Report for Assignment 2

Data-Intensive Computing SS2024

Lukas Mahler (11908553) Julian Flür (11807481)

Contents

Introduction

In this report we present our solution of the second assignment for the course Data-Intensive Computing, which contained 3 main parts: 1. Redo the first assignment using RDDs, 2. Create a vectorized TF-IDF pipeline using Spark ML and 3. Implement a SVM-based classifier on the pipeline of task 2 which predicts the category type of a review.

Problem Overview

Data set

We are again working on a data set of Amazon reviews. While there are 10 attributes, such as **helpful** or a **reviewTime**, we are only interested in two of them. Namely:

- **category**: the category that the product belongs to
- **reviewText**: the content of the review; this is the text to be processed

Each review is in exactly one category and requires no preprocessing.

The review text on the other hand has to be split into unigrams where each token is one word, those tokens are used to calculate the TF-IDF for each review. We split on whitespaces as well as a list of special characters and digits. Also all the text is cast to lower case and certain stopwords are ignored for the analysis.

Chi-square value

The chi-square value is a metric, expressing the dependence between a token and a category. Essentially the more often a term is used in a category and the less it is used in reviews from other categories the more important it is.

$$\chi^2_{\text{tc}} = \frac{N(AD - BC)^2}{(A+B)(A+C)(B+D)(C+D)}$$

Methodology and Approach

Spark is used with the following configuration when running on the server:

```
spark: SparkSession = SparkSession.builder \
    .appName("cluster") \
    .config("spark.executor.instances", 435) \
    .getOrCreate()
sc: SparkContext = spark.sparkContext
sc.addPyFile(str(BASE_PATH / "src" / "exercise2.zip"))
```

Using 435 executor instances maximizes the synchronizity for the full dataset, allowing to reduce the runtime of the notebook considerably. As the cluster does not have our package installed, we zip it and add it to Spark with the addPyFile method. Adding that zip allows us to import our package when the notebook is executed on the cluster.

Development oan the Devset is done loclly (spark runs on our machine), to reduce the load of the cluster. Developing locally also simplifies debugging, as the code is executed by a single machine instead of being distributed amongst multiple workers. Following configuration is used for local development:

```
spark: SparkSession = SparkSession.builder \
    .appName("local") \
    .config("spark.driver.host", "localhost") \
    .config("spark.driver.bindAddress", "localhost") \
    .getOrCreate()
sc: SparkContext = spark.sparkContext
```

Task 1: RDDs

We repeat the tasks of the first assignment, this time utilizing Sparks' RDDs.

Load Dataset

First, the dataset is loaded, using the previously created SparkContext, by calling the textFile method. This method loads the the given file line by line, then the map method is used to convert each line into a dictionary by passing in json.loads, which extracts a dictionary from a string in json format.

Now that we the RDD of our dataset loaded, we can calculate the number of reviews in the dataset by calling the count method on the RDD, which we will need later to calculate the Chi-square values.

Number of reviews per category

We also directly count the number of reviews per category, similar to the last assignment, where we created 2 jobs: one to calculate the number of reviews per category, and another to calculate the Chi-square values, which used the results of the first job.

Category counts are calculated in the same manner as for the first job: For each review we map a tuple of `(<category_name>, 1)`, which is reduced by `reduceByKey(operator.add)`, which calculates the sum for each category (as the category is the key). A dictionary of the calculation is retrieved by calling `collectAsMap()`.

The stopwords are loaded from the src directory into a set, as a set in Python uses a hashmap as data structure, allowing to check if a value is contained in the set in O(1).

In order to obtain the Chi-square values, category counts per token `(<token>, {<category>: <number_of_reviews>, ...})` are calculated. The list of tokens for each review is generated by splitting the reviewText with the same method as in the last exercise: We replace the special splitting characters (specified in the specification of Assignment 1) with spaces, and then replace all characters surrounded by spaces (single character tokens) also with a space. The resulting text is then splitted by the space character, yielding the unfiltered tokens for the review. The unfiltered tokens are deduplicated by using Pythons' set. We iterate over this set and return a list of tuples for each review: `[(<token>, <category>), ...]`.

The token retrieval described above is done inside the flatMap method, which yields each token and category combination separately, allowing to use Sparks' filter method instead of iterating over the tokens directly in Python, which potentially results in a better performance due to the optimization possibilities for Spark.

Each token is filtered against the stopwords set using the filter method for RDDs and then mapped using the mapValues method: `{<category>: 1}`. Mapping to a dictionary simplifies merging them in the following reduction step. We apply the reduceByKey method to the dictionaries for each key, by using the merge_dicts method we created for the last assignment. The result of the merge_dicts method is a dictionary with the number of reviews (value) for each category (key).

Chi-square calculation and filtering for top 75 tokens per category

In the previous steps we collected all information required to calculate the Chi-square values. FlatMap on the RDD containing the number of reviews dictionary for each token is applied, which allows to calculate the Chi-square values for each token while also allowing change the indexing from per token to per category. This is achieved by the calculate_chi_square_per_token method, which takes the category counts (number of reviews per category) for the current token, the category counts for the full dataset and the total number of reviews for the full dataset.

The Chi-square calculation is done in the same way as last assignment, filling in the formula with the calculated values:

```
def calculate_chi_square_per_token(
    cur_category_counts: tuple[str, dict[str, int]],
    category_counts,
    reviews_cnt):
doc_cnt_term = sum(cur_category_counts[1].values())
for category, doc_cnt_cur_term in cur_category_counts[1].items():
    a = doc_cnt_cur_term
    b = doc_cnt_term - a
    c = category_counts[category] - a
    d = reviews_cnt - a - b - c
```

```
    yield category, (cur_category_counts[0],
        calculate_chi_squares(a,b,c,d,reviews_cnt))
```

The flatMap returns `(<category>, (<token>, <Chi-square>))`, to which we then apply a groupByKey, which collects all values for the same category (key), and then apply `mapValues(list)`, which maps the grouped values to a list of tuples. We call mapValues on the result, in which we sort each list and then only return the top 75 tokens. As a last step, we sort the category order by applying sortByKey.

The 2 calls to mapValues could have been done in one step as well, which could potentially result in a slightly better performance.

Top joined tokens and output generation

Last step, we produced the RDD with the top 75 tokens per category. This RDD is converted into the first part of the output text file by converting each list of tokens into strings as specified for the assignment and then joining them together with "\n".

Calculation of the overall top tokens is done by applying flatMap, which yields only each token, the category is omitted. Those tokens are then deduplicated by calling the distinct method. The result is then sorted and collected to a list, to then be converted to a string and appended to the other result, which results in the final result, which is written to "output_rdd.txt".

Comparison to Assignment 1

The selected tokens are very similar, however the Chi-square values are considerably higher for assignment 1. Those differences however would not create a big difference for the ranking, as both apporaches yield similar top tokens for each category. For example in the category "Toys_and_Game" the tokens toys, toy, son, lego, doll, etc. appear for both approaches in the top 10.

The differences could be caused by a slightly different splitting logic or some other differences during the calculation of the chi square values.

Task 2: Spark ML TF-IDF pipeline

In task 2 we use Spark Dataframes and Spark ML to create a TF-IDF pipeline, which will be used in task 3 to predict categories.

For this task we load the dataset into a Spark Dataframe by utilizing the read.json method of SparkSession.

Pipeline initialization

To create the pipeline, we first define all stages of the pipeline by initializing the respective classes. First is the RegexTokenizer, which is used to split the review text into tokens, based on a given regex. The regex from the last exercise is once again reused, as it exactly specifies the characters used to split the text.

We also set minTokenLength to 2 to automatically exclude tokens of a single character, which is required by the speicification and previously had to be done by an additional processing step.

In the second step of the pipeline we add a StopWordsRemover, which removes often used terms which only act a noise as they are regularly used to express meaning in the language (e.g. the, is, ...). This step replaces the loading of the stopwords textfile and filtering the tokens by that textfile.

After the first two steps we have the cleaned tokens, but in order to use the 'category' column for Chi-square calculation, we need to add an StringIndexer, which converts the values of 'category' into numerical values, which will be needed in a later step.

Our tokens need to be converted to numerical values, in order to be processable as features for a machine learning model. One way to transform them is calculating the TF-IDF values for each token. In order to obtain the term frequencies (the TF part), we create a CountVectorizer, which maps each token value to a number and counts the occurences of each token in the review.

To convert the TF values into TF-IDF, we need to weight the term frequencies by their inverse document frequency (IDF) which reduces the influence of tokens, which appear in many reviews and increases the influence of tokens

which appear in only a few reviews (special terms usually have more influence to the meaning than filler words). We weight the TF values by using an instance of IDF (the Spark class).

The last step is selecting the top 2000 overall tokens by utilizing Chi-square calculation. For that, we use ChisSqSelector with the argument of numTopFeatures set to 2000. With this option, the selector only uses the top 2000 terms (by their Chi-square value) and omits the rest. Alternatively, the selector could also select e.g. the top x% of tokens.

Pipeline fitting and token extraction

We combine all steps of the pipeline by initializing a Pipeline object with a list of all steps as the argument for stages. Using this configuration, we can run all stages with a single call to fit and then to transform instead of having to call each stage separately.

Once the dataframe is transformed accordingly, we extract the top 2000 tokens as required by the assignment specification. However, extracting the tokens is not that simple, because the tokens have been transformed to numericals by the CountVectorizer.

In order to obtain the tokens we first extract the top 2000 features as selected by ChiSqSelector (a list of 2000 integers), which are converted back by looking up their string value using the vocabulary produced by CountVectorizer when transforming the tokens.

Once they are converted, the tokens are written to the file 'output_ds.txt'.

Comparison to Task 1

The ds file contains more tokens than the rdd file, which makes sense as for the ds file 2000 tokens were selected, for the rdd file only $75 * 24$ tokens where selected, which also were deduplicated. Both documents share many tokens, especially those, which have a big impact of the category of the review. For example following tokens: action, flavor, game, horror.

One of the differences between both approaches is, that for the rdd file very specific terms, which only differentiate between one category and rest had a better chance to end up in the file, as the only condition was whether the token is good to identify that single category. On the other hand for the ds file, all categories are combined and terms which differentiate well between multiple categories had a better chance. This also explains some of the differences between both files.

Task 3: SVM Classifier

The third and final task is to fit a SVM Classifier to th preprocessed data. Pyspark calls such this machine learning model `LinearSVC`. To this end we will fit a pipeline which enabeles us to fit the binary classifier to our multiclass problem.

First we create an `LinearSVC`object. When doing so we can specify the columns for the features and the labels and other parameters, e.g. maximum iterations.

After the fit method is called and the DataFrame Pysparkis provided. The resulting object provides us with a method to predict the classes for the training as well as the test set.

One vs. Rest

In order to solve the issue of a multiclass classification task with MLlib a binary classifier is fit with the One vs. Rest approach.

For each class we train a classifier which predicts a score for each class against all others. The prediction is computed by choosing the class which achieves the highest score. This means for our 22 classes of reviews we train 22 classifiers and for the prediction 22 scores are calculated.

The API for the `OneVsRest` object is the same as for the classifier.

First we create the object with the binary classifier as parameter, then we call the fit method to for the training data.

Grid Search

To tune the hyperparameters we aim to perform a grid search with cross validation. The MLlib implementation of this is `CrossValidator`. However this does not work with `oneVsRest` and `LinearSVC`.

To this end we had to use a simpler approach with for loops. However we showcase the approach we have the code included for a logistic regression model.

PCA

We also compare the model to a model with greatly reduced number of features. This is achieved by calculating 10 principal components. The PCA is performed in the preprocessing step already, since it is used with all models to compare.

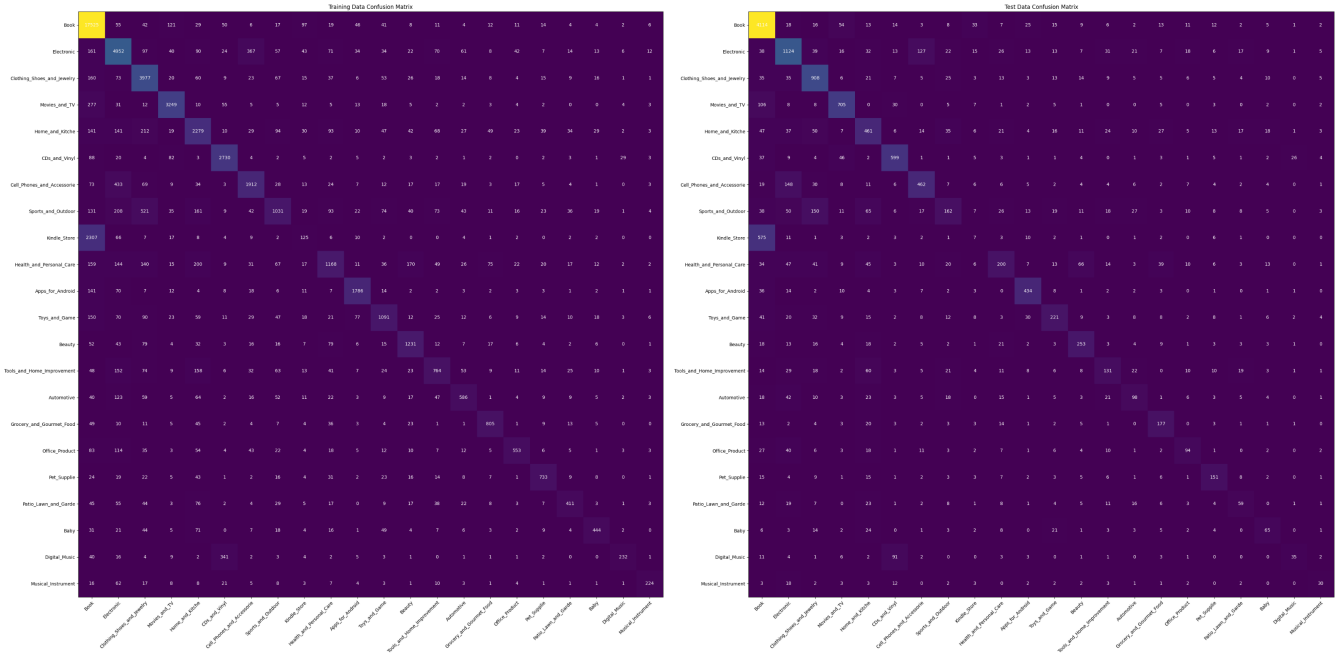
Conclusions

Result

The numbers below are given for the devset, however on the complete data set the results are compareable.

When having a look at the classifiers it can be said that they performed well without PCA.

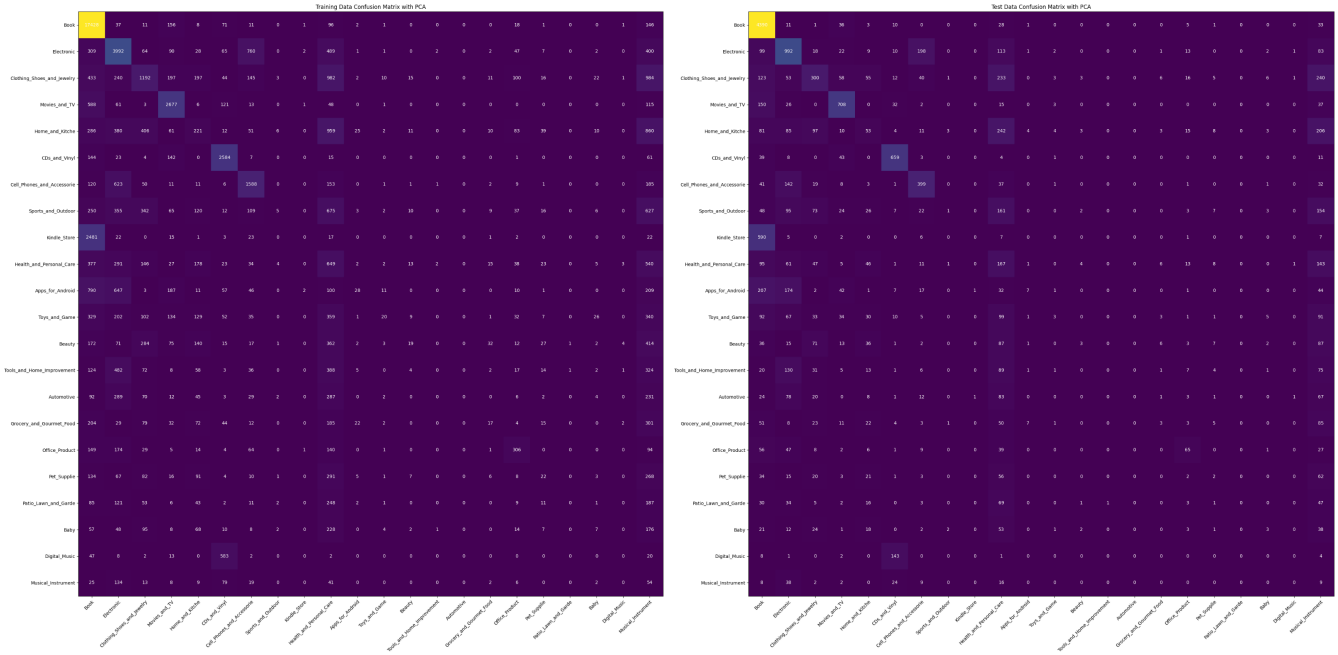
In the confusion matrices it becomes obvious that the most of the classes are easily distinguished. Most of the confusions stem from categories which naturally overlap.



Other more frequent missclassifications are in cellphone accessories and electronics or sports/outdoors and clothing/shoes/jewlery.

PCA	training data	test data
no	0.736	0.646
yes	0.431	0.432

With PCA the F1-score significantly drops but we have to keep in mind that we only 10 features, which is a significant decreas in complexity.



Checking the Confusion matrix above there are some obvious categories which are missclassified very often. Still there are the obvious missclassification due to related classes like books and kindle content.

Runtime

In the table below the runtime is given.

task	runtime
Task 1: Assignment 1 on RDDS	4 min 50 sec
Task 2: TF-IDF pipeline	19 min 05 sec

For the third task the time could not accurately be measured due to the high load on the provided server.