Final Report (Group 15)

Joachim Boltz 9940177, Naida Dzigal 1228695, Taranpreet Kaur Bambrah 11717209

May 2023

Contents

1 Introduction

In this assignment, we applied Spark to process large text corpora.

In Assignment 1, we utilized the Amazon Review Dataset and implemented partial and similar functionality in parts 1 and 2 of this assignment. However, this time, we leveraged Spark to accomplish the same tasks. In Part 3, we employed the processing pipeline we developed as input for a text classification task. To minimize cluster usage, we will utilize the development set when generating the requested files for submission. Additionally, we will compare the results with those obtained in Assignment 1, using only the development set.

2 Problem Overview

The problem statement involves working with the Amazon Review Dataset and implementing various tasks using Spark. The tasks are divided into three parts.

Part 1 focuses on working with Resilient Distributed Datasets (RDDs). The goal is to calculate chi-square values and output the sorted top terms per category. The results should be written to a file called output_rdd.txt. A comparison is required between the generated output_rdd.txt and the output.txt generated in Assignment 1. Observations from the comparison should be included in the submission report.

In Part 2, the objective is to convert the review texts into a vector space representation using TFIDF-weighted features. This is done using the Spark DataFrame/Dataset API and building a transformation pipeline. Built-in functions for tokenization, casefolding, stopword removal, TF-IDF calculation, and chi-square selection should be used. The selected terms should be written to a file called output_ds.txt, and a comparison should be made with the terms selected in Assignment 1. The findings should be briefly described in the submission report.

Part 3 focuses on text classification. The extracted features from Part 2 are used to train a text classifier, specifically a Support Vector Machine (SVM) classifier. The pipeline from Part 2 is extended to include the SVM classifier, and vector length normalization is applied. The goal is to predict the product category from a review's text. Best practices for machine learning experiment design should be followed, including splitting the data into training, validation, and test sets, making experiments reproducible, and using grid search for parameter optimization. The performance of the trained classifiers should be evaluated using the MulticlassClassificationEvaluator with F1 measure as the criterion. The results and interpretations should be included in the submission report.

Efficiency and documentation are emphasized throughout the assignment. The preferred format for submission is a Jupyter notebook, but Spark jobs are also acceptable with careful documentation of code, outputs, and graphs. The submission report should include sections covering Introduction, Problem Overview, Methodology and Approach (with a figure illustrating the strategy and pipeline), Results, and Conclusions. The report should not exceed 8 pages.

3 Methodology and Approach

3.1 Part 1 RDDs

In this task, we were supposed to repeat the steps of assignment 1, which involve calculating chi-square values, generating sorted top terms per category, and creating a combined dictionary, using RDDs and transformation operations. The resulting output was to be saved in a file called ”output_rdd.txt”. Additionally, we compared the contents of ”output_rdd.txt” with the previously generated ”output.txt” from Assignment 1.

The initial step involves loading the required libraries and the dataset. The text in each review is tokenized to extract individual terms. The terms are split based on whitespace, digits, and specific delimiter characters. Additionally, stopwords and single-character terms are removed. The resulting terms are then mapped with their corresponding categories.

The calculations for chi-square values are performed based on the frequencies of terms in different categories. The joint attributes are joined, and the necessary calculations are made to determine A, B, C, and D values as required in the exercise description. The chi-square values are then computed using the formula given in the lectures. The results are sorted in descending order and grouped by category. The top 75 terms with the highest chi-square values are selected for each category.

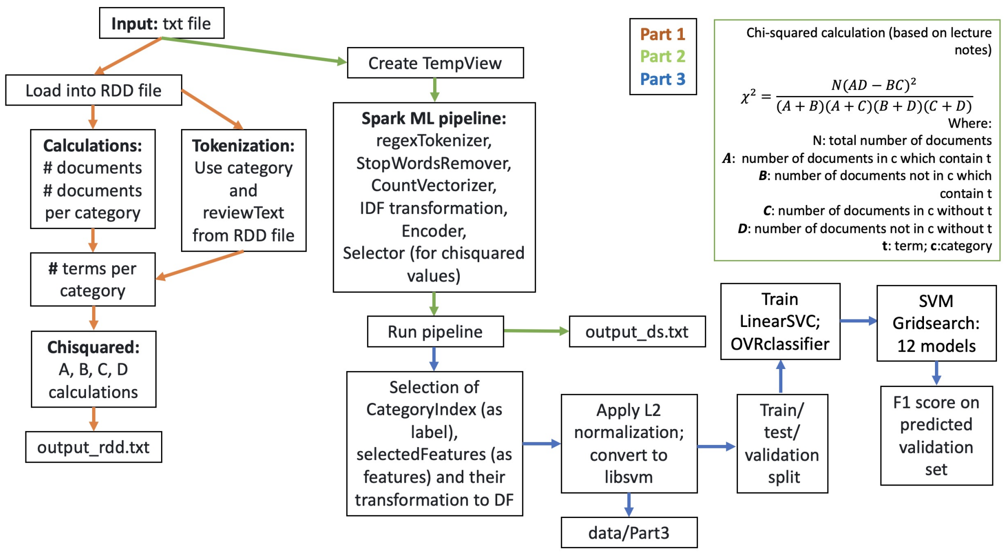
To generate the output, the terms with their corresponding chi-squared values were written to the ”output_rdd.txt” file. The format of the output is adjusted to match the required format. Additionally, a dictionary was created from the last line of the output, which contains the sorted top terms from all categories.

3.2 Part 2 Datasets/DataFrames: Spark ML and Pipelines

Here, we utilized the Spark DataFrame/Dataset API to convert the review texts into a classic vector space representation with TFIDF-weighted features. Our objective is to build a transformation pipeline that will be used in Part 3. The terms selected in this manner will be written to a file named ”output_ds.txt”.

The first step was to read the reviews JSON file from the HDFS directory and create a DataFrame with the columns ”category” and ”reviewText”. Here we use the Spark DataFrame API to select the necessary columns for further processing. The next part of the pipeline involved various preprocessing steps, including case folding, tokenization, stopword removal, and vocabulary limiting / cleaning up. After the preprocessing steps, the TF-IDF calculation was performed to obtain the TF-IDF-weighted features. To prepare for the text classification task, the categories needed to be encoded as labels. Then, in the next step, chi-squared selection was used to select the top 2000 terms based on their importance for the classification task. The next step involved stacking all the preprocessing and transformation stages into a pipeline. The final step was to fit the pipeline to the reviewTextDF DataFrame and transform it to obtain the final DataFrame with the selected features.

After applying the pipeline and obtaining the final DataFrame, the selected features (terms) were extracted, sorted and saved to the file ”output_ds.txt”. These terms were the result of chi-squared feature selection.



Input: txt file
Load into RDD file
Calculations: # documents # documents per category
Tokenization: Use category and reviewText from RDD file
# terms per category
Chisquared: A, B, C, D calculations
output_rdd.txt
Create TempView
Spark ML pipeline: regexTokenizer, StopWordsRemover, CountVectorizer, IDF transformation, Encoder, Selector (for chisquared values)
Run pipeline
output_ds.txt
Selection of CategoryIndex (as label), selectedFeatures (as features) and their transformation to DF
Apply L2 normalization; convert to libsvm
data/Part3
Train/ test/ validation split
Train LinearSVC; OVRclassifier
SVM Gridsearch: 12 models
F1 score on predicted validation set
Part 1
Part 2
Part 3
Chi-squared calculation (based on lecture notes)
$\chi^2 = \frac{N(AD-BC)^2}{(A+B)(A+C)(B+D)(C+D)}$
Where:
N: total number of documents
A: number of documents in c which contain t
B: number of documents not in c which contain t
C: number of documents in c without t
D: number of documents not in c without t
t: term; c:category

Figure 1: Pipeline and strategy of our approach.

3.3 Part 3 Text Classification

Our code involves the training and evaluation of a multiclass classification model using Support Vector Machines (SVM) with the One-vs-Rest (OVR) strategy in Apache Spark. We normalized the input data from part 2 using an L2 input norm (which is just a vector length normalization) by using the 'Normalizer' class in Spark ML. We then store these features in the 'normFeatures' column of the dataframe.

The model training and evaluation is done by splitting the data into training, test and validation sets. We instantiate a linear support vector machine classifier and then apply the One-vs-Rest strategy to enable multi-class classification. We fit the OVR classifier on the training data using the LinearSVC classifier and then make predictions on the validation set. The predictions are evaluated using the accuracy metric from the MulticlassClassificationEvaluator. The test error is calculated as 1 minus the accuracy metric value.

Then, for the text classification we built three different models. First we made a test SVM model in which we manually created a SVM classifier with `LinearSVC(maxIter=10, regParam=0.1)`, then we made a parameter grid search on the dataset with 2000 features, then we made the same parameter gridsearch with 200 / 50 reduced features. For the test classifier we used a simple 80:20 train test split, for the two grid searches we again used 20% as test and split the remaining 80% into 80:20 train and validate.

We configured the grid search in the following way:

```
paramGrid = ParamGridBuilder()\
    .addGrid(lsvc.regParam, [0.2, 0.1, 0.01]) \
    .addGrid(lsvc.standardization, [False, True])\
    .addGrid(lsvc.maxIter, [5, 10])\
    .build()
```

For the validation data we used an F1 metric with a multiclass classifier:

```
MulticlassClassificationEvaluator(metricName="f1")
```

4 Results

Below we review the results from our text classification pipeline in part 3 and we also compare our outputs from part 1 and 2 to that of exercise 1.

4.1 Part 3

The grid search chose the following parameters:

- MaxIter: 10
- Standardization: True
- RegParam: 0.01

For the SVM classification we got the following results on our F1 score:

- Test classifier: 0.31 error rate
- Grid search 2000 features: 0.31 error rate
- Grid search 200 features: 0.48 error rate
- Grid search 50 features: 0.64 error rate

4.2 Part 1 and 2: comparison to Exercise 1

Specifically when it comes to a comparison with Exercise 1, we compared two outputs: our resulting output txt files and the runtime of our code.

When it comes to the output text files, the top outputs are almost identical, but the less frequent terms towards the bottom of the text file start to deviate. This is however to be expected, as MapReduce and Spark differ in several aspects:

- Data Partitioning: MapReduce and Spark use different strategies for data partitioning. MapReduce divides the input data into fixed-sized chunks and processes them independently. On the other hand, Spark partitions the data based on the RDD (Resilient Distributed Dataset) abstraction, which allows for more flexible data partitioning. Different partitioning strategies can lead to variations in how the data is distributed and processed, potentially affecting the classification results.
- Processing Model: MapReduce and Spark have different processing models. MapReduce follows a batch-oriented processing model, where data is processed in discrete stages (map and reduce). Spark, on the other hand, employs a more iterative and interactive processing model. This difference can impact how the classification algorithm operates and may lead to variations in the output.
- Algorithmic Implementations: MapReduce and Spark may use different implementations of the classification algorithm. While the underlying algorithm may be the same, the specific implementation details can differ. These differences can arise from variations in libraries, optimizations, or even the programming languages used. The variations in implementation can result in slight differences in the classification output.

- Optimization Techniques: Spark provides several optimization techniques, such as in-memory caching and lazy evaluation, which can enhance performance. These optimizations can lead to improved processing speed but may introduce subtle differences in the output compared to MapReduce, which lacks some of these optimization capabilities.
- Resource Management: MapReduce and Spark have different resource management systems. MapReduce typically relies on a dedicated resource manager like YARN or Hadoop, while Spark has its own built-in resource management system. Differences in resource allocation, availability, and scheduling can influence the overall performance and potentially impact the classification output.

When it comes to the runtime of our code, we note that Spark was significantly faster (about 9-10x faster than our MapReduce code from Exercise 1. This can also be due to the fact that we spent significantly longer on optimising our code in Exercise 2 and that spark in general has many in-built features for optimisation compared to MapReduce. Our full notebook takes about 20 minutes to run with the first part being complete in a few minutes. In Exercise 1 the first part alone took about 30 minutes.

Finally, it is important to note however that, even though our outputs from exercise 1 and exercise 2 differ somewhat, they are still consistent.

5 Conclusions

Overall, our group was able to build a pipeline for calculating chi-squared values of Amazon product reviews using pyspark functionalities. We were able to convert the reviews texts to a classic vector space representation using TFIDF-weighted features using a transformation pipeline. Finally, we trained a text classifier on the features extracted from part 2 and split the data into a training, test and validation set. We were able to feed our data to a Spark ML pipeline that uses Support Vector Machines and used a grid search for parameter optimization.

In part 1 we achieve a similar and consistent (but not identical) result as in Exercise 1. Our results from part 3 show that with a grid search of 50 features we achieve an error rate of 0.64 but we improve that to 0.31 with a 2000 parameter search.