Data Science

Data-intensive Computing
Text Processing and Classification using PySpark

Outline

Introduction

- In this project we will apply Spark to process large text corpora
- We will be using *Amazon Review Dataset*
- The goal of this project is to learn a **LinearSVC** model that can predict the product category from a review's text
- This project is divided into three parts:
    - Calculating chi-square values using RDDs and transformations
    - Create a Spark Machine Learning Pipeline to pre-process the data
    - Extend the pipeline such that a **LinearSVC** classifier is trained

- The tasks were implemented using **PySpark** and **jupyter-notebook 6.0.3** in **18-node Hadoop Cluster** at TU

Characteristics of dataset

- *Amazon Review Dataset* has around **70** million records and **10** features.
- The dataset is in JSON format and the features are:
  - **reviewerID** - string - the ID of the author of the review
  - **asin** - string - unique product identifier
  - **reviewerName** - string - name of the reviewer
  - **helpful** - array of two integers [a,b] - helpfulness rating of the review: a out of b customers found the review helpful
  - **reviewText** - string - the content of the review; this is the text to be processed
  - **overall** - float - rating given to product asin by reviewer reviewerID
  - **summary** - string - the title of the review
  - **unixReviewTime** - integer - timestamp of when review was created in UNIX format
  - **reviewTime** - string - date when review was created in human readable format
  - **category** - string - the category that the product belongs to

Part 1 – RDDs and transformations (1)

This part of the project was done in two steps:

- **Step 1: Tokenization, case folding and removing stop words** – a function is created that will be called in *flatMap()* function for every row. For each line the function will set all the line to lower case (for case folding) and than after substituting the delimiters like whitespaces, tabs, digits, and common delimiter characters (.!?,;:()[]{}-_"`~#&*%$\/) we will get only the unique words. Each word will be checked if it is a stopword before returning. This function will return three different outputs:
  - **(categoryCount-category, 1)** will count categories
  - **(wordCount-word, 1)** will count the occurrence of unique words in reviews
  - **((category,word), 1)** will count the number of reviews in category which contain the word

  After we will **reduceByKey** the output of the mapper by summing the values

  To store **categoryCount-** and **wordCount-** values, we will use **collectAsMap** action

Part 1 – RDDs and transformations (2)

- **Step 2: Calculate the chi-square values** - we will calculate chi-square values for each **(category,word)** tuple. To achieve this we created a function called chi_square_calc.
    - The function will output: **(category, word:chi-square)** tuples which we will **reduceByKey()** in order to group them by category and sort them
    - Then we will use **mapValues()** to sort the chi-square values in the descending order, but we first need to split the values we concatenated in the **reduceByKey()**, and also to take top 200 values for each category
    - In the end we will join categories with their **(word:chi-square)** tuples by space-separating them

Part 2 - Pre-processing

- **RegexTokenizer** from *pyspark.ml.feature* is used for tokenization to unigrams of the text in *reviewText* feature using whitespaces, tabs, digits, and common delimiter characters (.!?,;:()[]{}-_"`~#&*%$\/)
- **StopWordsRemover** from *pyspark.ml.feature* is used for stopword removal and case-folding
- To convert word tokens from the above steps to a classic vector space representation with **TFIDF-weighted** features, we used:
  - **CountVectorizer** from *pyspark.ml.feature*, to calculate **TF** - term frequency (*the number of times that term f appears in document d, while document frequency DF is the number of documents that contains term t*)
  - **IDF** from *pyspark.ml.feature* , to calculate **TFIDF = TF * IDF** (*where IDF is the inverse document frequency*) that we will need for the *chi-square calculation*
- **StringIndexer** from *pyspark.ml.feature* is used to index category feature in order to use it in the next step
- **ChiSqSelector** from *pyspark.ml.feature* is used to calculate *chi-square* of the features selected by TFIDF (using top **4000** features)

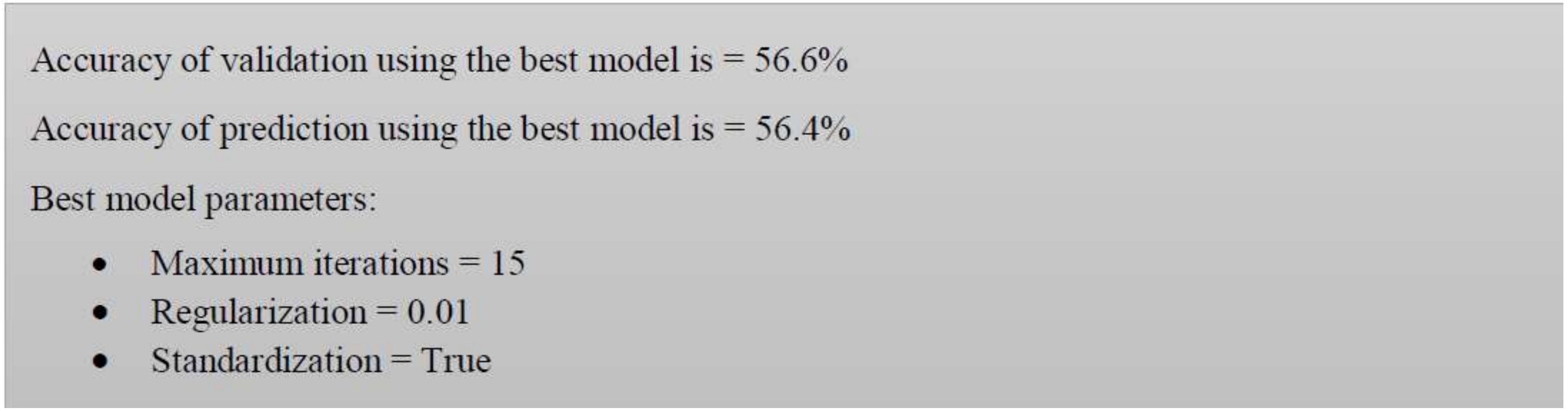
All these steps are added as stages to the Spark ML Pipeline that we will create in this first part of the project.

Part 3 - Text Classification (1)

- **TFIDF** features extracted in the first part of the project will be normalized using **L2 Normalizer** from *pyspark.ml.feature*
- Since we are dealing with multi-class problems, together with **LinearSVC** classifier, we used **OneVsRest** classifier from *pyspark.ml.classification*.
- We extended the pipeline created in the first part by adding **Normalizer**, **LinearSVC** and **OneVsRest** to it
- In order to make experiments reproducible we set a random seed number
- Data is splitted into three different sets - **train**, **validation** and **test**.
  - 80% of the data is used for training and 20% for testing
  - We further split the training set into 80% for training and 20% for validation. That leaves us with **64%** of the entire dataset for training, **16%** for validation and **20%** for testing.
- In order to apply different parameters to the **LinearSVC** classifier we used **ParamGridBuilder** from *pyspark.ml.tuning*
  - **maxIter** = 10, 15
  - **regParam** = 0.001, 0.01, 0.1
  - **standardization** = True, False

Part 3 - Text Classification (2)

- In order to find the best hyperparameters for our model, we used **TrainValidationSplit** from *pyspark.ml.tuning*
  - It is less expensive than **CrossValidation** because evaluates each combination of parameters only once and will produce reliable results in the size of our training set
  - In order to evaluate the performance of the trained classifiers, we used **MulticlassClassificationEvaluator** from *pyspark.ml.evaluation* with **f1 measure**
  - We set **trainRatio=0.8** so that 80% of the data will be used for training and 20% for validation
- Best model from **TrainValidationSplit** is used to predict the test set. Results are below:



Accuracy of validation using the best model is = 56.6%

Accuracy of prediction using the best model is = 56.4%

Best model parameters:

- Maximum iterations = 15
- Regularization = 0.01
- Standardization = True

- *Note: only 15000 random rows from the dataset are used because of the time spark jobs took to finish.*

Additional requirements

- As intermediary tasks of the project were also:
    1. Produce a file that contains all the terms selected from **ChiSqSelector**
    2. Using **RDDs and transformations**, output a file containing the following:
        - One line for each product category (categories in alphabetical order), that contains the top 200 most discriminative terms for the category according to the chi-square in descending order, in the following format: <category name> term_1st:chi-square_value term_2nd:chi-square_value ... term_200th:chi-square_value and
        - One line containing the merged dictionary (all terms space-separated and ordered alphabetically)

- The output of the first task was to be compared with the last line of the generated output of the second task containing the merged dictionary. Brief observations from the comparison are:
    - In the first file we select top **4000 features** with the best chi-squares in the whole dataset, meaning that we can have different number of features per category (and not always **200**, like in the second file). Therefore, these two files differ. While in the second file we have **3605** unique words, only **~1300** words are also in first file. The remaining (**~2700**) words are different.

Jobs

- Because of the large amount of the data, notebooks created for the **part 2** and **part 3** of the project were run in terminal
- In order to run files in terminal, we had to convert jupyter-notebook files to python files using the following commands:
  - **jupyter nbconvert Part2.ipynb --to script**
  - **jupyter nbconvert Part3.ipynb --to script**
- To execute those python created files via spark-submit, the following commands were used:
  - **spark-submit --executor-memory 8G --num-executors 4 --total-executor-cores 16 --conf spark.ui.port=5051 Part2.py**
  - **spark-submit --executor-memory 8G --num-executors 4 --total-executor-cores 16 --conf spark.ui.port=5051 Part3.py**