TECHNISCHE UNIVERSITÄT WIEN
Vienna | Austria

Vienna University of Technology

Data Intensive Computing 2024S

Exercise 2 - Text Processing and Classification using Spark

Dzhamilia Kulikieva - 12340251
(e12340251@student.tuwien.ac.at)

Luiza Corpaci - 12037284
(e12037284@student.tuwien.ac.at)

Claudia Kößldorfer - 11825357
(e11825357@student.tuwien.ac.at)

May 29, 2024

1 Introduction

In this project we applied methodologies of Apache Spark that demonstrate practical approaches to handling and analyzing large text datasets, from preprocessing to feature extraction and model training.

The primary focus of Part 1 is to preprocess the raw review data using RDD transformations and actions in Apache Spark. Part 2 involves converting review texts to TF-IDF-weighted features using the Spark DataFrame/Dataset API and building a transformation pipeline.Part 3 focuses on training a text classifier using the features extracted in Part 2. At the end a short summary is given with main results for each part of the analysis.

2 Part 1 RDDs

2.1 Problem Overview

The Task here was to perform the same steps as in the previous assignment, but utilizing Spark and specifically Resilient Distributed Datasets (RDDs). We should perform text preprocessing on the review dataset and calculate chi-square values for the terms in the review-text, to identify terms which can be used to discriminate between product categories.

2.2 Methodology and Approach

To start of a SparkContext (***sc***) needs to be created to be able to use RDDs. After this is done we define functions to perform parts of the process and calculations.
To read our review data we used ***sc.textFile*** which creates an RDD of string, each string being a line of the input file. The stopwords text file is not read into an RDD, but rather turned into a broadcast variable. These are used to share smaller read-only data across nodes, which is exactly what we need the stopwords for.
The stopwords are used in the transformation of our reviews RDD, specifically in the call of our ***preprocess*** function. To apply the preprocessing on every line of our data we use the function ***map***, which is a transformation. In preprocess the data is redused to only ***category*** and ***reviewText*** and casefolding, tokenization and stopword filtering is applied. The result of a ***map*** call on the RDD is another RDD, in this case with the category of the line and the set of used words in the review text. The result of this is also cached in memory to allow subsequent actions to reuse the data. At the same time the unprocessed RDD is removed from memory and disk storage using ***unpersist***.
Up until now due to how RDDs work the preprocessing hasn't actually been executed, as the evaluation of transformations is lazy, but in order to get the total line count of our document (needed for the chi-square calculation) we need to have everything up to that point executed. This is automatically caused by using an action in this case ***count*** on the RDD.

In the next steps the other values needed for chi-square are calculated using transformations.

```
category_line_count = reviews_rdd.map(lambda x: (x[0], 1))
    .reduceByKey(lambda a, b: a + b).collectAsMap()
```

The line count for each category is calculated by applying ***map*** which transforms each line into a tuple of the category and the value 1. Then ***reduceByKey*** combines the results from

this by the key (category) and sums up the values. As we want to use these results in a further transformation, the chi-square calculation, we want to broadcast it, but it isn't possible to do this with out using ***collectAsMap*** on the result first. Spark doesn't allow RDDs to be broadcasted directly within a transformation, so we turn this RDD into a dictionary first. For the next two calculations we need category and word pairs:

```
category_word_pairs = reviews_rdd.flatMap(lambda x: [((x[0], word), 1) for word in x[1]])
    .cache()
```

We use ***flatMap*** here as we have multiple words for each element in the RDD. So we get key-value pairs with the key being a tuple '(category, word)' and the value being 1. This result is again cached in memory. From this we get the count of a word per category, by using ***reduceByKey*** to sum up those key-value pairs by the key. This is the RDD on which we will apply our chi-square calculation, so it can stay an RDD. And we get the count of a word in the entire document with ***map*** we extract the word from the key and create an new key-value pair '(word, 1)'. On this we then apply ***reduceByKey*** and sum up the values by the key. This RDD we want to broadcast again, so we transform it first with ***collectAsMap*** into a dictionary.

Now we have all values to calculate chi-square with. The function to do so is defined the same as in Assignment 1 in order to obtain the same results.

```
((A * D - B * C) ** 2) / ((A + B) * (A + C) * (B + D) * (C + D))
A...Number of lines in category that contain word
B...Number of lines not in category that contain word
C...Number of lines in category without word
D...Number of lines not in category without word
```

This function is called in a transformation on the RDD of count of a word per category. ***map*** is used to apply the function onto each key-value pair. The category, the word and the count are extracted from the RDD and input into the function, as are the values of our broadcasted dictionaries and the integer line count.
On the results the ***groupBy*** transformation is used to get the results per category and then the function ***get_ output*** is mapped onto the RDD. This function prepares our calculated results for the output file. It filters out the top 75 terms and then uses the function ***get_ dictionary*** to remove the chi values and return only the words themselves. For the topterms the output string format for the file is then also created. This function is run per category, as that is the split in the RDD. Which means that the returned lines per category with the chi-values of the top terms are already returned as they should be. However, the returned dictionaries are also split up by category. On this we call our fourth action in this code, the previous ones being ***count*** and ***collectAsMap*** twice, and ***collect*** all the results, triggering code execution.
Due to this we can then extract the dictionaries and category output lines from the output. Then we sort the output lines per the first word (so the category) and then just need to merge the dictionaries together.
This is done again using RDDs. With ***parallelize*** the list of dictionaries is turned into an RDD on which we then use ***flatMap*** to split the dictionaries into single words. With ***distinct*** we remove any duplicates and then also sort the results, which we first need to ***collect*** again. Afterwards we can write the output to a text file. Iterating over the category output lines and adding our sorted and merged dictionary at the end.

Lastly we just need to stop the SparkContext again.
On the devset the runtime was around two minutes, while on the whole dataset it was around 10 hours.

3 Part 2 Datasets/Dataframes: Spark ML and Pipelines

3.1 Problem Overview

The primary goal of Part 2 is to prepare the data by converting the review texts into TF-IDF-weighted feature vectors and selecting the most relevant features using the Chi-Square method. This preparation is essential for building a text classifier in Part 3.

3.2 Methodology and Approach

For the task 2 we defined various stages of the pipeline to preprocess the text data and extract features, such as Tokenization (*RegexTokenizer()*), Stopwords Removal(*StopWordsRemover()*), TF-IDF Calculation(*CountVectorizer()* and *IDF()*), Feature Selection(*ChiSqSelector()*). We chose *CountVectorizer()* instead of *HashingTF()* because *HashingTF()* doesn't provide a way to map back to terms because of possible collisions. In the Feature Selection stage we set numTopFeatures=2000 to select top 2000 terms. The pipeline contains the following calls:

```
# Casefolding
df = df.withColumn("reviewText", lower(col("reviewText")))

# onverting category to numeric
indexer = StringIndexer(inputCol="category", outputCol="categoryIndex")

# 1. tokenization
tokenizer = RegexTokenizer(inputCol="reviewText", outputCol="words",
            pattern=r'\s+|\t+|\d+|[(){}.!?,;:+=-_"\'~#@&*%€$§\\/]+', gaps=True)

# 2. Stopwords removal
stopwords_remover = StopWordsRemover(inputCol=tokenizer.getOutputCol(),
                    outputCol="filtered_words")

# 3. tf-idf calculation with CountVectorizer
hashingTF = CountVectorizer(inputCol=stopwords_remover.getOutputCol(),
            outputCol="rawFeatures")
idf = IDF(inputCol=hashingTF.getOutputCol(), outputCol="features")

# 4. Chi-square
selector = ChiSqSelector(numTopFeatures=2000, featuresCol=idf.getOutputCol(),
                         outputCol="selectedFeatures", labelCol=indexer.getOutputCol())

pipeline = Pipeline(stages=[indexer, tokenizer, stopwords_remover, hashingTF,
            idf, selector])
```

4 Part 3 Text Classification

4.1 Problem Overview

The goal in this Part was to train a text classifier from the features extracted in Part 2 to predict the product category from review's text.

4.2 Methodology and Approach

For Task 3 we extended the pipeline from task 2 by applying Normalization with L^2 norm to the feature vectors to prepare them for classification and adding a Support Vector Machine (SVM) classifier. Since SVM is inherently a binary classifier, the OneVsRest strategy is used to handle multi-class classification. The pipeline stages are combined into a single pipeline object.

```
normalizer = Normalizer(inputCol=selector.getOutputCol(), outputCol="normFeatures")
svm = LinearSVC(featuresCol=normalizer.getOutputCol(), labelCol="categoryIndex")
ovr = OneVsRest(classifier=svm, labelCol="categoryIndex")
pipeline = Pipeline(stages=[indexer, tokenizer, stopwords_remover, hashingTF, idf,
            selector, normalizer, ovr])
```

The normalized data is then split into train (70%), test (15%) and validation (15%) sets

In order to speed up subsequent computations we created Chi-Square Selector for 500

A parameter grid is defined for tuning hyperparameters using grid search. The parameters include the number of features selected by Chi-Square, the regularization parameter, the maximum number of iterations, and whether to standardize the data.

Parameter	**Values**
selector.numTopFeatures	500, 2000
svm.regParam	0.01, 0.1, 1.0
svm.maxIter	10, 50
svm.standardization	True, False

Table 1: Parameter Grid

After the grid search, the model is evaluated with different hyper-parameter combinations by using Cross-validation to select the best model.

The strategy and the pipeline process are illustrated in the *Final Diagram*

Figure 1: Final Diagram

5 Evaluation

Our evaluation involves measuring the f1-score metric, the fit time, the train time and the evaluation time.

We did a qualitative analysis of selected example in Section 6.3.

6 Results

6.1 RDDs

As we used the exact same formula to calculate the chi-squared values in Part 1 as was used in Assignment 1, we expected to also have the same results. What needs to be ignored here are slight formatting differences between the files, mainly the presence of parentheses in the Assignment 1 output.
We found that the merged dictionaries of both outputs are exactly the same, so the same words were evaluated as important with both codes.
However, the exact chi-squared values differ slightly:

```
output.txt
"Apps_for_Android" "games:0.03218410859441414 play:0.028506244154381008 ...
output_rdd.txt
Apps_for_Android games:0.03218444638871939 play:0.028506541002803816 ...
```

In three cases this difference led to words being switched around in the ranking of importance. For Sports_and_Outdoor and Toys_and_Game the two last words in the ranking were switched.

```
output.txt
"Pet_Supplie" ... lab:0.00820362739859408 chewer:0.008202439899135733 ...
"Sports_and_Outdoor" ... molle:0.0015548678338871472 rack:0.0015546826051653494"
"Toys_and_Game" .. sculpt:0.0012126559442704125 grandsons:0.0012126057987329815"

output_rdd.txt
Pet_Supplie ... chewer:0.008202439899135733 lab:0.008200934468190486 ...
Sports_and_Outdoor ... rack:0.0015547294249244209 molle:0.0015545815289659513
Toys_and_Game ... grandsons:0.0012126057987329815 sculpt:0.0012123142077404764
```

6.2 Spark ML and Pipelines

After comparing the terms selected by the current TF-IDF and chi-square selection pipeline (output_ds.txt) with those selected in the previous task (output_rdd.txt) for the dev data, we observed that both methods identified several common terms, indicating a degree of consistency in term importance across different methods.

The output_rdd_dev.txt file, which is the result of term selection using RDD, includes a higher frequency of specific terms associated with product categories, such as "games," "engine," "diaper," "lotion," and "reading." These terms are directly connected to the product categories and show a strong correlation with their respective contexts.

However, the output_ds_dev.txt file, generated using the Spark Pipeline methods, selects a more generalized set of terms like "account," "action," "advice," "author," and "battery."

The differences in term selection show the variance between RDD-based and DataFrame-based approaches, where one prioritizes category-specific terms and the other provides a more generalized vocabulary, likely due to different tokenization, stopword removal, and feature selection techniques.

6.3 Text Classification

For the Text Classification task we have the following best model [1]:

```
fregParam=0.01, maxIter=10, standardization=True
fit_time=136.1888542175293
transform_time=5.932336807250977
evaluate_time=22.03845977783203
f1_score=0.648560079586016
```

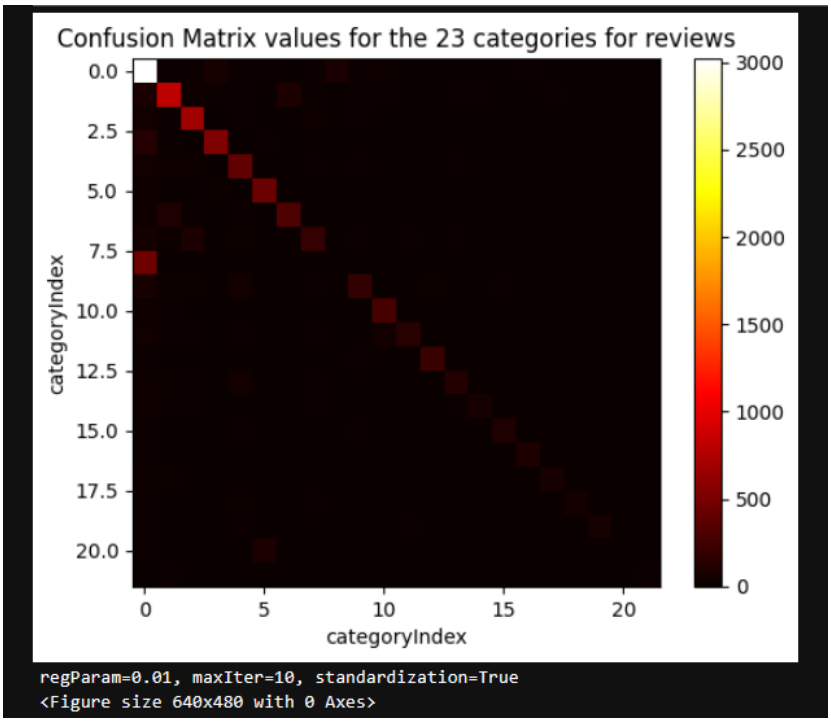

Figure 2: Heatmap for best model

7 Conclusions

This project involves three main parts: preprocessing raw text data using RDDs, converting text data to a structured format using Spark DataFrames and Pipelines, and training a text classifier using the processed data.

[1] the best and the only.

The usage of RDDs enabled efficient preprocessing and calculations by using the available transformations, which allowed grouping the data by the necessary keys easily. Figuring out how and when to use broadcast variables and at which times it was necessary to perform actions on the RDDs was a bit of a challenge at first. At times it made the RDDs feel very inefficient due to the lazy evaluation and sudden very long runtimes at the actions.

The Spark Pipeline seems to be a user-friendly abstraction of a machine learning pipeline, with a variation of tools available. Of note is the lack in flexibility of modelling, being limited to these transformations and problematic beyond them. For example, we struggled with implementing a filter for single characters, after applying the tokenizer & stopwords remover.

The task of multi-class classification of the reviews dataset proves to be an appropriate use-case for showcasing the challenges of data-intensive computing. We reached f1-score values of 60%+ for the dev set, with fit_time approx 2min, transform_time approx. 6s, evaluate_time approx 22s.

We observed that unreliable access to cluster resources made our experimental setup unreliable too, especially for high-demand contexts.