**Course: Data Intensive Computing**

**Text Processing Fundamentals using MapReduce**

**Group 16**

**Group Members:**

Naz Erum

# Introduction

This assignment applies the MapReduce paradigm to process a large-scale Amazon Reviews dataset to select terms that best distinguish product categories. By calculating chi-square (χ²) test to select the most discriminative unigrams from the 2014 Amazon Reviews dataset. Given the dataset’s massive size (~56GB), a distributed processing approach using MapReduce is employed to parallelize preprocessing, feature extraction, and feature scoring across multiple compute nodes.

The implementation leverages distributed processing using mrjob in Python and operates over both a development subset and a full-scale dataset.

## Goal of the Project

The primary goal of this assignment is to identify the top 75 discriminative unigrams per product category that are most associated with their respective categories in the Amazon Reviews dataset (2014 edition).

## Dataset Sizes

The dataset is provided as JSON lines with each review (document) containing fields such as reviewText, category, overall, and more.

* reviews\_devset.json (small): for testing and development.
* reviewscombined.json (56GB full set): for final evaluation.

# Problem Overview

The assignment involves designing a scalable system that:

* Preprocesses review texts by:
  + Tokenizing using specified delimiters,
  + Applying case folding,
  + Removing stopwords,
  + Filtering out single-character tokens.
* Computes per-category document frequencies for each token.
* Calculates the chi-square statistic for every (token, category) pair:

A black text on a white background

AI-generated content may be incorrect.

where A,B,C,D represent document counts and N is the total number of documents.

* Selects the 75 tokens with the highest chi-square values for each product category.
* Outputs:
  + Top 75 tokens per category (with scores),
  + A merged, alphabetically sorted dictionary of all selected terms.

The dataset is processed first on a smaller dev set (reviews\_devset.json) for validation, then on the full dataset (reviewscombined.json).

# Methodology

**Pipeline Overview**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Phase** | **MRJob / script** | **Purpose** | |  | | --- | |  |  |  | | --- | | **Main <key, value> emitted** | |
| Pre-processing and DF | MRDocFreq / mr\_doc\_freq.py | Tokenise reviews and count document frequency (DF) of every unigram per product category | ((token, category), count) plus two sentinel pairs to accumulate corpus-level *marginals* |
| Chi-square computation and Top-75 | MRChiSquare / mr\_chi\_square.py | Compute χ² for every (token, category) and keep the 75 best scoring terms per category | (category, (token, χ²)) |
| Orchestration | mr\_runner.py | |  | | --- | |  |  |  | | --- | | Run ① and ② sequentially, pass intermediate file paths, collect final output | | NA |

**Text pre-processing**

Every review line is parsed from JSON, lower-cased, split with the delimiter regular expression and filtered against the course stop-word list and single-character tokens. The helper tokenize() encapsulates these steps ​. A set() is used inside the mapper so that each document contributes at most one count per distinct token, which is exactly what document frequency requires.

* 1. **Phase 1 – Category-specific document frequency (mr\_doc\_freq.py)**
     1. Mapper\_init()

Loads the stop-word list once per task and initialises an in-memory defaultdict(int) named local\_counts to act as a *combiner-cache*. ​

* + 1. Mapper()
       - For each review it:   
         1. extracts the category;   
         2. creates a unique token set;   
         3. increments three counters:  
            • ('\_\_TOTAL\_\_', category) – number of documents in the category (needed later for C);  
            • (token, category) – **A** in the χ² contingency table;  
            • (token, '\_\_TOTAL\_\_') – corpus-wide token total (A + B).
    2. Mapper\_final()

Flushes the local cache so that *one* record per (token, category) reaches the shuffle.

* + 1. Combiner() and reducer()

Both intermediate (Snappy) and final (Gzip) outputs are compressed through the JOBCONF dictionary to keep network and disk I/O low

* 1. **Phase 2 – Feature selection with χ² (mr\_chi\_square\_2.py)**
     1. Side input loading - mapper\_init()

The DF file from Phase 1 is shipped via --dfcounts and parsed into three in-memory tables: A, A+B, and A+C

* + 1. Mapper()

Because all counts are already resident in RAM, the actual input split is ignored.

A, B, C, D and N are calculated and chi-square value is derived

* + 1. Combiner() and first‐step reducer()

A bounded max-heap (heapq.nlargest) retains only the 75 best-scoring tokens per category at each stage, minimising network traffic. The reducer also forwards every selected token once using a special key '\_\_\_TOKENS\_\_\_' so that all chosen features can later be merged into one global dictionary line ​.

* + 1. Dictionary merge

A second MRStep with a single reducer receives the formatted category lines and many ('\_\_\_TOKENS\_\_\_', token) pairs.

It passes category lines through unchanged and, for the special key, deduplicates plus lexicographically sorts the tokens before emitting

* 1. **Job orchestration (mr\_runner.py)**

# Conclusion

* 1. **Output Correctness**

The output meets all the requirements:

* Categories are listed alphabetically.
* Each category is followed by its top 75 tokens, ranked by descending chi-square score.
* Merged dictionary contains all selected terms sorted alphabetically.
* Preprocessing steps (tokenization, case folding, stopword removal, short token filtering) are correctly applied.
  1. **Code / Program correctness**

The system is logically organized into two independent MapReduce jobs:

* First job calculates document frequencies.
* Second job computes chi-square scores and selects top tokens.
* Preprocessing is cleanly separated into its own module.
* Each <key, value> transformation is appropriate for the data flow and Hadoop architecture.

Intermediate outputs and final outputs have been manually inspected for correctness during development with the devset.

* 1. **Performance considerations**

1. Single pass over the raw collection – all statistics needed for χ² are gathered during Phase 1; Phase 2 touches only its compact result.
2. In-mapper combining – local\_counts collapses duplicate keys before the shuffle, reducing emitted records by > 90 % in practice.
3. Top-k pruning early and often – χ² step keeps *k = 75* from the combiner onwards; worst-case key cardinality per category is bounded.
4. Compression – Snappy for map output and Gzip for final output strike a good balance between CPU cost and I/O volume.
5. Pure Python heap operations – heapq outperforms manual sorts inside every reduce call.
6. No secondary sort / custom partitioner needed – the category itself is the key, ensuring natural grouping.