Project Report Exercise 1

Stefan Öhlwerther
TU Wien
11809642
Vienna, Austria

Tobias Raidl
TU Wien
11717659
Vienna, Austria

Samo Kolter
TU Wien
11810909
Vienna, Austria

1 INTRODUCTION

Exercise 1 of the Data-intensive Computing course revolved around applying basic MapReduce programming techniques to process large text corpora. The goal of this assignment was to select terms that discriminate well between categories in the Amazon Review Dataset 2014. The assignment required preprocessing (tokenization, case folding, and stopword filtering) to obtain terms occurring in the review texts, and then writing MapReduce jobs to calculate chi-square values for each combination of term and category in the review dataset. The top 75 most discriminative terms per category had to be obtained and stored in a file. In this report, we present a MapReduce solution using the `mrjob` Python package, which achieves a runtime of well under 15 minutes for the full dataset (assuming the TU Wien LBD cluster is not overloaded).

2 PROBLEM OVERVIEW

As we learned in the lecture, the Chi-Square metric χ^2 is a measure of the dependence between categorical stochastic variables. In the context of Natural Language Processing, we can use $\chi^2_{t,c}$ to measure how strongly the appearance of a certain term t in a collection of documents depends on a given category c. What this means in practice is that terms producing a noticeably larger χ^2 value can be considered 'characteristic' for a particular category. By computing the metric for every combination of category and term, aggregating by category, and then ordering the values for every term within a category from largest to smallest, we obtain lists of χ^2 values for every category. Within each list, the first k terms corresponding to the first k χ^2 values are then considered to be the k most characteristic terms for that category.

For a collection of documents, we can compute $\chi^2_{t,c}$ for every term t that occurs in any of the documents and category c a document may be associated with, like so:

$$\chi^2_{tc} = \frac{N(AD - BC)^2}{(A+B)(A+C)(B+D)(C+D)}$$

$A \dots$ number of documents in c which contain t
$B \dots$ number of documents not in c which contain t
$C \dots$ number of documents in c without t
$D \dots$ number of documents not in c without t
$N \dots$ total number of retrieved documents

In this exercise, we should use the Amazon Review Dataset 2014 to obtain the top 75 terms that distinguish the categories of Amazon products. The dataset is a text file, where every line contains a product review in JSON format. For computing the χ^2 metric, the `reviewText` and `category` properties of each review are relevant. The former represents a document from the *corpus* of documents, while the latter contains the category the document is associated with.

As the dataset contains 142.8 million reviews from 24 product categories and its total size is 56 GB, smart distributed computing across several machines is required to complete the task in a reasonable time.

3 METHODOLOGY AND APPROACH

To complete the task, a combination of two MapReduce jobs is used. The first one is merely concerned with writing the number of documents per category to a file. This file is then read by the second, 'main' job that computes the χ^2 values. Finally, we parse the output of the job and write it into a file. This file then contains the desired result.

In the following paragraphs, we describe the steps in more detail. A visual depiction of the data flow and key-value pair design that was used can be found on the last page of this report.

3.1 Category Counter Job

The mapper inputs are lines of the dataset, where each line represents one document in JSON format. The `category` attribute is extracted and forwarded. The mapper emits the key-value pair.

$$< category, 1 >$$

The combiner receives these key-value pairs as input and builds the sum over each value to return

$$< category, \#\ occurrences >$$

This step reduces the size of intermediate results that are consequently sent to the mappers. Note that those values do not yet represent the final result because combiners run locally on a data node and therefore can only count the categories of the documents that are provided locally. To arrive at the final result, the reducers execute the same task, but with the output of the combiners. This leads to the correct number of categories across all documents. The output of the mappers is then written to a JSON file that is used by the job for the computation of the χ^2 values.

3.2 Chi Square Job

The job for computing the χ^2 values uses the same review dataset as input as the previous task. In addition, its mappers use the `stopwords.txt` file, while the reducers require the `category_counts .json` from the previous job.

3.2.1 *Mapper.*

Each mapper again processes a line containing review data in JSON format. First, the `reviewText` and `category` fields are extracted from the review data. The review text is then tokenized into terms, by separating it at any character (or sequence of characters) that is (are) neither alphabetical nor '<', '>', '^', or '|'.

We assume that this approach is equivalent to tokenization using whitespace, tabs, digits, and a set of punctuation characters and other special characters (expected in the exercise description). The code we were given for exercise 0 uses the same approach (with a comment explaining that it does exactly the kind of tokenization that we need for this task), so we just copied the regex that was used there.

Next, terms that are included in the stopword list from `stopwords .txt` are removed, as well as those with a length of 2 characters or less. The key of the output is a term, and the value is a tuple, containing the documents category and 1, i.e.

$$< term, (category, 1) >$$

3.2.2 Combiner. The combiner, for each term, counts the number of distinct categories of documents it appears in. The input is the output of the mapper. This is still done by the mapper itself and therefore does not provide the complete results, but again takes the load off the reducer to do as much computation as possible in parallel. The output is

$$< term, (category, count) >$$

3.2.3 Reducer. In the reducer, we finally have all the data we need to calculate χ^2 values. (see the formula above). They are obtained like so: First, the outputs from the previous stage are aggregated together (in a dictionary) to obtain the number of documents containing the term for each of the categories it appears in. At the same time, we count the total number of documents containing the term n_t. Now, we just need to iterate over the pairs of $category, count$ in the dictionary and can calculate the A, B, C, and D N components required for the computation of χ^2_{ct}. A is then simply $count$ while B can be computed by subtracting $count$ from n_t. Every reducer also accesses the data from $category_counts.json$. From this, we first obtain the number of documents in the current category to calculate C, and then we also read N to get D and finally χ^2_{ct}.

The key-value pair format that is yielded in the aforementioned loop in the reducer is

$$< None, (category, \chi^2_{ct}, term) >$$

None is emitted as the key, as this means that results will be collected by a single reducer. This makes sense in our case, as the outputs are relatively small and splitting across different reducers would introduce unnecessary overhead. The value is a tuple containing the term, category, and the χ^2 result for this specific combination.

3.3 Obtaining final result

To get the data we are interested in, we now only need to iterate over the produced key-value pairs, where only the values are of interest as the key is `None` for all of them. As we iterate over the values, we add items to a dictionary whose values are lists for each category. As those lists should be ordered by χ^2 values, we make use of priority queues (available in Python in the `heapq` package), keeping only the top 75 values and associated terms (dropping the smallest one every time a new value arrives). This should be better than pushing all χ^2 values with the associated terms into lists per category and sorting each of them, both in terms of memory and runtime efficiency, as in our case the number of top terms is significantly smaller than the total number of terms. This approach is also suggested in option 2 in this Stack Overflow answer. In any case, after this step, we end up with the top 75 terms and have also accumulated the unique terms and can order them alphabetically. We write those results into a text file with the following content:

(1) One line for each product category (categories in alphabetic order), that contains the top 75 most discriminative terms for the category according to the χ^2 test in descending order, in the following format:
`<category name> term_1st:chi^2_value term_2nd:chi ^2_value ... term_75th:chi^2_value`
(2) One line containing the merged dictionary (all terms space-separated and ordered alphabetically)

4 CONCLUSION

Although our proposed solution has to read each line of the review dataset twice, i.e., once for each job, this approach proved more efficient both wrt. execution time (<15 min) and memory usage.

In contrast, the first idea was to emit for each unique pair of term and value for one document to emit two tuples. For one using the key and for the other the category as key. With this information, we could, in the case of the category being the key, calculate A and C. On the other hand, with the term as the key, it was also possible to calculate A and additionally derive B. This alone proved to be very inefficient since the concept of having only minimal information where necessary was violated.

Hence, the approach described above was developed, significantly reducing the communication cost by omitting the yield statements with the category as key in the second job. To facilitate this change, we needed to precompute the number of documents per category, to calculate all the necessary variables needed for the chi-squared statistic. Another possibility would have been to precompute the number of occurrences of each term across all reviews. This was discarded early since the intermediate results would have been significantly larger due to the imbalance between distinct terms and categories.

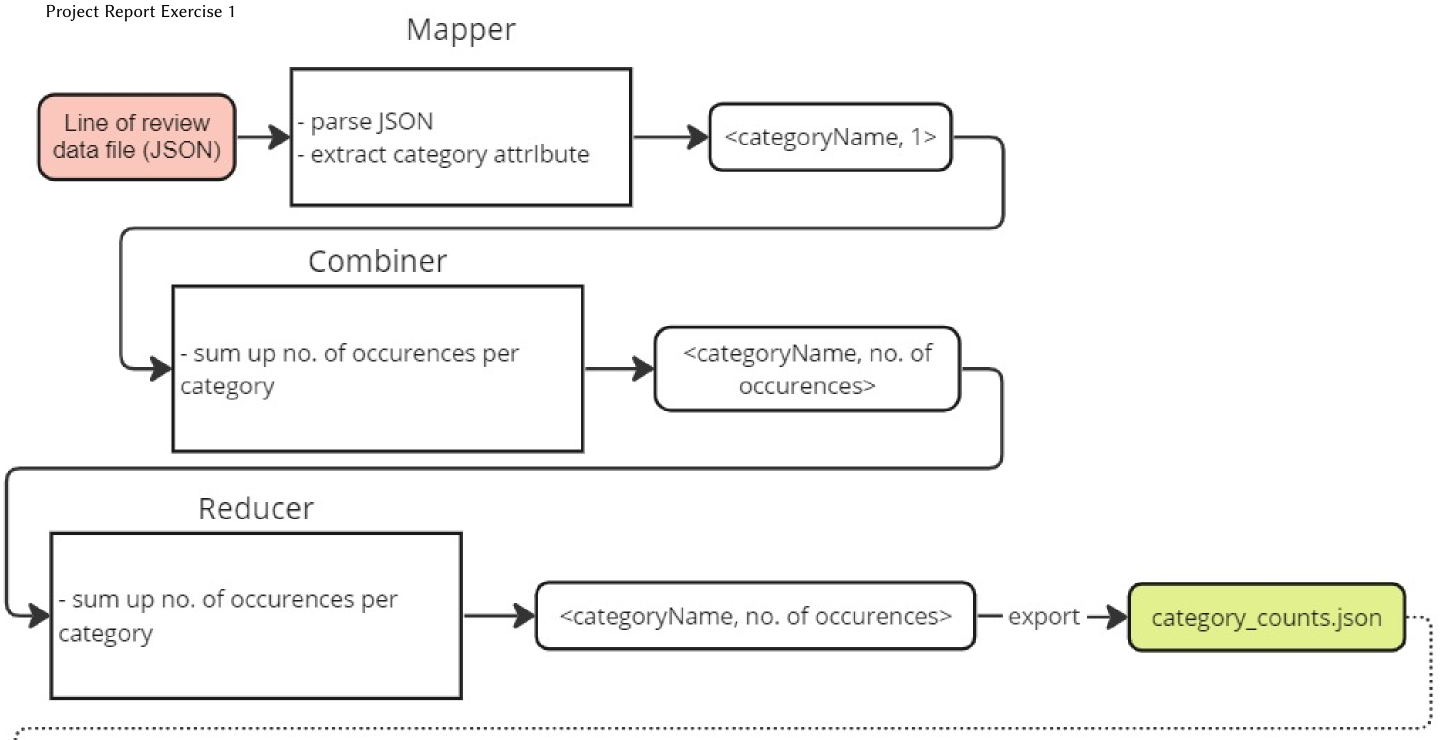
One major observation that allowed us to develop an efficient solution is using a dictionary to only emit unique terms per review. This allows us to assume in further steps that if a term occurs, that it won't occur again for the same review. Otherwise, we would have needed to encode the review ID to ensure that each document is only counted once for a specific term.

4.1 Additional Remarks, Learnings

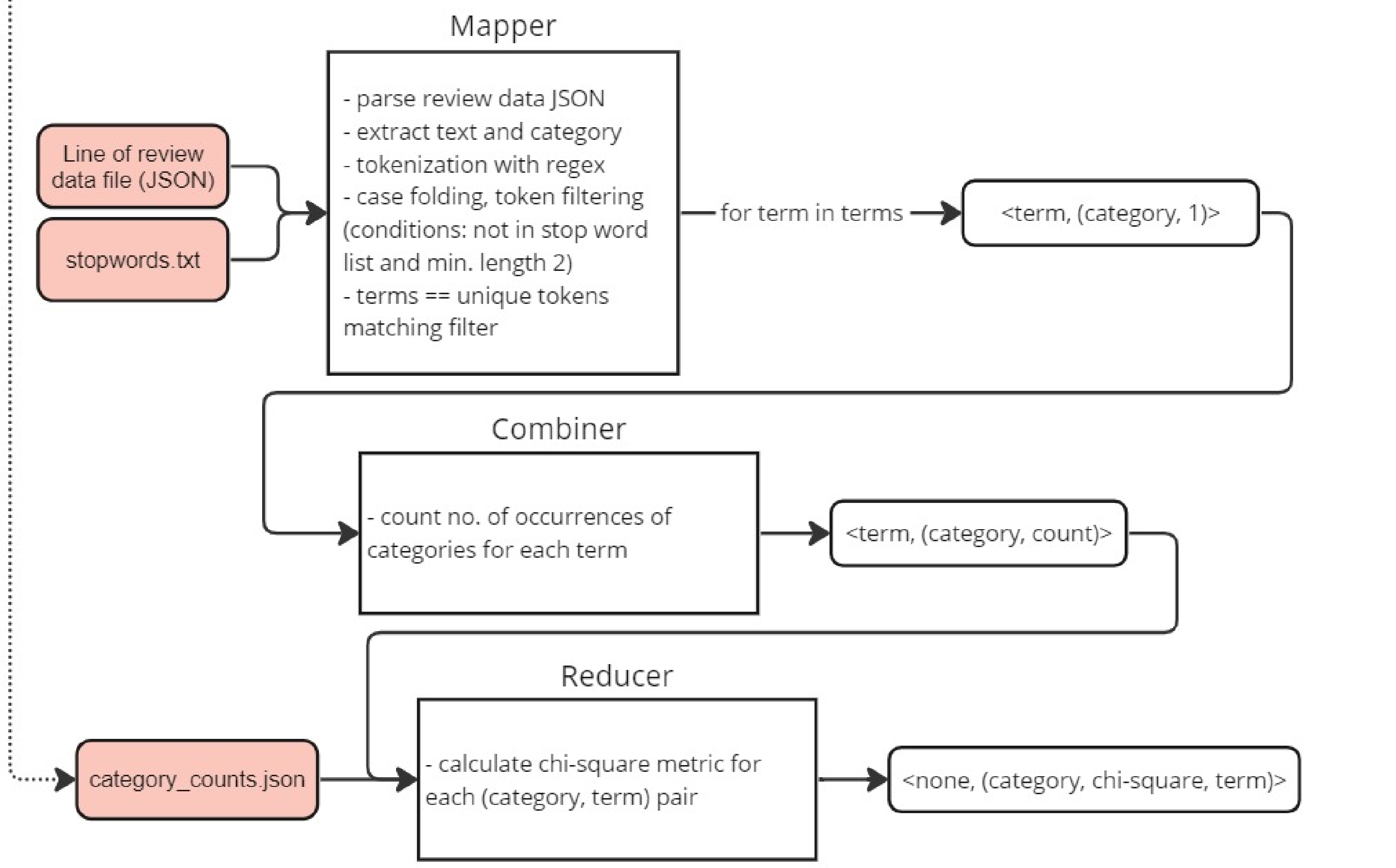
Finally, we want to describe our key learnings from this exercise. These are personal observations while developing a solution for the combined dataset, for which we did not find official sources in the documentation but proved to be necessary for the jobs to perform correct results on the combined dataset.

(1) combiners are not allowed to produce new keys. This is mainly because the combiner is not guaranteed to be executed for each individual mapper call, and more importantly, Hadoop plans the distribution of the keys before the execution of the combiner.
(2) Outputs of yield statements should be rather small. Specifically, value lists should contain basic data types only.

Job 1: Category Counter


Mapper
Line of review data file (JSON)
- parse JSON
- extract category attrIbute
<categoryName, 1>
Combiner
- sum up no. of occurences per category
<categoryName, no. of occurences>
Reducer
- sum up no. of occurences per category
<categoryName, no. of occurences>
export
category_counts.json

Job 2: Chi-Squared


Mapper
Line of review data file (JSON)
stopwords.txt
- parse review data JSON
- extract text and category
- tokenization with regex
- case folding, token filtering (conditions: not in stop word list and min. length 2)
- terms == unique tokens matching filter
for term in terms
<term, (category, 1)>
Combiner
- count no. of occurrences of categories for each term
<term, (category, count)>
Reducer
category_counts.json
- calculate chi-square metric for each (category, term) pair
<none, (category, chi-square, term)>