Project Report DIC Exercise 2

STEFAN ÖHLWERTHER, TU Wien 11809642, Austria
TOBIAS RAIDL, TU Wien 11717659, Austria
SAMO KOLTER, TU Wien 11810909, Austria

1 INTRODUCTION

Exercise 2 of the Data Intensive Computing course at TU Wien revolved around Apache Spark, a programming interface for large-scale data analytics. Compared to the MapReduce paradigm, which we explored during assignment 1, Spark offers a more high-level API that makes combining several data processing steps significantly easier. The subtasks of this exercise gave us an opportunity to explore Spark's RDD and DataFrame APIs, and also explore Machine Learning on large-scale data with Spark MLLib. We again use the Amazon review dataset from assignment 1.

2 PROBLEM OVERVIEW

The exercise was split into three parts with corresponding subtasks, each focusing on a different aspect of Spark.

2.1 Part 1: χ^2 Computation with RDDs

Part 1 was essentially a reimplementation of the MapReduce task from assignment 1 using Spark's *resilient distributed datasets* (RDDs). We again had to extract the terms occurring in the Amazon review dataset (across all reviews where each review is associated with one of 22 product categories), calculate χ^2 values for each term and category combination, and finally output the top 75 terms by χ^2 value for each category. Those steps were implemented as a series of RDD transformations. We were also asked to compare the obtained results with those from assignment 1.

2.2 Part 2: TF-IDF Computation Pipeline using DataFrames and MLLib

In part 2, we had our first encounter with Spark's DataFrame API, which facilitates efficient processing of structured data. A DataFrame organizes data into rows and columns, similar to SQL. However, unlike in SQL, columns can be easily added, removed and modified on-the-fly, without defining a schema beforehand. We were also introduced to the concept of *pipelines* that is used in Spark MLLib to define series of data transformations for various purposes, including the development of Machine Learning models. A pipeline is constructed as a series of *transformers* that receive one or more columns of the DataFrame (and optional additional data) as input and output the transformed data into one or more DataFrame columns. The goal of this part of the exercise was to come up with a pipeline that converts the review texts into a vector space representation with TF-IDF-weighted features (each feature being a term occurring in the corpus) and then picks the 2000 terms with the highest χ^2 value. The selected terms had to be compared with the terms selected in assignment 1.

2.2.1 *TF-IDF.*

The $TFIDF(t, d, D)$ for a given term t and document d from the corpus of all documents D is

$$TFIDF(t, d, D) = TF(t, d) \cdot IDF(t, D)$$

Authors' addresses: Stefan Öhlwerther, TU Wien 11809642, Vienna, Austria; Tobias Raidl, TU Wien 11717659, Vienna, Austria; Samo Kolter, TU Wien 11810909, Vienna, Austria.

where the *term frequency* $TF(t,d)$ is simply the number of occurrences of t in d and the *inverse document frequency* is defined as $\mathrm{IDF(t,D)} = \log \frac{|\mathrm{D}|+1}{\mathrm{DF(t,D)}+1}$, with $\mathrm{DF(t,D)}$ being the number of occurrences of t across all documents.

In layman's terms, the TF-IDF is a measure of how "important" a term is for a given document. We multiply the raw term frequency with the inverse document frequency to reduce the impact of terms that generally often occur across the whole corpus (as those are then as well less characteristic for a particular document).

Note also that this is merely how Spark implements TF-IDF (as documented here). The exact definition of TF and IDF differs between Machine Learning libraries, so results cannot be compared directly without checking the documentation for how the calculation was implemented and adapting if necessary.

2.2.2 χ^2 Feature Selection. If we want to train a standard Machine Learning algorithm for detecting the product category of a given Amazon review, we cannot just use the TF-IDF of all terms occurring in the review text, as this would result in too many features. Therefore, we were tasked with applying feature selection to retrieve the top 2000 terms based on the χ^2 test for feature independence. The feature selection algorithm would pick the top 2000 terms based on the class labels of the documents and the TF-IDFs for each term, retaining those producing the 2000 largest χ^2.

2.3 Part 3: Multiclass Text Classification Pipeline Based on TF-IDF

Part 3 extended the pipeline from part 2 to build a pipeline for training and evaluating a text classification model. It had to include a Support Vector Machine (SVM) classifier, which receives normalized(!) TF-IDF features. As SVMs are binary classifiers, the model had to be adapted accordingly to facilitate multi-class predictions. During the training process, different parameter settings had to be explored using grid search. To adhere to best practices in Machine Learning, a suitable train/validation/test split had to be done and measures had to be taken to make the results reproducible. Finally, the performance of the trained classifiers had to be evaluated.

3 METHODOLOGY AND APPROACH

We decided to implement our solutions for this exercise in PySpark and therefore provide Jupyter Notebooks as part of our submission. We developed and tested the code on the LBD cluster of TU Wien.

3.1 Part 1

The code with the solution for this part can be found in a Jupyter notebook called `1_RDDs.ipynb`. For details regarding the reasoning behind the applied data processing steps, please refer to the text cells in it. Generally speaking, the approach for obtaining the results did not change significantly compared to assignment 1 – the key-value pairs produced in the steps are mostly identical. However, a few tweaks were done and obviously, the code looks different as Spark provides convenient abstractions that don't exist in MapReduce.

3.2 Pipeline for Parts 2 and 3

In the following, all stages of the pipeline are described. A flowchart visualization can be found at the end of this document.

(1) Tokenizer. The tokenizer splits the string in the column "reviewText" into terms. Delimiters are defined as any characters other than alphabetic ones or "<", ">", "̂" and "|". Multiple consecutive delimiters are treated as one. For all terms, alphabetic characters are transformed to lowercase. A new column is added to the table named "words" which consists of all the terms.

(2). Remover. The remover adds a new column named “filtered”. This is a filtered version of the “words” column, without terms that appear in the stop words text file.

(3). Indexer. For easy mapping to category names later on, categories are given indices. This happens according to their alphabetical order in ascending fashion. To store these indices, a new column, “categoryIndex”, is established.

(4). Term Frequency. In order for the calculation of the TF-IDF, the term frequency (TF) has to be counted. Therefore, a new column “rawFeatures” is created, which per document contains all occurring terms and their count.

(5). Inverse Document Frequency. In this stage, the IDF is being calculated for each term and multiplied with the corresponding term frequency, counted in the previous stage. The result is the TF-IDF, our feature vector. It is stored in an additional column, named “features”.

(6). Normalizer. Each feature is normalized individually by the following formula, with $p = 2$ because the specification asks for an L2 normalization.

$$normalized\_vector = sum(abs(vector)^p)^{1/p}$$

(7). Chi Square Selector. The feature selection is carried out by a chi-square selector. It takes the normalized feature vectors and the category indices as input. The top 2000 features are stored in a new column "selectedFeatures".

(8). One vs. Rest Classifier. A support vector machine is defined with categoryIndices as labels and selected<features as features. The result of this classifier is binary, but in our case multiple categories exist. Therefore, a one vs. rest classifier is created with the one SVM for each category. The SVM with the highest confidence will be chosen as predictor.

(9). Hyperparameter tuning with parameter grid on TrainValidationSplit. Hyperparameter tuning on is performed an 80/20 train/validation split.

The following parameters were used for hyperparameter tuning:

(1) numTopFeatures (ChiSquareSelector): 50, 2000
(2) regParam (SVM): 0.1, 0.01, 0.001
(3) standardization (SVM): True, False
(4) maxIter (SVM): 10, 100

```
param_grid = ParamGridBuilder() \
.addGrid(css.numTopFeatures, [50, 2000]) \
.addGrid(svm.regParam, [0.1, 0.01, 0.001]) \
.addGrid(svm.standardization, [True, False]) \
.addGrid(svm.maxIter, [10, 100]) \
.build()
```

4 RESULTS

In this section, the results for each of the subtasks are presented.

4.1 Part 1: RDDs - Comparison to MapReduce approach

4.1.1 Output. The implementation of the specifications stated in Exercise 1 using RDDs produces similar results. Results for the χ^2 statistics for category and term pairs are identical. The output files differ in the ordering for terms with equal statistics based on the computation retaining six digits after the comma.

This leads to slight variations in the selected terms when the χ^2 statistic is equal for the last selected terms. The differences seem to be caused by the randomness of the computation and sorting methods employed by the different implementations.

For example, the results from Exercise 1 list the following terms for category Musical Instrument in the following ordering *xlr*, *pickguard*, and *numark* whereas using RDDs the terms are in another order *pickguard*, *numark*, and *xlr*.

Further investigation indicated that the results for Exercise 1 are in alphabetical descending order for terms with equal χ^2 values, and when using RDDs no clear pattern in the ordering could be observed.

4.1.2 Tradeoff of Performance vs. Readability? When evaluating our solution on the full dataset, we noticed a significant slowdown compared to our MapReduce solution. The code ran for approximately 1 hour and 25 minutes, while the MapReduce solution only took around 15 minutes to produce the same result.

However, we found that the programming experience with Spark RDDs was much better. The first thing we noticed is that to achieve the same result as with MapReduce (`mrjob`), less code is needed in Spark. For example, the first MapReduce job of our solution for assignment 1 (consisting of 2 jobs) which counts the number of documents per category becomes this very concise piece of code:

```
category_rdd = input_rdd \
    .map(lambda input_string: json.loads(input_string)['category'])
category_counts = category_rdd.countByValue()
```

The result of this computation can now easily be passed to the RDD, which requires it for the computation of the final output. Spark handles local variables referenced inside RDD transformations (e.g., in map functions) smartly: They are automatically broadcast to all data nodes that do the computation as long as they can fit into main memory.

Overall, it can be said that programming with Spark RDDs is much simpler than using MapReduce and tends to result in more concise, readable code. The fact that RDD transformations can easily be chained also helps.

4.2 Part 2: top 2000 terms selected by ChiSquareSelector

When comparing the obtained terms with the assignment 1, we notice that some terms do appear in both lists.

Intuitively, the selection of top 2000 terms based on χ^2 scores calculated from the extracted TF-IDF features for each term should include terms that can be considered as appropriate keywords to distinguish between texts with different types of content/categories. When skimming through the list of terms that were collected and written to `output_ds.txt` one may get the impression that this is the case. For instance, the word ‘actor’ is indeed very likely to be included in movie reviews. But in general, it is still hard to find words that one could clearly associate with one category only. The term ‘guitars’ could, for example, appear in the review of some piece of digital music or of a musical instrument.

We also tested how much overlap there is between the terms, and found out that about half of the tokens appearing in the output of assignment 1 also appeared in the output of this exercise. Vice versa, about 38 percent of tokens of the output of this exercise also appeared in the output of assignment 1.

4.3 Part 3: Text Classification

The following table presents the classifier performance on the test set across all hyperparameter settings. F1 score was used as an evaluation metric.

We noticed late into our development that the initial pipeline design was flawed. We did not use the correct input column for the one vs. rest classifier, which had the effect that all models were trained using 2000 features. Hence, the interpretation of the following results is limited.

Table 1. Classifier Performance on Validation Set for Different Hyperparameter Settings

Index	numTopFeatures	regParam	standardization	maxIter	Evaluation Metric
13	2000	0.1	True	100	0.5984936410926261
1	50	0.1	True	100	0.5986381683598597
5	50	0.01	True	100	0.5847428310202759
17	2000	0.01	True	100	0.5847428310202759
7	50	0.01	False	100	0.5321925656014915
19	2000	0.01	False	100	0.5321956439198107
11	50	0.001	False	100	0.5370540692140691
23	2000	0.001	False	100	0.5369471246616051
3	50	0.1	False	100	0.5315731057698442
15	2000	0.1	False	100	0.5316636476177978
9	50	0.001	True	100	0.5600604477127051
21	2000	0.001	True	100	0.5602350764154308
4	50	0.01	True	10	0.5820901461374169
16	2000	0.01	True	10	0.5820901461374169
0	50	0.1	True	10	0.5819229493156207
12	2000	0.1	True	10	0.5819229493156207
8	50	0.001	True	10	0.5788854651035343
20	2000	0.001	True	10	0.5788854651035343
10	50	0.001	False	10	0.3105719993622409
22	2000	0.001	False	10	0.3105719993622409
2	50	0.1	False	10	0.2246326892778781
14	2000	0.1	False	10	0.2246326892778781

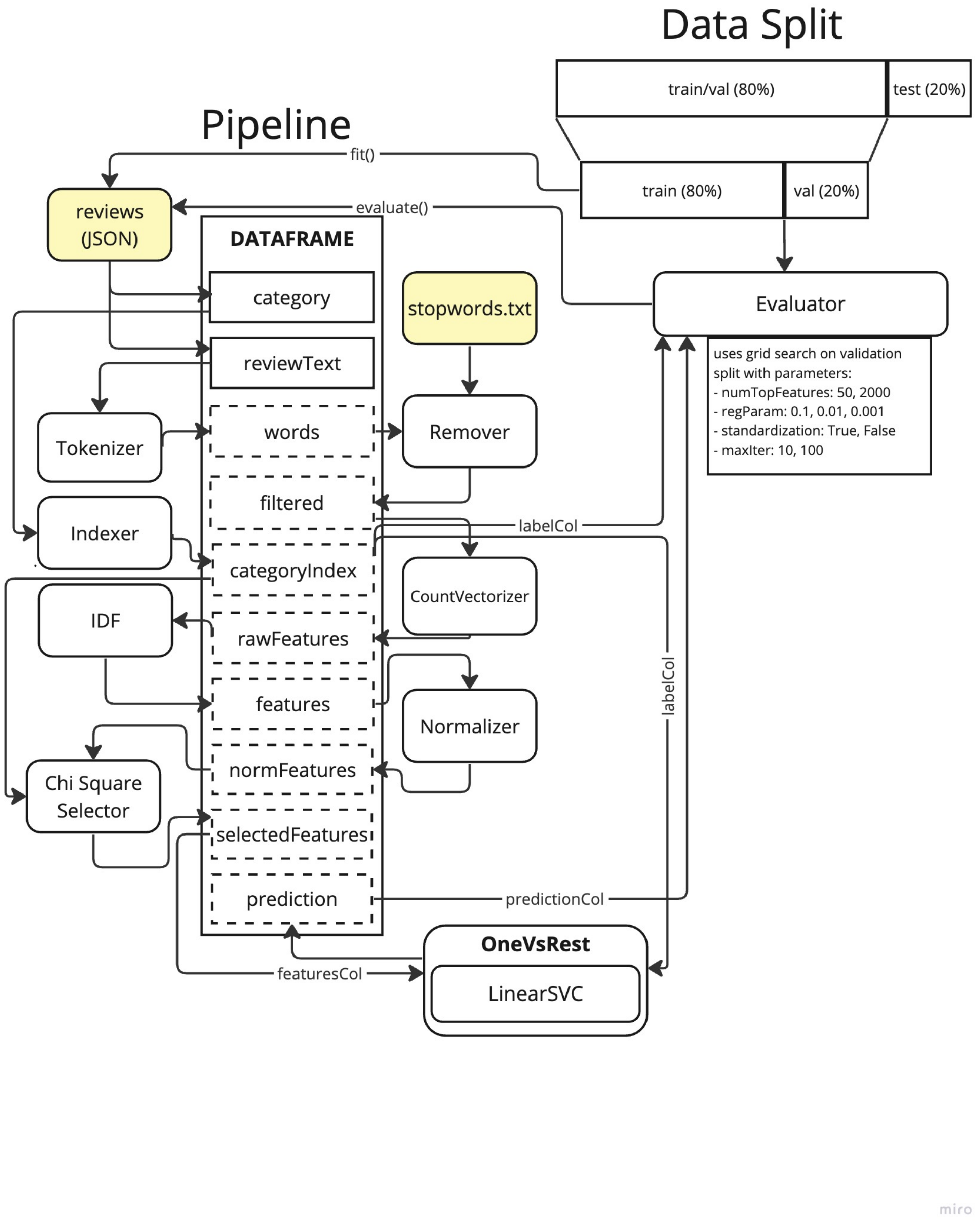
Nevertheless, it can be said that using 100 iterations for the support vector machine is superior in all cases compared to a maximum of 10 iterations. Furthermore, standardization tends to improve the quality of the model. Regarding the regularization parameter, higher values tend to produce better results.

We hypothesize that the importance of the number of top features in the chi squared selection would have had immense importance. Although, we are not sure which setting would be superior in this setting. In general, we argue that 2000 features would be prone to overfitting whereas 50 features could be insufficient in identifying minority classes and distinguishing very similar ones.

Additionally, we performed extensive investigation into the performance metrics on a per class level, as well as the patterns in misclassifications. These results are presented in the notebook and are a starting point to further improve the model. Improvements could be made by weighting the misclassification of minority classes or classes of particular interest to outweigh the dominance of some classes.

5 CONCLUSION

Overall, the exercise provided a good opportunity to explore Spark. We gained hands-on experience with RDDs, DataFrames, pipelines, and Machine Learning using MLLib. We also learned that it is always a good idea to check the correctness of one's implementation at least one day before the actual project deadline so that we have enough time to rerun experiments in case of errors like the one we identified (too late) in part 3.



uses grid search on validation split with parameters:
- numTopFeatures: 50, 2000
- regParam: 0.1, 0.01, 0.001
- standardization: True, False
- maxIter: 10, 100