VU Data Intensive Computing - Exercise 2

Alexander Lorenz (12239877)
e12239877@student.tuwien.ac.at

Tobias Grantner (12024016),
tobias.grantner@tuwien.ac.at

Maximilian Kleinegger (12041500)
maximilian.kleinegger@student.tuwien.ac.at

1 Introduction

This report presents the challenges and approaches required to perform text classifications in Spark with Big Data on the Amazon Reviews dataset[1]. Simply put, we aim to understand the differences between Spark Resilient Distributed Datasets (RDDs) and DataFrames through two parts of the exercise. Additionally, we need to train a text classification algorithm based on the review text, which allows us to dig deep into SparkML and all its features. Therefore, this report includes a description of the problem, detailed information about it, the approach taken to solve it, and a conclusion on the topic.

2 Overview

This section explains all the three parts we aim to solve with our Spark jobs. The goal of the first part is to implement the chi-square values as described in Exercise 1 using Spark with RDDs.

The second goal is to perform text classification with a Support Vector Machine (SVM) on the dataset. For this, we need to perform the following tasks in the second and third part:

- In part 2, we use built-in functions for tokenization to unigrams, case folding, stopword removal, TF-IDF calculation, and chi-square selection to build a pipeline that selects the top 2000 terms overall.
- In part 3, we extend this pipeline to perform text classification via SVM on the previously L2 normalized vector space.

The methodology, approach, and results for all parts can be found in the respective subsections of the following sections.

3 Methodology and Approach

This section describes the methodology and approach took in every part to achieve our prior defined goals.

3.1 Part 1

In this part, we needed to perform the same chi-squared calculation as in Exercise 1, but with RDDs in Spark instead of MapReduce jobs. For this, we simply transformed the MapReduce jobs we implemented in Exercise 1 into methods that RDD transformations, like *.map* and *.flatMap*, can use. This was easily done because Spark offers multiple engines, one of which is MapReduce. Therefore, we did not need to come up with a different approach to solve it. Thus, our approach stayed the same. In short, we did:

The first step pre-processes the raw input of the JSON file. We used the *.map* transformation to extract the category and the review text from each line. Furthermore, we tokenize the review text into unigrams by splitting, case folding, and filtering out duplicates, stopwords, and single-character tokens. We then emit a key-value pair for each term in the review, with the key being a tuple of the category c and the term t, and the value being 1. Additionally, the mapper emits one more key-value pair, with the key being a tuple of the category c and `None`, and the value being 1.

[1] https://cseweb.ucsd.edu/~jmcauley/datasets/amazon/links.html

In the second step, those pairs are *.reduceByKey* to count the number of documents in a category that contain a term $n_{t,c}$ and the number of documents in a category n_c, respectively, for each category c and term t.

In the third step, we *.map* each of the pairs to contain the term t as a key and the (category c, count $n_{t,c}$) as the value.

In step 4, those pairs are then *.groupByKey*, before being used to calculate the number of documents across all categories that contain a term n_t and the total number of documents n in step 5. This is done by summing up the $n_{t,c}$ values across all categories to get the total number of documents containing the term n_t. On the other hand, if the term is `None`, the values represent the number of documents n_c for the corresponding category. Therefore, by summing up all values of n_c under the key `None`, we obtain n as a side effect. These results are then emitted by a *.flatMap* under the category c as the key and the resulting (term t, $n_{t,c}$, n_t) as a value.

Lastly, after *.groupByKey* for each category, we can simply calculate the chi-square values for each term and emit the top 75 values for each category in steps 6 and 7.

As you can see, we converted our three MapReduce steps containing mappers and reducers into multiple mappers and reducers performing the same tasks in the same order. This was possible because RDDs have the same functionality, through transformations and actions.

3.2 Part 2

In this section, our main objective is to construct a transformation pipeline that encompasses both preprocessing and feature extraction of Amazon reviews in preparation for the training of an SVM text classifier in Task 3. Additionally, we apply this pipeline to the Amazon development set to compare the obtained results with those from Assignment 1.

To construct a pipeline and manage each stage effectively, we use the built-in functionalities of Apache Spark's MLlib API. Initially, the reviews are tokenized into individual words where each review is split into individual words based on a specified pattern. In the following, common stopwords are eliminated to improve the quality of the text representation. Next, the tokenized words are transformed into numerical features using the count vectorizer class, which denotes the frequency of each term within the reviews. Subsequently, feature scaling is applied using IDF (Inverse Document Frequency) to adjust the weights of features based on their significance across documents, in order to emphasize more unique terms. Next, the categories are encoded into numerical form using the StringIndexer class. Finally, a Chi-Square feature selector is employed to select the 2000 most relevant terms according to the chi-squared test.

However, certain adjustments will be implemented in task 3 to enhance efficiency during grid search evaluation.

3.3 Part 3

Lastly, the objective of this section is to use the previously constructed pipeline for feature selection and train a classifier that is able to predict the category of a review based on its text. Additionally, we optimize the performance of the classifier by trying out different configuration parameters for the feature selection and the classifier itself.

Our first approach was to simply use the chi-squared pipeline from the previous task and extend it with a one-vs-all support vector machine (SVM) classifier. But before we can start training our model, we split our data into a training and a test subset, where the training set is used to train the model, and the test set is used to evaluate the performance of the model. In order to find the optimal configuration for our pipeline, we additionally split our training dataset into two subsets, where one is used to train a model on different configurations and the so-called validation set serves the purpose of comparing the performance of one configuration against the others. To do so, we can make use of a grid-search implementation offered by Spark. Grid search is the process of evaluating all configurations on a parameter space and choosing the best-performing one. The parameters we used can be found in table 4.3

While this approach worked quite well, we were unhappy with the runtime performance it yielded. One factor contributing to the poor performance is likely that the steps of the pipeline are re-executed for every parameter combination. However, not all steps in the pipeline are affected by the changing parameters, and therefore do not have to be re-run when the parameters change. This is how we arrived at the pipeline design depicted in figure 1, where we split the pipeline into two parts, a preprocessing pipeline and a model pipeline. The preprocessing part consists of the tokenization and stopword removal

steps, while the model pipeline takes care of all steps onwards. By first applying the preprocessing pipeline to the training and test data, persisting the intermediate result and passing the preprocessed data to the Spark grid-search implementation, we were able to cut down on the runtime of the parameter optimization.

However, the previously described approach, which we submitted as our final solution, did not optimize all the steps of the pipeline, which are possible to optimize. Theoretically, all steps before the Chi-squared selector in the model pipeline could be moved to the pre-processing pipeline. In cases where the parameter defining the number of top features to select does not change, even the Chi-squared selector and the normalizer steps could be moved to the pre-processing pipeline. Unfortunately, the grid-search implementation of Spark does take separate training and validation sets as arguments, which could be pre-processed independently. It demands a single dataset, which it splits itself during the grid-search process. This hinders us from applying any step starting from the count vectorizer before the grid search since it would result in data leakage from the training to the validation dataset. For that reason, we decided to write our own grid-search implementation, which is able to pre-compute the intermediate results down to the normalizer step right before the model building and prevent re-computation for each parameter configuration. Sadly, due to the Python implementation being generally less efficient with our configuration and difficulties in parallelizing Python code, our implementation turned out still be slower than the Spark implementation.

As a result, we opted for the Spark grid-search implementation in the end, which although we have to sacrifice some optimization potential is the approach resulting in the best runtime performance.

4 Results

This section describes the results obtained in the different parts. All necessary scripts and tools used to compare and evaluate the different outputs are provided in the *evaluation.ipynb* notebook.

4.1 Part 1

In part 1, we compared the *output_rdd.txt* with the original *output.txt* to check whether we obtained different results or different chi-squared values. To easily compare them, we checked each category in both files to see if the terms have different values with a delta of *0.001*, or if we have different terms in the categories, using Python.

The result of the evaluation is that we have 6 different terms overall; therefore, 3 terms from exercise 1 are missing in part 1, and vice versa. However, all other terms present do not differ by more than the specified delta. Furthermore, all 6 terms, as you see in Table 1 for the respective categories, also do not differ by more than the delta in their values, meaning that the difference in results is simply due to the sorting of the top 75 values per category of both algorithms.

Category	**Term**	**Value**	**Missing In**
Apps_for_Android	devs	86.6496	output_rdd.txt
	glu	86.6496	output_rdd.txt
	teamlava	86.6496	output.txt
	zynga	86.6496	output.txt
Tools_and_Home_Improvement	bidet	159.7236	output_rdd.txt
	kidde	159.7236	output.txt

Table 1: Differences in Categories and Terms

4.2 Part 2

With the chi-squared selector, we chose a total of 2000 words across all Amazon reviews and retrieved their chi-squared values based on the TF-IDF metric of each word in each review.

Trying to compare this output with the result of the previous part is rather difficult. In part 1, we calculated chi-squared values according to the following formula:

$$\chi^2 = \frac{n \cdot (A \cdot D - B \cdot C)^2}{(A+B) \cdot (A+C) \cdot (B+D) \cdot (C+D)}$$

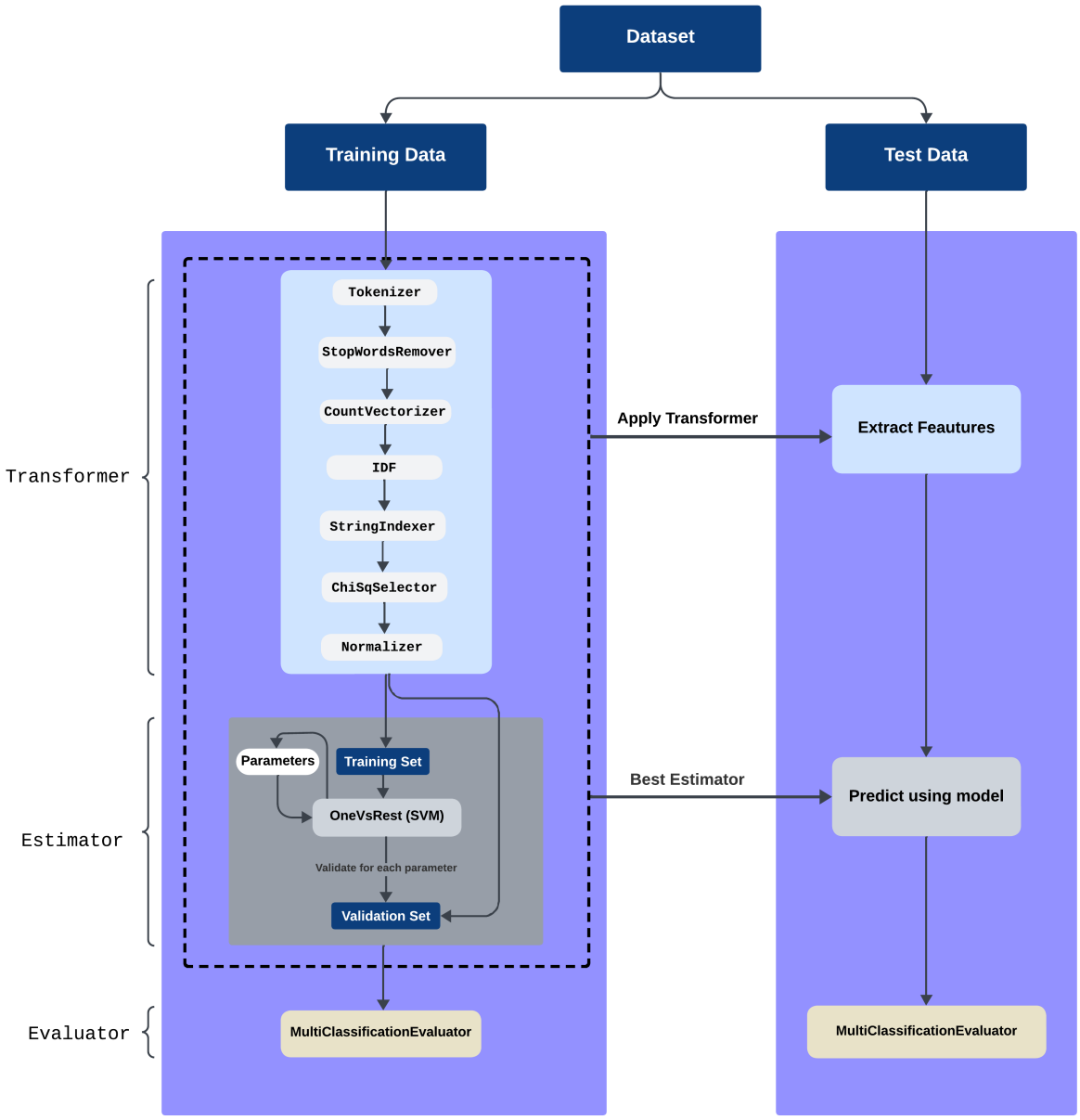

Figure 1: Machine Learning Pipeline

$n =$ total number of documents
$A = n_{t,c}$ ...number of documents in c which contain t
$B = n_t - A$ ...number of documents not in c which contain t
$C = n_c - A$ ... number of documents in c without t
$D = n - A - B - C$ ...number of documents not in c without t

Here we can see, that the chi-squared values are calculated based on metrics like the number of documents that contain a term. This means, that the frequency at which a term occurs within a document does not contribute to the resulting chi-squared values for this term. Furthermore, a term only ever gets assigned a single chi-squared value.

The chi-squared values we calculate in part 2 are based upon the TF-IDF metric. The TF-IDF, or Term Frequency Inverse Document Frequency, consists of two parts. The first being the term frequency, which as its name suggests, is the number of occurrences of a term within a single document, contributing positively to the TF-IDF value the more a term occurs in a document. Conversely, the inverse document is based on the document frequency, describing how many documents a term occurs.

A higher document frequency results in a lower TF-IDF value. Because of the way it is calculated, applying the chi-squared selector to TF-IDF features thus produces a chi-squared value for each word in every review, making it difficult to compare the obtained values to the result of the previous part.

We can however compare the set of unique words selected in part 1 and part 2. In the first part, a total of 1464 unique tokens were selected across all reviews and categories, whereas the second approach selected 2000 words. The intersection of these words, the words occurring in both sets, consists of 750 words. This corresponds to more than half of the words selected by part 1 and 37.5% of the latter part. While the two approaches select a lot of common words, the discrepancy between them is larger than we initially expected. However, the different results make sense considering the difference in calculation.

4.3 Part 3

The grid search systematically explores every predefined parameter configuration, as listed in Table 4.3, evaluating the model's performance against the validation set. For each combination, the grid search extracts the estimator mappings and validation metrics from the trained *trainValidationSplit* model to assess performance.

Parameter	Values
`svm.maxIter`	10, 50
`svm.regParam`	0.001, 0.01, 0.1
`selector.numTopFeatures`	20, 2000
`svm.standardization`	`false`, `true`

Table 2: Parameter Grid

Initially, an overview is provided by plotting the F1 scores for all 24 potential configurations and their corresponding models.

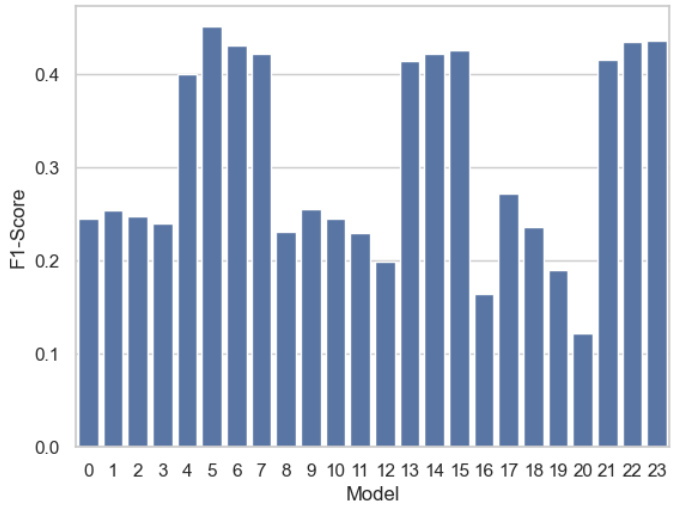

Figure 2: Model Performance Comparison

This initial comparison of model performance 2 highlights a visible trend, showing that for every 4 to 5 models tested, there's a significant improvement in performance. This suggests that every 4 to 5 models involve a varying parameter that notably enhances performance. However, to further investigate what has driven the model performance, we examine the parameters of models achieving more than 30% at F1 score.

The table indicates that all models performing significantly better than others share the same number of selected top features determined by the chi-squared selector. This could be attributed to the fact that these 2000 features cover a broader variance of actual words in the reviews, resulting into more explanatory variables. Consequently, this may lead to better differentiation and more predictive power.

To further investigate this, a plot 3 is generated to display each parameter plotted against the F1 score separately, aiming to discover further insights into the impact of the number of selected values by the chi-squared selector on model performance. Hence, color mapping is applied to see if perhaps parameters that are part of a model that performs well also have their NumTopFeatures set to 2000.

maxIter	numTopFeatures	regParam	standardization	f1-score
10	2000	0.001	False	0.399768
50	2000	0.001	False	0.451065
10	2000	0.001	True	0.430844
50	2000	0.001	True	0.422536
50	2000	0.01	False	0.413916
10	2000	0.01	True	0.422287
50	2000	0.01	True	0.426214
50	2000	0.1	False	0.415682
10	2000	0.1	True	0.435021
50	2000	0.1	True	0.435813

Table 3: Grid Search Models filtered by F1-Score larger than 0.3

The first plot shows the F1 score based on the number of selected features, where a color encoding is employed for the different values. It can be observed that with the top 2000 selected features, the F1 score mostly distributes around 0.4 to 0.45. Only two out of 12 models show significantly worse performance. Conversely, with only 20 selected features, the F1 score distributes around 0.25. It appears that a higher number of selected features contributes to better performance.

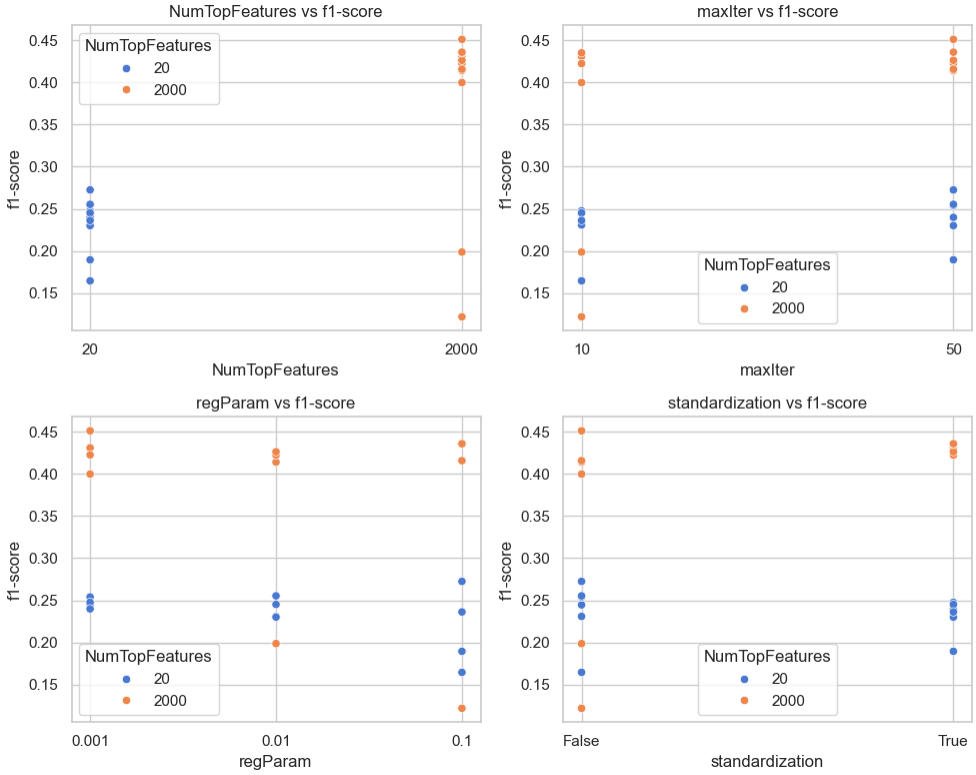

Figure 3: Model Parameters vs F1-Score

When examining the evaluations of the other three parameters, it becomes evident in each plot that the orange data points, corresponding to models utilizing 2000 selected features, consistently yield the highest F1 scores. In other words, regardless of the parameter settings for *regParam*, *standardization*, and *maxIter*, if the *numTopFeatures* are set to 2000, the model achieves relatively high performance. Conversely, models with low dimensionality perform poorly across all parameters, not exceeding an F1 score of 0.3.

Something worth noting is that the two models utilizing the promising 2000 features but still performing poorly seem to be affected by the absence of standardization, low maximum iteration numbers, and moderately regularized parameter choices.

In the final step, the best model makes predictions on the test set, achieving an F1 score of 0.47, indicating nearly a 2% improvement compared to the best model's prediction on the validation set.

In conclusion, the model's performance on the test set has remained consistent, indicating that

the dimensionality of 2000 selected features has not caused overfitting, which often leads to decreased performance. Considering that the model has been trained on a relatively small portion of actual Amazon reviews, it can cautiously be inferred that the classifier is reasonably okay, given the limited amount of data that was exposed for training.

5 Conclusion

This section summarizes all realizations we made concerning Spark and the development process around it. First, we decided to implement the algorithms in both Scala and Python to have the option of selecting the best and most efficient solution for the hand-in. From this, we concluded several things: First, the PySpark API and the Python framework are easier to use than the Scala version. Especially when searching for help with rare errors, the Python community provides better support. Secondly, even though Scala is the "native" Spark language, it is not always more efficient. Third, if you want peak performance, you need to implement it yourself to control everything and avoid mistakes, such as data leakage, through thoughtful optimizations.