Data - intensive Computing Assignment 1

Text Processing Fundamentals using MapReduce

Tuvshin Selenge e11815791
Paul Hönes e12333412
Lukas Fitz e11815353
Felix Kapfer e12429669
Ameer Hamza e12448315

Link to GitHub Repository: 194048_dic_sose2025_a1

Introduction

In this assignment, we apply MapReduce programming techniques (using Python's **mrjob**) on the TU Wien Hadoop cluster to extract discriminative terms from the Amazon Review Dataset 2014. The dataset comprises 142.8 million reviews spanning **22** product categories and is available in HDFS under a shared directory.

To ensure accurate, scalable feature extraction, we first preprocess each review by:

1. **Tokenization** – splitting into unigrams
2. **Case folding** – lowercasing all tokens
3. **Stopword filtering** – removing all tokens in stopwords.txt

After validating our pipeline locally on a small development set, we run MapReduce jobs over the full HDFS input to compute the chi-square statistic for every unigram in each category. The chi-square measure highlights terms that are disproportionately frequent in one category versus the others. We then:

- rank all unigrams by their chi-square value per category,
- select the top 75 terms for each of the 22 categories, and
- merge these ranked lists into a single output.txt one line per category, plus a final line with the combined vocabulary in alphabetical order.

This produces a concise feature file that can be used for downstream classification or analysis on the most discriminative words per product category.

Problem Overview

In large-scale text classification, selecting a compact yet informative feature set is critical for both predictive accuracy and computational efficiency. In this assignment, we identify discriminative unigrams from 142.8 million Amazon reviews across 22 categories, ranking for each category those words whose presence best distinguishes its reviews from all others via the χ^2 test.

The input is a 56 GB JSON file in HDFS, each line a review with text, metadata, and a category label. Loading all data into memory and computing per-term statistics centrally would be infeasible in time and space. Instead, we distribute work across a Hadoop cluster using the Python “mrjob” library.

The challenges are twofold:

1. **Scalability & Efficiency**
   With over 140 million reviews, we must minimize network shuffle, avoid redundant work, and leverage combiners and in-mapper aggregation to meet our runtime target.
2. **Statistical Correctness**
   Computing χ^2 requires careful handling of zero counts and large marginals to avoid numerical instability. Precise global counts must be maintained and correctly joined in the second stage to ensure valid term rankings.

By chaining two MapReduce steps, first gathering distributed frequency counts, then computing and ranking χ^2 scores, we obtain a parallelizable, statistically sound workflow that yields a concise, high-quality feature set for downstream classification.

Methodology and Approach

1. Data Ingestion and Preprocessing

We begin by ingesting raw Amazon review data in JSON format. Each record contains metadata like reviewer ID, reviewer name, etc. and the review text. In the first MapReduce step, we need to tokenize the review text into individual terms and normalize them to lowercase. Further the punctuation, numerals and the given stopwords text file. To ensure consistent filtering across nodes, the stopwords file is distributed to each mapper using the mrjob's --file argument which allows local loading at runtime. To reduce the processing volume, each review's terms are de-duplicated at the mapper level so that each word contributes at most one count per review. We therefore maintain the Hadoop counters to record the total number of reviews processes as well as the count per product category. These values are later written to the counters.txt file.

2. Word Count MapReduce Pipeline

The Word Count MapReduce stage which is implemented by the wordCountJob.py and organized by the wordCountWrapper.py and uses the mrjob framework to facilitate both local and cluster execution.
The Mapper 1 reads each JSON line, extracts category and reviewText. Then it cleans and tokenizes the text, filters out stopwords and emits (word, category) → 1. It also increments two counters where one is for the total reviews and one is for the per-category calculation. The Combiner 1 locally aggregates counts for each word-category pair to minimize the input and output of the network. The Reducer 1 globally sums each word-category pair and returns the pair of category and total_count. The Final Reducer 1 puts all category counts for each word into a single dictionary, which produces the form super category - {category 1: count 1, category 2: count 2 and so on}.
At the end of this stage, the wrapper script collects the Hadoop counters into the counters text file which contains the two values of the total number of reviews and a JSON style mapping of per-category review counts.

3. Chi-Squared Feature Selection

In the Chi-Squared Analysis stage with the python script chiSquaredJob.py, we can identify the most discriminative terms for each product category using the χ^2 statistic.
In the mapper initialization function, mapper_init, we read the counters.txt to obtain the global review counts. For each word category-frequency dictionary, we compute a table for each category.

- A: Is the number of reviews in this category containing the word
- B: Is the number of reviews in other categories containing the word
- C: Is the number of reviews in this category not containing the word
- D: Is the number of reviews in other categories not containing the word.

The next step is to calculate

$$\chi^2 = N \times (A \times D - B \times C)^2 / ((A+B) \times (A+C) \times (B+D) \times (C+D))$$

where N is the total number of reviews.

Also important is to skip any zero-cell tables to avoid division errors and ensure statistical correctness and return category → word and χ^2. We also skip malformed entries and use counters to flag anomalies like those for debugging.

First the Mapper 2 emits the category, word and χ^2 pairs. The Reducer 2 in the second step gives us the outputs for each category with the dictionary of its top 75 discriminative words and in the end returns the union of all top words across categories. Then the final reducer produces the union of all selected words from all categories in the format:

- One line per category: Category → {word_1 :χ^2_1, word_2:χ^2_2, ….}
- And another line with the global vocabulary list which is sorted alphabetically.

4. Execution and Environment Configuration

For the execution we came up with a simple strategy. All scripts support two execution modes which are controlled by the environment variables. The first one is Local Mode (ENVIRONMENT = 0) which runs entirely on the local filesystem using the mrjob's inline runner. The Cluster Mode (ENVIRONMENT = 1) submits jobs to the Hadoop cluster via the streaming JAR. The resource parameters like memory for the map and reduce tasks are tuned in the main.sh to match the cluster capabilities. This shell script also organizes both MapReduce steps which ensures clean output directories, configures input/output paths (HDFS vs local environment) and passes the required files into each job and timestamps the final result for easy retrieval.

5. Implementation Details and Best Practices

- Robustness: Both jobs handle malicious input well as it skips invalid JSON or zero-division cases. This is necessary because otherwise it could create severe problems for the final output. There is an increment in internal mrjob counters to flag anomalies like missing stopwords file.
- Modularity: Shared utilities logging, argument parsing reside in the utils space and promotes code reuse and clarity.
- Scalability: Using the mrjob's combiner and Hadoop's distributed computation which enables linear scaling over the large review datasets which we need to use for the assignment.
- Reproducibility: Inline docstrings, consistent environment variable handling and explicit versioning of tools (otherwise they would not work) is also very important and ensures that others can reproduce and extend the analysis.

6. Runtime Performance and Output Summary

When we observe the console while it is running on the full 56 GB dataset we can see different messages:

- The raw intermediate size with a size of around 330 megabyte is reduced to the final output of around 63 KB.
- Furthermore, we can see the final execution time from the program start to the program end. The performance is around 24 minutes which is quite fast and optimized due to combiner usage and efficient token filtering with the knowledge that our first performance took about 40 minutes with the same result in the end.
- Regarding the output, we can say that the interesting content for us is in the final output part-00000 file after running the Chi-Squared job. We have the category name followed by a JSON style map of the top words per super category.
- Here is a little insight into the file. The first category looks like this:

```
Apps_for_Android    {
        'games': 2 537 104.82,
        'play': 2 247 334.64,
        'graphics': 1 713 251.91,
        'kindle': 1 623 395.54,
        ‘addicting': 1 242 808.22,
        …
        }
```

Those domain-specific words within the brackets are strongly associated with the “Apps_for_Android” which are ordered by the descending chi-score. These terms are therefore highly predictive for the android app category. For the other categories we can observe similar patterns.

Conclusion

In summary, we successfully demonstrated how MapReduce, in this case using Python’s mrjob framework, can be effectively used for processing and analyzing large text datasets. By applying a two-stage MapReduce approach to the Amazon Review Dataset 2014, which comprises 142.8 million entries, the 75 most discriminative terms for each of the 22 product categories were identified using the chi-square (X^2) statistic.

During the initial MapReduce stage, preprocessing techniques including tokenization, case folding, and stopword filtering were used to normalize the textual data. This was followed by local aggregation using combiners and in-mapper deduplication to efficiently calculate global and per-category frequency counts.

In the following stage, these aggregated counts enabled the calculation of category-specific chi-sqaured values, with handling zero-count scenarios to ensure numerical stability. The resulting output consists of ranked lists of the top discriminative unigrams per category as well as a comprehensive alphabetical vocabulary. The implementation of performance optimisations, including combiners, optimised resource allocation, and environment-controlled execution modes, resulted in a significant reduction of processing time for the full 56 GB dataset to less than 25 minutes. This represents an approximate 40% improvement over the initial implementation.