Word Count and Data Analysis with PySpark

Samyak Shah
ID: 50604267

Contents

1 Introduction

In this assignment, we perform word count and data analysis using **PySpark** on the book *"History of the World War, Volume 1 (of 7): An authentic narrative of the..."* sourced from Project Gutenberg. The analysis includes:

1. Basic Word Count Implementation
2. Extended Word Count with Additional Preprocessing
3. Data Flow Analysis
4. Word Co-Occurrence Analysis
5. Data Bias Identification and Review of a Streaming Data Paper

1.1 Why *History of the World War, Volume 1*?

- **Comprehensive Content:** It is very extensive and provides details with ample of data for a proper and meaningful analysis.
- **Public Domain:** It is easy to access by others and analysers, and there should perhaps be no issues with compliance and copyrights.
- **Structured Narrative:** Facilitates effective text processing and analysis.

2 Part 1: Word Count and Data Analysis with PySpark [50 Points]

2.1 1. Basic Word Count Implementation [20 Points]

2.1.1 1.1. Setting Up the Environment

Pyspark Installation.

```
!pip install pyspark
```

2.1.2 1.2. PySpark Code for Basic Word Count

```
from pyspark import SparkConf, SparkContext
import string
if SparkContext._active_spark_context:
    SparkContext._active_spark_context.stop()



conf = SparkConf().setAppName("ExtendedWordCount").setMaster("local")
sc = SparkContext(conf=conf)
textFile = sc.textFile("a.txt")
wordCounts = textFile.flatMap(lambda line: line.split()) \
                     .map(lambda word: (word, 1)) \
                     .reduceByKey(lambda a, b: a + b)

sortedWordCounts = wordCounts.sortBy(lambda x: x[1], ascending=False)

results = sortedWordCounts.collect()

for word, count in results:
    print(f"{word}: {count}")
```

2.1.3 1.3. Explanation of the Code

- **Reading the File:** The text file is read into an RDD named ‘textFile‘.
- **Word Count Logic:**
  - `flatMap`: Splits each line into words.
  - `map`: Transforms each word into a ’(word, 1)’ pair.
  - `reduceByKey`: Sums the counts for each word.
- **Sorting:** The words are sorted in descending order based on their counts.

2.1.4 1.5. Comments on the Results

- **High-Frequency Words:** Common words such as ”the”, ”and”, ”to” appear frequently, which are most occurring in English literature.
- **Vocabulary Insights:** The analysis provides insights into the author’s writing style and his thematic usage of vocabulary.

2.2 2. Extended Word Count [20 Points]

2.2.1 2.1. Enhancements Implemented

To gain more meaningful insights, the basic word count is enhanced with the following features:

- **Case Insensitivity:** Convert all words to lowercase.
- **Punctuation Removal:** Remove punctuation from words for uniformity.
- **Stop Words Removal:** Exclude common stop words (e.g., "the", "and") to focus on meaningful words.

2.2.2 2.2. PySpark Code for Extended Word Count

Below is the PySpark code that implements the extended word count with additional preprocessing.

```
wordCounts = textFile.flatMap(lambda line: line.split()) \
                     .map(lambda word: (word, 1)) \
                     .reduceByKey(lambda a, b: a + b)

sortedWordCounts = wordCounts.sortBy(lambda x: x[1], ascending=False)

results = sortedWordCounts.collect()

for word, count in results:
    print(f"{word}: {count}")
import string

stopWords = set([
    "the", "and", "to", "of", "a", "in", "that", "is", "it", "was",
    "he", "for", "with", "as", "his", "on", "be", "at", "by", "i",
    "this", "not", "but", "are", "from", "or", "have", "an", "they",
    "you", "her", "she", "him", "their", "my", "we", "me", "so",
    "if", "them", "will", "would", "can", "all", "there", "what",
    "about", "up", "out", "who", "get", "which", "when", "make",
    "just", "like", "now", "than", "then", "some", "into", "could",
    "these", "no", "only", "its", "over", "also", "other", "how",
    "your", "because", "been", "do", "did", "didn't", "does", "didn't",
   "were", "had"
])
def preprocess(line):
    translator = str.maketrans('', '', string.punctuation)
    return line.translate(translator).lower()

processedText = textFile.map(preprocess)

extendedWordCounts = processedText.flatMap(lambda line: line.split()) \
                                  .filter(lambda word: word not in
   stopWords) \
                                  .map(lambda word: (word, 1)) \
                                  .reduceByKey(lambda a, b: a + b)

sortedExtendedWordCounts = extendedWordCounts.sortBy(lambda x: x[1],
   ascending=False)

```

```
top15Words = sortedExtendedWordCounts.take(15)

print("\nTop 15 Most Common Words (After Cleaning):")
for word, count in top15Words:
    print(f"{word}: {count}")
!pip install pyngrok
from pyngrok import ngrok

ngrok.set_auth_token("2om1zYwDuD9fXJywSj2Jr2lCG3z_36d4WoZxEHzVQbveZye44")

public_url = ngrok.connect(4040)
print(f"Spark UI is accessible at: {public_url}")
```

2.2.3 2.3. Comments on the Extended Results

- **Focused Vocabulary:** Removing stop words and punctuation results in a more focused set of words that are actually contributing.
- **Enhanced Insights:** The extended word count provides deeper insights.

2.2.4 2.4. Output

```
Top 15 Words by Frequency:
german: 219
war: 196
against: 166
army: 160
germany: 158
french: 101
one: 100
general: 99
great: 97
first: 97
austria: 93
made: 90
project: 85
russia: 81
any: 79
```

2.3 3. Data Flow Analysis [10 Points]

2.3.1 DAG Visualization

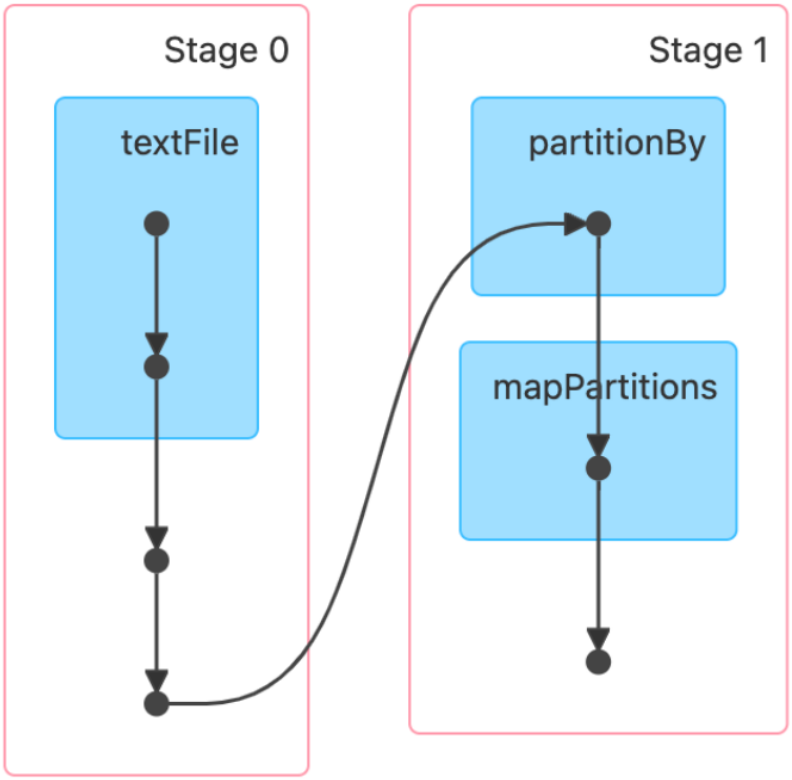

Figure 1: Directed Acyclic Graph

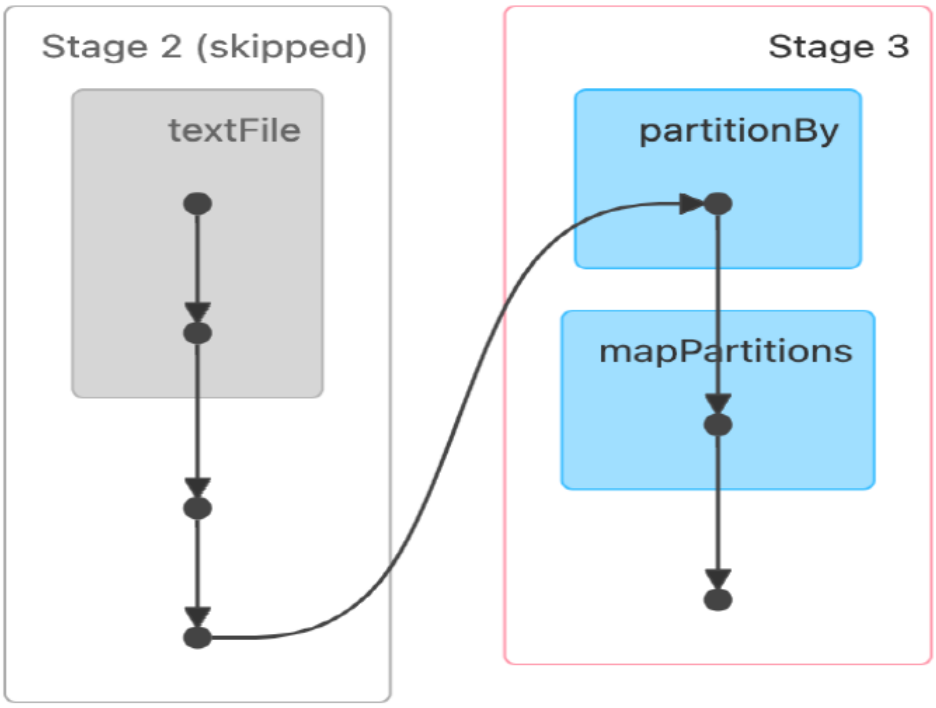

Figure 2: Directed Acyclic Graph

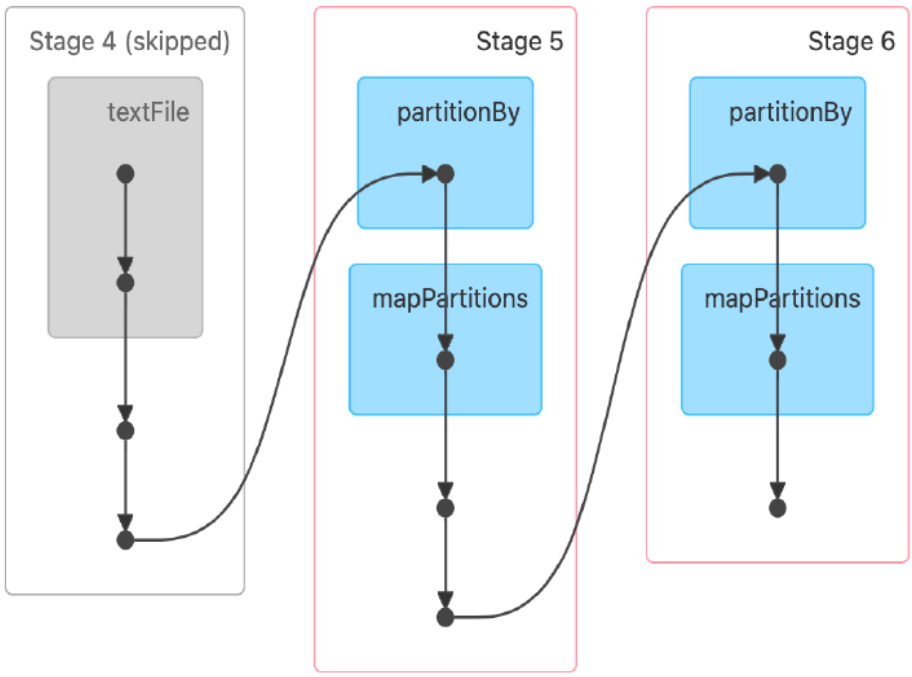

Figure 3: Directed Acyclic Graph

2.3.2 3.3. Lineage Graph (Simple DAG)

Below is a simplified lineage graph representing the RDD transformations:

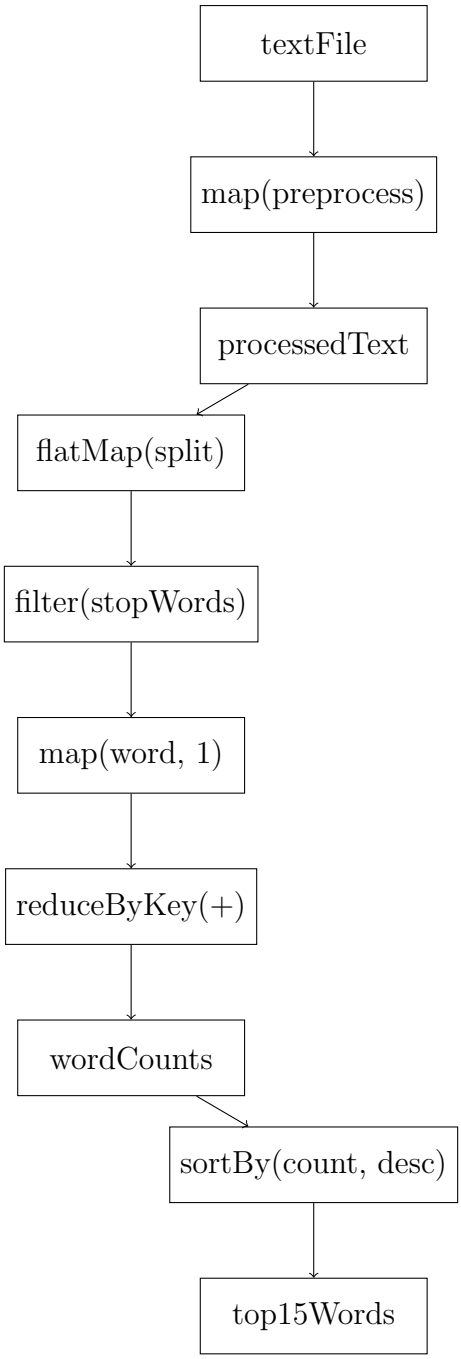

Figure 4: Lineage Graph of RDD Transformations

2.3.3 3.4. Explanation of the Lineage Graph

- **textFile:** Initial RDD created by reading the text file.
- **map(preprocess):** Preprocess each line by removing punctuation and converting to lowercase.
- **flatMap(split):** Split lines into individual words.
- **filter(stopWords):** Remove stop words from the RDD.
- **map(word, 1):** Map each word to a '(word, 1)' pair.

- **reduceByKey(+):** Aggregate counts for each word.
- **sortBy(count, desc):** Sort words by their counts in descending order.
- **top15Words:** Retrieve the top 15 most common words.

3 Part 2: Word Co-Occurrence with PySpark [30 Points]

3.1 1. Word Co-Occurrence Implementation [20 Points]

3.1.1 1.1. PySpark Code for Word Co-Occurrence

Below is the PySpark code that performs word co-occurrence analysis by counting bigrams.

```
from pyspark import SparkConf, SparkContext


if SparkContext._active_spark_context:
    SparkContext._active_spark_context.stop()


conf = SparkConf().setAppName("WordCountAnalysis").setMaster("local[*]")
sc = SparkContext(conf=conf)


textFilePath = "a.txt"

import string

stopWords = set([
    "the", "and", "to", "of", "a", "in", "that", "is", "it", "was",
    "he", "for", "with", "as", "his", "on", "be", "at", "by", "i",
    "this", "not", "but", "are", "from", "or", "have", "an", "they",
    "you", "her", "she", "him", "their", "my", "we", "me", "so",
    "if", "them", "will", "would", "can", "all", "there", "what",
    "about", "up", "out", "who", "get", "which", "when", "make",
    "just", "like", "now", "than", "then", "some", "into", "could",
    "these", "no", "only", "its", "over", "also", "other", "how",
    "your", "because", "been", "do", "did", "didn't", "does", "didn't",
   "were", "had"
])


def preprocess(line):
    translator = str.maketrans('', '', string.punctuation)
    return line.translate(translator).lower()

processedText = textFile.map(preprocess)

def generateBigrams(words):
    return zip(words, words[1:])

bigrams = processedText.flatMap(lambda line: generateBigrams(line.split
   ()))
filteredBigrams = bigrams.filter(lambda bigram: bigram[0] not in
   stopWords and bigram[1] not in stopWords)
bigramPairs = filteredBigrams.map(lambda bigram: (bigram, 1))
bigramCounts = bigramPairs.reduceByKey(lambda a, b: a + b)
sortedBigramCounts = bigramCounts.sortBy(lambda x: x[1], ascending=False
   )
top10Bigrams = sortedBigramCounts.take(10)
```

```
44
45 print("\nTop 10 Most Frequent Bigrams:")
46 for bigram, count in top10Bigrams:
47     print(f"{bigram[0]} {bigram[1]}: {count}")
48
49 from pyngrok import ngrok
50
51 ngrok.set_auth_token("2om1zYwDuD9fXJywSj2Jr2lCG3z_36d4WoZxEHzVQbveZye44"
       )
52
53 public_url = ngrok.connect(4040)
54 print(f"Spark UI is accessible at: {public_url}")
```

3.1.2 1.2. Explanation of the Code

- **Generating Bigrams:**
  - `generateBigrams`: This function takes a words and returns a list of bigrams using the zip function.
  - `bigrams`: Generates bigrams from each line of the processed text.
- **Filtering Bigrams:**
  - `filteredBigrams`: Filters out bigrams where stop words are there.
- **Counting Bigrams:**
  - `bigramPairs`: Maps each bigram to a `((word1, word2), 1)` pair.
  - `bigramCounts`: Aggregates the counts for each bigram using `reduceByKey`.
- **Sorting and Collecting Results:**
  - `sortedBigramCounts`: Sorts the bigrams in descending order based on their counts.
  - `top10Bigrams`: Collects the top 10 most frequent bigrams.
- **Output:** Prints the top 10 most common bigrams along with their counts.

3.1.3 1.3. Sample Output

```
Top 10 Most Frequent Bigrams:
project gutenberg™: 48
against germany: 41
united states: 39
project gutenberg: 21
army corps: 20
german army: 18
world war: 17
von hindenburg: 16
against austria: 16
electronic works: 16
```

3.2 2. Analysis of Co-Occurrence [10 Points]

3.2.1 2.1. The Role of Bigrams in Analyzing Word Relationships

Bigrams are pairs of words in a text. Analyzing bigrams provides deeper insights into the relationships and structures within the text beyond individual word frequencies.

Reason for analysing Bigrams

a. **Contextual Understanding:** Bigrams capture the context in which words appear.

b. **Phrase Identification:** They help in identifying common phrases and expressions, which are crucial for tasks like language modeling and speech recognition.

c. **Improved Accuracy:** Incorporating bigrams can enhance the accuracy of natural language processing applications by providing contextual clues that single words alone might not offer.

d. **Semantic Relationships:** Bigrams can reveal semantic relationships between words, aiding in sentiment analysis, topic modeling, and more.

e. **Enhancing Machine Learning Models:** Features based on bigrams can improve the performance of machine learning models in various NLP tasks.

4 Part 3: Data Bias and Review [20 Points]

4.1 1. Bias Identification [10 Points]

4.1.1 Scenario 1: A fitness app tracks user activity but excludes data from users without internet access.

Type of Bias: Selection Bias

Explanation: Selection bias occurs when the sample obtained is not representative of the population to be analyzed. In this, excluding offline unavailable users will finally give a dataset that does not represent all user populations in the whole sense. This would lead to incomplete insight because the behaviors and needs of offline users would be left out.

Potential Solution: To mitigate selection bias, the fitness app should aim to include data from all users, online and offline. It should be designed to have data synchronization when users are offline, automatically updating once internet connectivity is re-established. It should also apply different means to collect data in order not to depend entirely on access to the internet.

4.1.2 Scenario 2: A recruitment algorithm favors candidates from top universities without considering applicants' experiences or skills.

Type of Bias: Algorithmic Bias (Favoritism Bias)

Explanation: Algorithmic bias exists when an algorithm repeatedly produces prejudicial results because of some flawed assumption in the process of machine learning. Favouring candidates from top universities, apart from all actual experiences or skills of candidates, gives undue advantage to certain groups and might result in missing highly qualified candidates from less prestigious institutions.

Potential Solution: The recruitment algorithm should be redesigned to base its assessment on a holistic set of criteria, including work experience, skills, certifications, and performance in practical assessments. The bias in them should be regularly checked for fairness regularly and their parameters adjusted so that the assessment score maintains fairness and objectivity when considering candidates.

5 Review of Streaming Data Paper [10 Points]

5.1 Paper Reviewed: "Discretized Streams: Fault-Tolerant Streaming Computation at Scale"

Review: Using discretized streams, "Discretized Streams: Fault-Tolerant Streaming Computation at Scale" offers a framework for handling streaming data. The authors present a method that allows for scalable, fault-tolerant processing of continuous streams by effectively dividing them into manageable micro-batches. By incorporating and utilizing the advantages of both paradigms, this closes the gap between real-time stream processing and batch processing.

Because the streams are discretized, the system have to manage data so that node failures or network partitions never interfere with continued processing. The framework has high scalability, which is demonstrated by extensive tests processing massive data streams with very high data and low delay and latency.

The paper also shows implementation details. It show the use of distributed computing techniques to optimize performance. Additionally, integration with already-existing big data platforms, such Apache Spark, demonstrates its usefulness. The authors enhance this toolbox of real-time data analytics by adding value by making Spark Streaming more scalable and fault-tolerant, building on its existing capabilities.

This idea and proposed solution tackles two significant problems: flaw tolerance and scalability. The article is important for the development of compute streaming. The discretized stream model provides workable solutions that can be mixed into the majority of real-world applications, such as IoT data processing, social media monitoring, and financial analytics. This paper is very comprehensive, and shows us the analysis of streaming data processing. This paper is very important for advancement in data streaming.

6 References

1. Biewald, L. (2014). *Spark: Lightning-Fast Big Data Analysis*.

2. Stonebraker, M., Çetintemel, U., & Abdul-Kader, S. (2013). Discretized Streams: Fault-Tolerant Streaming Computation at Scale. *Proceedings of the 2nd ACM SIGMOD Workshop on Cloud Computing for Data Management*.

3. Project Gutenberg. *History of the World War, Volume 1*. Retrieved from `https://www.gutenberg.org/cache/epub/74726/pg74726.txt`

4. **PySpark Documentation**. Retrieved from `https://spark.apache.org/docs/latest/api/python/`