Data-intensive Computing: Assignment 2

194.048 - SS2025

Group 49: Tobias Huber (11904665), Jonas Kruse (12434740), Tim Greß (12412672), Gabriel Kitzberger (12024014), Niklas Kessler (12434735)

1 Introduction

In this exercise, we implemented and evaluated a multi-stage text processing and classification pipeline using Spark. The foundations for this assignment were already laid in the previous assignment. The primary objective is to explore data processing techniques for processing large text corpora. The tasks include calculating chi-square statistics to identify informative terms, transforming text data into TF-IDF feature vectors, and training a SVM classifier for multi-class product category prediction.

2 Problem Overview

2.1 Dataset

Similar to assignment 1, we are using the Amazon Review Dataset 2014 for this exercise again, which contains more than 140 million reviews of products from 24 categories. It is given as a large JSON file, where each entry contains review text and product category, as well as other, not needed infor- mation like timestamps or helpfulness. The dataset was already provided in a shared folder on the Hadoop file-system.

2.2 Main Computational Task

2.2.1 Part 1: RDDs

Since part 1 is similar to assignment 1, we will refrain to rewrite the problem again but refactor the description of assignment 1: With "words," in this context, we actually refer to unigrams. Through several preprocessing steps such as tokenization, lowercasing, and stopword removal, we obtain a list of unigrams for each review.

Based on these preprocessing steps, we are to calculate the chi-squared scores. With

- **A**: Number of reviews in the given category containing the word.
- **B**: Number of reviews not in the given category but containing the word.
- **C**: Number of reviews in the given category not containing the word.
- **D**: Number of reviews neither in the category nor containing the word.
- **N** $= A + B + C + D$.

the chi-squared score of a unigram-category pair is given as:

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

The task of this exercise can be broken down to the following core computational steps:

1. **Calculating A, B, C, D, N:** For each unigram-category pair, we calculate the variables A, B, C and D as defined above.

2. **Calculating Chi-squared scores:** With the obtained variables and the formula above, we calculate the chi-squared scores.
3. **Ordering the terms:** Based on the calculated scores, we sort the terms and preserve the top 75 terms for each category.
4. **Merging the lists:** Finally, we additionally merge the lists, to obtain a dictionary with all terms.

2.2.2 Part 2: Datasets/DataFrames: Spark ML and Pipelines

The goal of this part is to prepare a vector space representation of the review texts with TF-IDF-weighted features by setting up a pipeline. We are using Spark Dataframe/Dataset API for that. Finally, we want to select the most informative terms again using chi-squared feature selection.

This task can be broken into the following steps:

1. **Tokenization** The review texts are split into individual tokens by using a set of given separators.
2. **Stopword Removal** Common stopwords are removed using the provided list of stopwords.
3. **TF-IDF calculation** The preprocessed tokens are then transformed into a vector space using TF-IDF (Term Frequency, Inverse Document Frequency).
4. **Chi-Squared Selection** Using a chi square test the top 75 discriminative features are selected.

2.2.3 Part 3: Text Classification

The next step is to fit a model, which should be able to predict the product category from a given review text.

This task can be broken into the following steps:

1. **Extend Pipeline of Part 2** We extend the previously set up pipeline with a normalization (L2) and then a SVM classifier for the actual classification
2. **Data Splitting** The data is split into train, validation and test set.
3. **Reproducibility** Set the seed values accordingly so that the results can be reproduced.
4. **Model Tuning with Grid Search** Grid search is used to find suitable hyperparameters. The base line (Top 2000 Features) are then compared with a more strict feature reduction. Additionally, different SVM hyperparameters are compared.
5. **Evaluation** The model performance is then assessed using the MulticlassClassificationEvaluator and F1 as a metric.

3 Methodology and Approach

3.1 Part 1: RDDs

Now, we take a look at the used methodology, this is quite close to the problem overview as this already provides a quite good indication on how to approach the issue.

3.1.1 Loading the data and preperation

Using pyspark the data is loaded into a RDD from the Hadoop file system. Additionally, the stopword file is loaded. Then the preprocessing including tokenization and stop word removal is done.

```
re_split = re.compile(
    r"[\s\t\d\(\)\[\]\{\}\.\!\?\,\;\:\+\=\-\_\"\'‘\~\#\@\&\*\%\€\§\\\/]+"
)
stop = load_stopwords(stopwords_path)
stop_bc = sc.broadcast(stop)

def clean_tokens(row):
    ...

    tokens = set()
    for token in re_split.split(text):
        if token and len(token) > 1 and token not in stop:
            tokens.add(token)
    return [(cat, t) for t in tokens]

data = rdd_json.flatMap(clean_tokens).persist()
```

3.1.2 Calculating A, B, C, D, N

The preprocessed data are then used to calculate the respective terms required for calculating the chi-squared scores.

The following code fragment is exemplary for these calculations:

```
term_count = (
    data
      .map(lambda x: (x[1], 1))
      .reduceByKey(lambda a, b: a + b)
      .collectAsMap()
)
```

The results then need to be broadcasted so that the variables can be used on each cluster:

```
term_count_bc = sc.broadcast(term_count)
cat_count_bc = sc.broadcast(cat_count)
N_bc = sc.broadcast(N)
```

3.1.3 Calculating chi-squared scores

Once the values are available, the chi-squared score can be calculated.

```
def chisq(record):
    ...
    numerator =  N * (A*D - B*C)**2
    denominator = (A+B)*(A+C)*(B+D)*(C+D)
    chi2  = numerator / denominator if denominator else 0.0
    return (cat, (term, chi2))
```

3.1.4 Ordering the terms

In the next step, the terms are ordered and the top 75 are kept. Also the categories are ordered alphabetically.

```
K = 75
top_per_cat = (
    rdd_chisq
      .groupByKey()
      .mapValues(lambda it:
                 sorted(it, key=lambda x: -x[1])[:K]).persist())
category_lines = (
    top_per_cat
      .map(lambda ct: (ct[0], " ".join(f"{t}:{c:.4f}" for t, c in ct[1])))
      .sortByKey() # alphabetical order
      .map(lambda kv: f"{kv[0]}\t{kv[1]}").collect())
dict_line = (
    top_per_cat
      .flatMap(lambda x: [t for t, _ in x[1]])
      .distinct()
      .sortBy(lambda x: x).collect())
```

This is then stored in a textfile in the same format as we had for exercise 1.

3.2 Part 2: Datasets/DataFrames: Spark ML and Pipelines

In this part a pipeline is set up, which includes the steps for chi-squared score calculation and its' predecessors accordingly.

3.2.1 Tokenization

The first step is to tokenize the review texts, this is done using the built-in RegexTokenizer, which is considering the a set of provided separators.

```
tokenizer = RegexTokenizer(inputCol="reviewText", outputCol="tokens",
    pattern="[\s\t\d\(\)\[\]\{\}\.\!\?\,\;\:\+\=\-\_\"\'‘\~\#\@\&\*\%\€\$\§\\\/]+")
tokenized = tokenizer.transform(df)
```

3.2.2 Stopword Removal

The tokenized review texts are further on cleaned by removing the provided stopwords accordingly. Pyspark provides a built-in stopwords remover which is used considering the provided stopword file.

```
stopper = StopWordsRemover(
    inputCol="tokens", outputCol="tokens_filt",
    stopWords=load_stopwords()
)
```

The result is a set of token for each review like: [gift, husband, nice, ...]

3.2.3 TF-IDF calculation

The next step is to count the tokens and calculate the TF-IDF. This is done in two steps, first we use the built-in CountVectorizer to count the tokens and provide them in a vectorized form.

```
tf = CountVectorizer(
    inputCol="tokens_filt", outputCol="tf",
```

```
    vocabSize=20_000, minDF=5
)
```

This provides a vector with the number of occurences of each token. This is an array similar to the following example:

- $[a, b, c] \rightarrow (3, [0, 1, 2], [1.0, 1.0, 1.0])$
- $[a, b, b, c, a] \rightarrow (3, [0, 1, 2], [2.0, 2.0, 1.0])$

In the given problem there are by far more than three tokens, why it looks like this:

- $(96130, [2, 3, 7, 8, 3...$
- $(96130, [0, 1, 3, 21, ...$

This vectorized token counts are further on used for the IDF model, which is again PySpark built-in. The array of token counts is used to fit the IDF model.

```
idf = IDF(inputCol="tf", outputCol="tf_idf")
```

3.2.4 Chi-Squared Selection

The features calculated IDF values are then used in the built-in ChiSqSelector to filter for the top 75 features by chi-squared scores.

```
chisq = ChiSqSelector(featuresCol="tf_idf", outputCol="selected_features", labelCol="label",
    numTopFeatures=2_000)
```

3.2.5 Pipeline

Using the previously instantiated classes, we now create a pipeline.

```
feature_pipe = Pipeline(stages=[tokenizer, stopper, tf, idf, label_indexer, chisq])
feature_model = feature_pipe.fit(df)
```

This pipeline we save so we can extend it in Part 3.

```
USER = getpass.getuser()
SAVE_PATH = f"hdfs:///user/{USER}/models/feature_pipe_part2"

feature_model.write().overwrite().save(SAVE_PATH)
```

3.3 Part 3: Text Classification

For computational reasons we executed our code on a downsampled version of the development set (5%).

3.3.1 Extend Pipeline of Part 2

In our code we are directly using the pipeline that was created earlier in exercise 2.

```
 feat_model = PipelineModel.load(PIPE_PATH)
```

3.3.2 Variance Threshold Selection

We chose VarianceThresholdSelector as a comparison method because it provides an unsupervised way to reduce feature dimensionality by removing low-variance features that are unlikely to be informative. This allows us to evaluate whether a simple, model-agnostic filter can outperform a supervised method like Chi-squared selection in terms of classification performance.

We created one pipeline for each selection-model.

The pipeline is first extended with an Normalizer. PySpark's built-in normalizer is used for that.

```
normalizer = Normalizer(inputCol="selected_features", outputCol="normalized_features", p=2.0)
```

Additionally, the built-in support vector classifier is instantiated accordingly.

```
svm = LinearSVC(featuresCol="normalized_features", labelCol="label", predictionCol="prediction")
```

These are then extending the pipeline. To use the different filter methods, we create a function to run both pipelines, one for each filter method.

```
def run_tvs(pipe, param_grid, name):
    tvs = TrainValidationSplit(
        estimator=pipe, estimatorParamMaps=param_grid, evaluator=evaluator,
        trainRatio=0.8, seed=SEED)
```

3.3.3 Data Splitting

To provide a base for later evaluations, the data set is split into a train, validation and test set.

```
train_data, temp_data = df.randomSplit([0.7, 0.3], seed=seed)
validation_data, test_data = temp_data.randomSplit([0.5, 0.5], seed=seed)
```

3.3.4 Reproducibility

To ensure that further execution lead to similar results, the execution is made deterministic using a fixed seed. This seed is also used in the data split above.

```
seed = 42
random.seed(seed)
np.random.seed(seed)
```

3.3.5 Model Tuning with Grid Search

For model tuning grid search is used, this means that for different hyper parameters there is a set of different values. Each combination of values is used to fit the model and evaluated on the validation set.

```
param_grid = ParamGridBuilder() \
    .addGrid(svm.regParam, [0.01, 0.1, 1.0]) \
    .addGrid(svm.standardization, [True, False]) \
    .addGrid(svm.maxIter, [10, 50]) \
    .build()

cv_2000 = CrossValidator(
    estimator=pipeline_2000, estimatorParamMaps=param_grid,
    evaluator=evaluator, numFolds=3, seed=seed
)
```

3.3.6 Evaluation

The evaluation is done using the model with the best result obtained from the grid search. The F1 score is then calculated on the test set.

Table 1: Comparison of feature selection strategies using TrainValidationSplit

Selector	**Val F1**	**Test F1**	**Best Parameters**
ChiSqSelector	0.3612	0.3814	`maxIter=10`, `regParam=0.1`, `standardization=True`
VarianceThresholdSelector (VTS)	0.4068	0.4202	`maxIter=5`, `regParam=0.1`, `standardization=False`

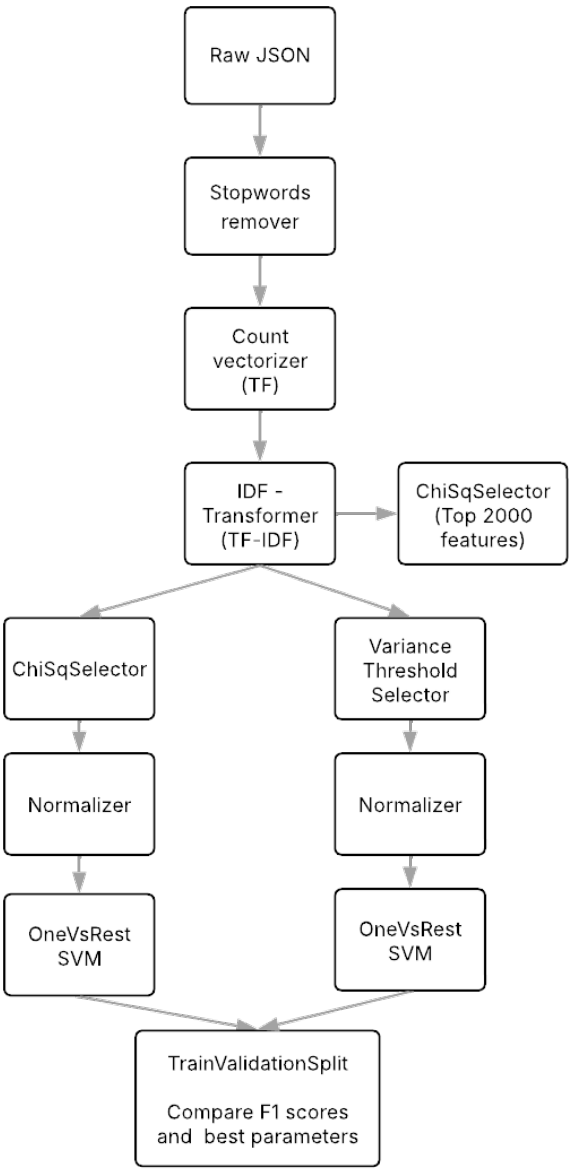

Figure 1: Pipeline Illustration

4 Comparison of the Output Files

When we look at our output files, the first one, `output.txt` (from our earlier MapReduce work), and the next one, `output_rdd.txt` (from our Spark RDD work), are quite *alike*. Both of these files give a detailed list of the top 75 important words for each product category, showing their special χ^2 scores. They also both end with a combined list of all these important words, usually sorted alphabetically. The main way they might look different is just in how this information is arranged; for example, `output.txt` often puts all words and scores for one category on a single line, while `output_rdd.txt` might list each top word on a new line with details like its rank, and its combined dictionary words might also be on

new lines, perhaps marked with "DICT". However, the third file, `output_ds.txt`, which we made using Spark DataFrames, is very *different*. This file *only* gives the combined list of important words that a tool called `ChiSqSelector` picked out; it doesn't show the top words and their scores for each separate category like the other two do, just one line of sorted words. So, while `output.txt` and `output_rdd.txt` give both the detailed scores for each category and a summary word list, `output_ds.txt` just offers the summary word list, and it's good to remember that the actual words found in these lists might not be exactly the same because we used different methods to find them.

In total we see that there are 750 terms in the intersection between the first exercise and the output of part 2. There are 750 terms in the terms of exercise 2 that are not in the output in exercise 2.

5 Conclusion

In this whole assignment, we tried out different ways to work with a large set of Amazon reviews. Our main goals were to find the most important words that describe each product category and then use what we found to teach a computer to sort text into these categories.

First, like in "Part 1", we used a method called MapReduce. This involved getting the review text ready by breaking it into words, removing common unimportant words, and counting them. A trick we used was to count words on each machine first to speed things up by sending less data over the network. Then, we calculated special scores (called χ^2 scores) for words to see how important they were for each category. From these scores, we picked the top 75 words for each category.

Next (in "Part 2"), we switched to using a tool called Spark. First, we did the same word-finding task using something in Spark called RDDs. This was much like the MapReduce way, involving similar steps of preparing the text, counting words, calculating χ^2 scores, and picking the top terms.

Then, we tried another Spark tool called DataFrames and ML Pipelines to do the same χ^2 analysis. This was a more modern way to tell Spark what to do, using steps like `RegexTokenizer` (to break up text), `StopWordsRemover`, `CountVectorizer` (to count words), `IDF`, and `ChiSqSelector` (to pick important words) all connected together.

Finally, the last part of the assignment was to build a model that could sort text into different categories automatically. We used a method called Linear SVM for this, again using Spark's ML Pipelines. This involved preparing the text data in special ways (like TF-IDF and picking the best words with `ChiSqSelector`), training the model, and finding the best settings for it using `TrainValidationSplit`. We then checked how well it worked on new data using a measure called the F1 score.

So, through this whole assignment, we learned a lot about different ways to handle big data and teach computers tasks like finding important words and sorting text. The methods we used showed they can handle large amounts of data well. This helped us understand which words are important for different product types and how well our text-sorting model worked.