MapReduce to calc ChiSquare-value of Amazon Reviews corpus

Alexander Lorenz
Vienna University of Technology

Tobias Grantner
Vienna University of Technology

Maximilian Kleinegger
Vienna University of Technology

1 Introduction

This report presents a challenge that requires writing a MapReduce program to address. Simply put, we aim to compute and retrieve the top 75 terms for each category from all Amazon reviews. The report includes a description of the problem, detailed information about it, the approach taken to solve it, and a conclusion on the topic.

2 Overview

This section explains the problem we want to solve with a MapReduce program. Our goal is to identify terms that effectively differentiate between different categories of reviews. We will accomplish this by implementing MapReduce jobs to compute chi-square values for terms in the review dataset[1]. The preprocessing steps include tokenization into unigrams, case folding, and stopword filtering using a predefined list. Additionally, tokens consisting of only one character will be filtered out. The MapReduce jobs will efficiently:

1. Calculate chi-square values for all unigram terms within each category.
2. Rank the terms based on their chi-square values and retain the top 75 terms per category.
3. Merge the ranked term lists across all categories for further analysis.

3 Methodology and Approach

In this section, we outline how we approached the problem and describe the steps we took to solve it. We implemented the MapReduce program in Python using the MRJob [2] library. The program consists of three MapReduce steps, which are visualized in Figure 1.

The first step pre-processes the input and calculates the number of documents in a category that contains a term $n_{t,c}$ and the number of documents in a category n_c for each category c and term t. It consists of a mapper, a combiner and a reducer. The mapper parses each line of the input file, extracts the category and the review text, and tokenizes the review text into unigrams by splitting, case folding and filtering out duplicates, stopwords and single-character tokens. The mapper emits a key-value pair for each term in the review, with the key being a tuple of the category and the term, and the value being 1. Additionally, the mapper emits one further key-value pair, with the key being a tuple of the category and `None`, and the value being 1. It is used to count the number of documents in each category n_c, which cannot be done by adding up $n_{t,c}$ for all terms, since a document can contain multiple terms and would result in documents being counted multiple times. The combiner and reducer are then used to aggregate the document counts. The result of the first step is a key-value pair for each term in each category, with the key being the term, and the value being a tuple of the category and the number of documents.

In the second job, we compute the number of documents across all categories that contain a term n_t and the total number of documents n. In the case of the term given in the key of the input not being `None`, the values contain the category and $n_{t,c}$ for the corresponding category. By summing up

[1] `https://cseweb.ucsd.edu/~jmcauley/datasets/amazon/links.html`
[2] `https://mrjob.readthedocs.io/`

the $n_{t,c}$ values over all categories, we get the number of documents across all categories that contain the term n_t. On the other hand, if the term is `None`, the values contain the category and n_c for the corresponding category. By summing up the n_c values over all categories, we get the total number of documents n. The second step emits a key-value pair for each category the term occurred in, with the key being the category, and the value being a tuple of the term, $n_{t,c}$ and n_t. For the term `None`, the value is a tuple of `None`, n_c and n.

The third job calculates the number of documents across all terms that are in a category n_c, calculates the chi-square values for each term in each category, ranks the terms based on their chi-square values and retains the top 75 terms per category. It again consists of only a reducer, which receives $n_{t,c}$, n_t, n_c and n for each term in a certain category. With these values, we can calculate the chi-square value for each term using the following formula:

$$\chi^2 = \frac{n \cdot (A \cdot D - B \cdot C)^2}{(A+B) \cdot (A+C) \cdot (B+D) \cdot (C+D)}$$

$$\begin{aligned}
n &= \text{total number of documents} \\
A &= n_{t,c} \text{ ...number of documents in } c \text{ which contain } t \\
B &= n_t - A \text{ ...number of documents not in } c \text{ which contain } t \\
C &= n_c - A \text{ ... number of documents in } c \text{ without } t \\
D &= n - A - B - C \text{ ...number of documents not in } c \text{ without } t
\end{aligned}$$

The reducer then ranks the terms based on their chi-square values and retains the top 75 terms per category. The result of the third step is a key-value pair for each category, with the key being the category, and the value being a list of the top 75 terms for the category.

What is now left to do is to bring the results per category in the form of `<category name> term_1st:chi^2_value term_2nd:chi^2_value ... term_75th:chi^2_value`, sorted by the category name, and create a dictionary of all terms occurring in the top 75 of any category. These steps could have been implemented in a fourth MapReduce step, collecting all results under a `None` key in the reducer. However, we decided to implement these steps in the Python script calling the MapReduce job, since we have to collect all resulting data in memory anyway.

4 Conclusions

Overall, this assignment proved quite beneficial for gaining a fundamental understanding of MapReduce programming techniques tailored for processing large text corpora and large amounts of data in general. As each of us came up with our own solutions initially, it was interesting to observe how one decomposed the problem domain into various stages, employing different alignments of key-value pairs. Nevertheless, during the local execution of map reduce tasks, we obtained identical chi-squared solutions, yet encountered minor variations in the ordering when values were equal. To combat this issue, we decided to sort the results of each category by term in ascending order when chi-squared values are equal, resulting in identical results of our solutions.

At last, we focused on improving the runtime efficiency. Techniques such as iteratively yielding key-value pairs, constructing keys consisting of tuples, and introducing a combiner were employed to achieve this objective, to lower reducer size, replication rates, and network costs.

Depending on server usage, we managed to achieve a runtime of approximately 25 minutes on the full dataset.

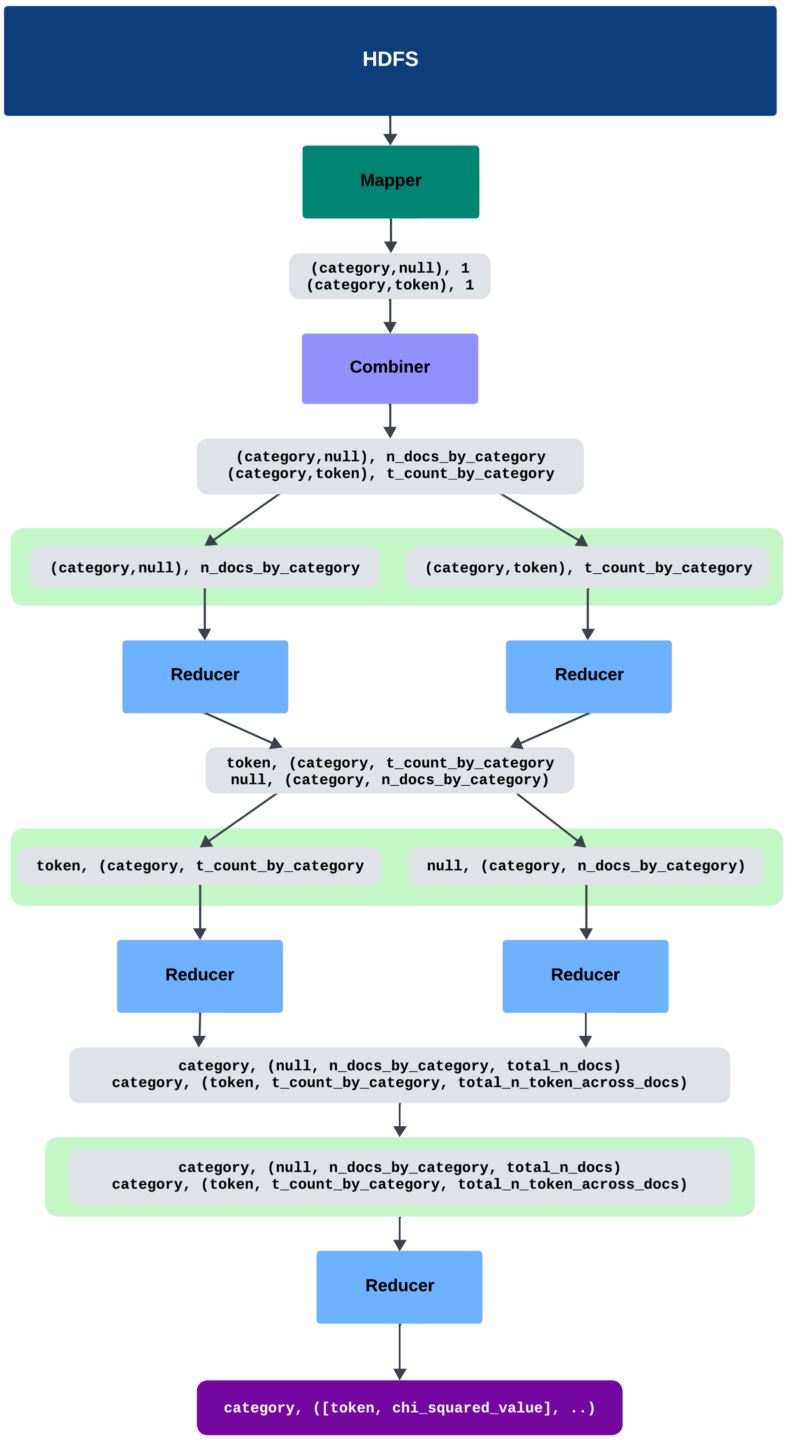
HDFS
Mapper
(category,null), 1
(category,token), 1
Combiner
(category,null), n_docs_by_category
(category,token), t_count_by_category
(category,null), n_docs_by_category
(category,token), t_count_by_category
Reducer
Reducer
token, (category, t_count_by_category
null, (category, n_docs_by_category)
token, (category, t_count_by_category
null, (category, n_docs_by_category)
Reducer
Reducer
category, (null, n_docs_by_category, total_n_docs)
category, (token, t_count_by_category, total_n_token_across_docs)
category, (null, n_docs_by_category, total_n_docs)
category, (token, t_count_by_category, total_n_token_across_docs)
Reducer
category, ([token, chi_squared_value], ..)

Figure 1: MapReduce flowchart