The Group Project of COMP30770 Programming for Big Data

Project Title: Sentiment-Based Validation: Using Spark to Uncover if "Good Reads" Authors Earn Their Reputation Through Amazon Reviews.

[22385231 : Artjoms Kucajevs] / [22715709 : Shlok Patel]

Section 1. Introduction

Dataset Overview

Our analysis utilizes three CSV files from two distinct sources, comprising a comprehensive collection of book reviews and author information. The primary dataset consists of Amazon book reviews (3 million entries, 2.3GB) and book data (212,400 entries, 180MB), while the secondary dataset contains Goodreads author profiles (209,517 entries, 113MB).

Amazon Book Data Structure (headers):

https://www.kaggle.com/datasets/mohamedbakhet/amazon-books-reviews

- **Book Details**: Title, description, authors, image, previewLink, publisher, publishedDate, infoLink, categories, ratingsCount
- **Reviews**: Id, Title, Price, User_id, profileName, review/helpfulness, review/score, review/time, review/text

  Title acts as foreign key between datasets

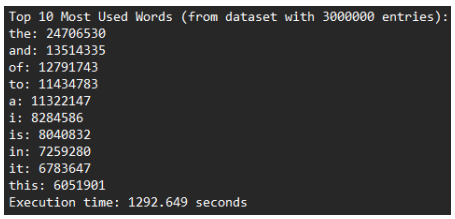
Goodreads Author Data Structure (headers):

https://www.kaggle.com/datasets/choobani/goodread-authors

- authorid, name, workcount, fan_count, gender, image_url, about, born, died, influence, average_rate, rating_count, review_count, website, twitter, genre, original_hometown, country, latitude, longitude

Volume Justification

The volume of our dataset presents meaningful computational challenges that validate our big data approach. A baseline word count operation using Scala without parallelization required over 20 minutes on a Ryzen 7 5000 CPU 8 cores that weren't taken advantage of in WordCount.scala. This substantial processing time demonstrates the necessity for optimized data handling techniques and provides ample opportunity to measure performance improvements through our implementation.



Variety Justification

The dataset exhibits significant variety across multiple dimensions:

Structural Variety: We integrate the found to be structured author data from Goodreads with the annoyingly unstructured amazon books reviews and those books data.

Relational Complexity: Each book in books details can have multiple associated reviews and authors, creating a complex network of relationships to analyse.

Cross-Dataset Integration: We identified over 37,000 authors appearing in both datasets, enabling good cross-referencing between review sentiment and good reads author profiles.

This variety is essential to our value proposition, as it allows us to perform sentiment analysis on book reviews and correlate findings with author information to determine whether Goodreads' "good authors" are supported by actual reader feedback.

Data Quality Considerations

While the Goodreads author data was relatively well-structured, the Amazon book data wasn't, so we attempted to clean out incorrect entries beforehand, though this is was difficult to do so without losing too much data. Due to this our implementation includes rigorous filtering logic throughout to address these challenges.

Section 2: Project Objective

Author sentiment analysis based on reviews and checking if the good reads authors are actually "good".

Using good/ bad keywords will classify books as good/ neutral/ bad based on their reviews (can be multiple reviews per book).

For each book there can be multiple authors, and an author can write multiple books. Authors will then accumulate sentiment word count from their books, getting a score.

We would then check these authors mapped with their classifications against the good reads authors to check how many of these authors are actually "good", *guilt by association if you may.*

Section 3. Traditional Solution

Initially we began testing the datasets, querying it through powershell which we weren't too familiar with, so decided to load the data into a mySQL db, the load process dumped a tonne of data in the process (this is where we realised the data was a lot less structured than expected). We tried cleaning the data an trying to put it into sql again but similar errors occurred.
Python was next, querying the data in python was nice as its basically pseudocode, but the overhead on pyspark didn't grant us results we liked. Due to overhead and the data size not being large enough the non parallelised code outperformed spark api. Bash was also used at some point but we saw no benefit.

We landed on scala, still fresh in our heads after last semester which supposedly worked very well with spark due to its jvm and functional nature. This proved to be correct and Scala is used to complete the project objective. As it is also easier to compare

First counted the words normally and then using spark rdds, to see first if this was achievable and to see if spark performed faster.

WordCount.scala has a row limit but for max rows it took 1292.649 seconds (21 mins) to execute

SparkWordCount.scala takes between 30-35 seconds.

For the actual objective I split the work load over 2 files. 1st to map authors to their sentiment rankings 2nd to match them to the good reads authors.

SeqAuthorSentimentClassifier.scala main steps:

Author Extraction

The code reads the books data CSV file to extract all unique authors and create a mapping between book titles and their authors

This is fairly quick as its linear and there no large volume of text processing

```
val allAuthorsSet = new HashSet[String]()
val titleToAuthorsMap = new HashMap[String, List[String]]()

var booksLine: String = null
var booksCounter = 0

while ({booksLine = booksReader.readLine(); booksLine != null && booksCounter < booksProcessLimit}) {
  val fields = parseCSVLine(booksLine)

  if (fields.length > Math.max(authorsIndex, titleIndex)) {
    val title = fields(titleIndex).trim
    val authorsField = fields(authorsIndex).trim

    if (!authorsField.isEmpty && authorsField != "[]") {
      val start = authorsField.indexOf('[')
      val end = authorsField.lastIndexOf(']')

      if (start >= 0 && end > start) {
        val bracketContent = authorsField.substring(start + 1, end).trim
        val authors = extractAuthorsFromBracketContent(bracketContent)

        // Add authors to the set of all authors
        authors.foreach(allAuthorsSet.add)

        // Add title-to-authors mapping if title is not empty
        if (!title.isEmpty) {
          titleToAuthorsMap.put(title, authors)
        }
      }
    }
  }

  booksCounter += 1
}
```

Review Sentiment Processing

This is the majour bottleneck in the pipeline although linear high constant factor due to 3 million review entries each with potentially large amounts of text.

For each review, multiple operations are performed:

String splitting to extract words

Word counting against sentiment dictionaries

HashMap lookups and updates

```
while ({reviewsLine = reviewsReader.readLine(); reviewsLine != null && reviewsCounter < reviewsProcessLimit}) {
  val fields = parseCSVLine(reviewsLine)
```

Inside this while loop is:

```
titleSentimentMap.put(posKey, titleSentimentMap.getOrElse(posKey, 0) + positiveCount)
titleSentimentMap.put(negKey, titleSentimentMap.getOrElse(negKey, 0) + negativeCount)
```

Which is constantly being updated

Score Calculation: After processing all reviews, for each book title:

It calculates a normalized sentiment score: (posCount - negCount) / totalWords

This score ranges from -1 (completely negative) to +1 (completely positive)

Moderately Fast as Operates on the aggregated data from previous steps

Results: 80.62 seconds run time

```
Author Sentiment Classification Results (Sequential):
----------------------------------------
Total unique authors: 152682
Authors with sentiment data: 133585
Authors classified as good (score > 0.0): 126224
Authors classified as bad (score < 0.0): 4005
Authors classified as neutral (score = 0.0 or no reviews): 22453
Execution time: 80.62 seconds
Books processed: 212000 (limit: 212000)
Reviews processed: 3000000 (limit: 3000000)
```

Value of results: only 4005/152682 authors classified as bad based on reviews, amazon reviewers seem to be quite nice. They would get a shock on reddit.

SeqGoodReadAuthorAnalyzer.scala main steps:

This code performs quickly sequentially as its essentially wordcount but for names of which there are not many

Load the names from txt file

Compare intersection

This step leverages Scala's Set data structure to perform efficient intersection operations. While the set intersections themselves are relatively fast (O(n)) it can get expensive for large quantities of data. It seemed to handle well for 126367 good reads authors and

```
// Find matches for each author category
println("Finding matches for each author category...")

// Find matches using Set's intersection
val matchedGoodAuthors = goodAuthors.toSet.intersect(goodreadsAuthors).toSeq.sorted
val matchedBadAuthors = badAuthors.toSet.intersect(goodreadsAuthors).toSeq.sorted
val matchedNeutralAuthors = neutralAuthors.toSet.intersect(goodreadsAuthors).toSeq.sorted
```

Results: 4.03 seconds

```
Input Statistics:
- Total Good Authors: 126370
- Total Bad Authors: 4010
- Total Neutral Authors: 22467
- Total Authors in Goodreads Dataset: 206360

Match Results:
- Matched Good Authors: 33673 (26.65% of good authors)
- Matched Bad Authors: 707 (17.63% of bad authors)
- Matched Neutral Authors: 2763 (12.30% of neutral authors)

Summary:
- Total Matched Authors: 37143
- Match Rate: 24.30% of all classified authors
- Execution Time: 4.03 seconds
```

Value of results: 37143 matches found between good reads authors dataset and books dataset, 707 of which are classified bad judging by comments sentiment. Don't really seem to be "good reads" authors afterall.

Section 4 MapReduce Optimisation:

The main bottleneck in the previous section was the wordcount with classifying words and accumulating them into authors in the previous section.
This was optimised using spark core api in scala, parallelising using rdds.

AuthorSentimentClassifier.scala:

Efficient Data Transformations Spark uses transformations like map, flatMap, filter, and reduceByKey that execute in parallel:

```
val allAuthors = booksDataLines
  .filter(_ != booksDataHeader)
  .flatMap(line => {
    val fields = parseCSVLine(line)

    if (fields.length > authorsIndex) {
```

Lazy Evaluation Spark uses lazy evaluation, meaning transformations are only executed when an action is called, allowing for optimization of the execution plan.

Efficient Joins and Aggregations The sequential version uses nested loops and HashMap lookups:

```
for ((title, authors) <- titleToAuthorsMap if titleScores.contains(title)) {
  val score = titleScores(title)
  for (author <- authors) {
    val current = authorScores.getOrElse(author, (0.0, 0))
    authorScores.put(author, (current._1 + score, current._2 + 1))
  }
}
```

Spark uses optimized distributed joins:

```
// Join titles with authors and calculate author sentiment
println("Calculating author sentiment scores...")
val authorSentiments = titleToAuthors                                    : RDD[(String, List[String])]
  .filter(_._2.nonEmpty)                                                 : RDD[(String, List[String])]
  .flatMap { case (title, authors) =>
    // Create (title, author) pairs for each author of the book
    authors.map(author => (title, author))
  }                                                                      : RDD[(String, String)]
  .join(titleScores)                                                     : RDD[(String, (String, Double))]
  .map { case (title, (author, score)) =>
    // Output: (author, (score, count))
    (author, (score, 1))
  }                                                                      : RDD[(String, (Double, Int))]
```

In-Memory Caching Spark also caches intermediate RDDs to avoid unnecessary recompution

```
// Aggregate sentiment counts by title and calculate score
val titleScores = titleSentiment                                                                    : RDD[((String, String), Int)]
  .reduceByKey(_ + _)                                                                               : RDD[((String, String), Int)]
  .map { case ((title, sentimentType), count) => (title, (sentimentType, count)) }                  : RDD[(String, (String, Int))]
  .groupByKey()                                                                                     : RDD[(String, Iterable[(String, Int)])]
  .mapValues { sentiments =>
    val sentimentMap = sentiments.toMap
    val posCount = sentimentMap.getOrElse("positive", 0)
    val negCount = sentimentMap.getOrElse("negative", 0)
    val totalWords = posCount + negCount

    if (totalWords > 0) {
      // Calculate normalized sentiment score: (positive - negative) / total
      val sentimentScore = (posCount - negCount).toDouble / totalWords.toDouble
      sentimentScore
    } else {
      0.0
    }
  }                                                                                                 : RDD[(String, Double)]
  .cache() // Cache for joining
```

GoodReadAuthorAnalyzer:

The file contains several explicit rdd operations:

Each of these operations is executed in parallel across all available cores

```
val goodAuthors = sc.textFile(goodAuthorsPath).map(_.trim).filter(_.nonEmpty).cache()
val badAuthors = sc.textFile(badAuthorsPath).map(_.trim).filter(_.nonEmpty).cache()
val neutralAuthors = sc.textFile(neutralAuthorsPath).map(_.trim).filter(_.nonEmpty).cache()
```

Below the distinct() operation is particularly important as it's a shuffle operation that requires redistributing data across partitions, demonstrating Spark's ability to handle complex distributed operations.

```
val goodreadsAuthors = goodreadsAuthorsRDD
  .filter(_ != goodreadsHeader)
  .map(line => {
    val fields = parseCSVLine(line)
    if (fields.length > nameIndex) {
      fields(nameIndex).trim.replaceAll("^\"|\"$", "")
    } else {
      ""
    }
  })
  .filter(_.nonEmpty)
  .distinct()
  .cache()
```

Below the distributed intersection() operation is much more efficient than implementing a custom solution.

```
// Find matches using Spark's intersection
val matchedGoodAuthors = goodAuthors.intersection(goodreadsAuthors).cache()
val matchedBadAuthors = badAuthors.intersection(goodreadsAuthors).cache()
val matchedNeutralAuthors = neutralAuthors.intersection(goodreadsAuthors).cache()
```

Performance Expectations

Linear Scaling: With sufficient data and proper parallelization, execution time could decrease nearly linearly with the number of cores. The machine all the testing was done on has an 8 core cpu which mean a potential 8 times reduction in time.

For example join operations between titleToAuthors and titleScores would be distributed across 8 cores, potentially reducing this bottleneck significantly.

This sounds lovely but realistically .... See results

Results:

Parallelised AuthorSentimentClassifier:

```
spark Author Sentiment Classification Results:
----------------------------------------
Total unique authors: 152847
Authors with sentiment data: 152847
Authors classified as good (score > 0.0): 126370
Authors classified as bad (score < 0.0): 4010
Authors classified as neutral (score = 0.0 or no reviews): 22467

Execution time: 25.57 seconds
```

3.2 fold decrease from the sequential version. Although not the 8 fold we had hoped for, this is still very good as this would scale linearly for even larger sets.

Parallelised GoodReadAuthorAnalyzer:

```
Input Statistics:
- Total Good Authors: 126367
- Total Bad Authors: 4010
- Total Neutral Authors: 22467
- Total Authors in Goodreads Dataset: 206360

Match Results:
- Matched Good Authors: 33673 (26.65% of good authors)
- Matched Bad Authors: 707 (17.63% of bad authors)
- Matched Neutral Authors: 2763 (12.30% of neutral authors)

Summary:
- Total Matched Authors: 37143
- Match Rate: 24.30% of all classified authors
- Execution Time: 2.39 seconds
```

This is also an improvement although the sequential was already fast enough at 4 seconds, this is still an improvement.

Results match deviation:

I/O bottlenecks: The program is I/O bound, adding more cores may not provide linear scaling

Data skew: If certain partitions contain significantly more data, some cores may be idle while others are overloaded

Overhead: More cores means more coordination overhead

Due to this the improvements are not as great as we had fathomed.

Overall Results value:

We successfully completed the value of the project. With finding 707 authors in the authors good reads dataset who we classified as bad from a book data review dataset. We were hoping for more but since only 37143 matches between the 2 dataset sources this is still an excellent result, which tells us the good reads authors are in fact majority good, and Amazon reviewers are also quite nice.

EXTRA

Initial Word Count calculations for reviews text:

To test spark with scala to see if there was an improvement there are 2 files SparkWordCount.scala and WordCount.scala in the git hub.

This is quite interesting as the difference in results is drastic, really showing spark improvements.

Sequentiall word count:

```
Top 10 Most Used Words (from dataset with 3000000 entries):
the: 24706530
and: 13514335
of: 12791743
to: 11434783
a: 11322147
i: 8284586
is: 8040832
in: 7259280
it: 6783647
this: 6051901
Execution time: 1292.649 seconds
```

Parallelised using spark core api:

```
Top 10 words in reviews: all csv
the: 24706530
and: 13514335
of: 12791743
to: 11434783
a: 11322147
i: 8284586
is: 8040832
in: 7259280
it: 6783647
this: 6051901
Execution time: 31.00 seconds
```

41 fold decrease!!

There are also some files that I used to compare and debug results throughout that I left in the github namely booksDataExtractSortAuthors.scala, authorsDataExtractSortAuthors.scala and CompareAuthorLists.scala, which were used to compare results of matched authors that are in both good reads and amazon books dataset.