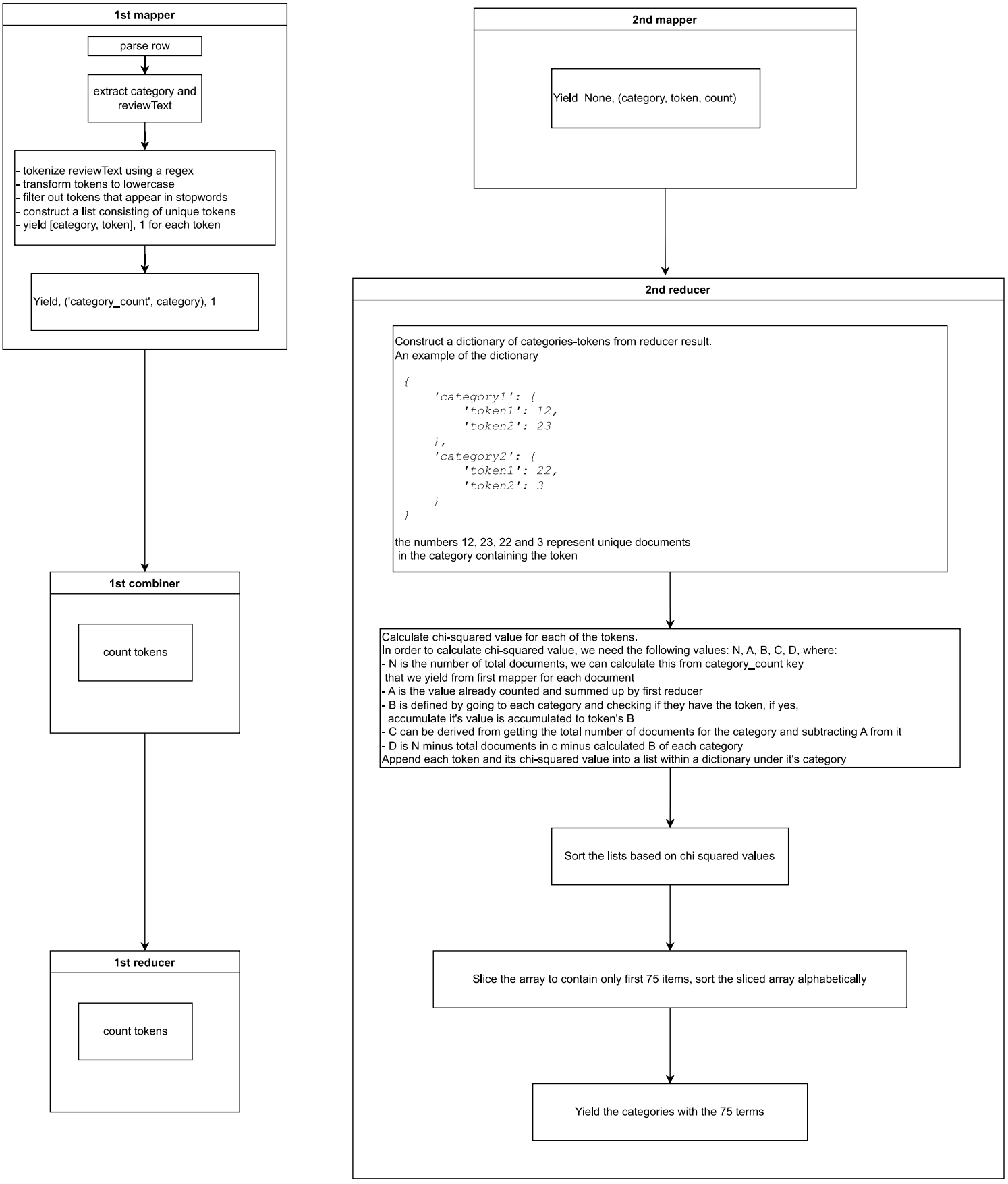

Figure 1: Pipeline