DSSP Data Camp 6-7 April 2020
"Large Scale Text Mining with Spark and Python"

The Main Task

Christos Giatsidis, Apostolos N. Papadopoulos, Michalis Vazirgiannis

A. INTRODUCTION

The area of *text analytics* (or *text mining*) includes techniques from multitude scientific areas (A.I., statistics, linguistics), and it has a wide range of applications (security, marketing, information retrieval, opinion mining).

While structured data are "ready" for analysis and evaluation, unstructured text requires transformation in order to uncover the underlying information. The transformation of unstructured text into a structured set of data is not a straight forward task and text analytics offers a wide variety of tools to tackle with the idioms, ambiguities and irregularities of natural language.

In this data camp, we will tackle a problem that involves **text analysis and feature extraction from text data**. The particular task is focused on a multi-label classification task. This is a task where each instance of the data may be assigned multiple labels. This is different than a multi-class task where each instance may be assigned only one label (which takes multiple values). We are going to utilize the cluster of machines running Apache Spark, in order to parallelize the work of processing, cleaning the data and extracting useful features from them.

B. TASK OVERVIEW

B1. Data Description

As a dataset for the multi-label classification task, we will use part of the "stackoverflow.com" data dump[1]. This is an xml data set of every question and answer post at stackoverflow.com. Every question in that platform is tagged with a few keywords that we will consider as labels of the text in the title and body of the question.

For our task we have limited the size of the dataset by maintaining only questions (their title and main body of text) that have either of the following tags : 'javascript', 'css', 'jquery', 'html'.

We are given the following data:

- ***train.tsv*** **- (in hdfs : /dssp/datacamp/train.tsv):** Our training data. It contains 4 fields separated by the tab character:
  - *id of the post*
  - *title of the question*
  - *full text of the question*
  - *the set of labels that have been assigned to the question/post. The labels are separated by the comma ',' character.*

[1] https://archive.org/download/stackexchange/

- ***test.tsv*** **- (in hdfs : /dssp/datacamp/test.tsv):** This file is similar in structure to the training data but is missing the labels. The final task of this datacamp is assign labels to the questions existing in this file.

B2. Starting code

You are given examples on various tasks that you may need in order to complete this data camp.

- *example1.py*: Simple examples of loading the text data into a PySpark Dataframe and manipulating the dataframe. Main points of this example:
  - *Initializing a Spark session*
  - *Going form an RDD to a DataFrame (and the opposite)*
  - *User Defined Functions: the equivalent of "map" for dataframes.*
  - *How map a function to an RDD when the function has more than one inputs*
    - *Broadcast variables*
- *example2.py:* It contains a simple example on one of the possible formulation to solve this task (more on that on the next section). We take the code from the previous example and:
  - *Produce TF-IDF features on the title*
  - *Transform the string labels to binary columns*
  - *Train and evaluate a model for ONE label*. We use a simple train/test split on the training dataset to evaluate our models as the test data do not come with ground truth.
  - *Define a metric for the multi-label task*
- *example3.py: This example is similar to the previous one with the only difference that it attempts to produce labels for the test data (test.tsv) instead of providing an evaluation on the training data.*
- *example4.py:* As many of the posts contain HTML tags, we provide in this example a few paradigms that might be useful when dealing with tags. The approach in this example use regular expression (a powerful way to express pattern matching) as they are a basic and quick way to remove/replace specific patterns from text BUT this is not the only way to deal with HTML tags. You are encouraged to utilize other approaches if these examples do not suit you.

B3. Important Notes:

1. **Final Evaluation:** In *example3.py* you will see that we use a special function to evaluate our predictions for the test data. While you are not given the tags for this set, the information is available in general. In order to keep a realistic view on this datacamp, we have removed the tags from the test dataset but we provide you a way to evaluate your final models. In the server machine we have installed a custom python library with one function: the library is call "dssp_evaluation" and in the "tools" module contains a function called "evaluate". This function can evaluate your predictions on the test data.

2. **Different formulations of the multi-label classification task.** As noted above, each instance can be assigned multiple labels. As such, we are trying to predict/identify multiple values. In the examples we provide, we model this task a multiple binary classification tasks i.e. for each label a different classifier. This is only one of the ways to model this problem and we encourage you to explore other ways as well e.g.:
    a. **A multi-class problem where each combination of labels is a unique label.** In this approach the combinations "html,css", "html" and "html,javascript" would be distinct values of ONE label.
    b. **Clustering and Similarity approaches.** These two are similar in their core idea; the main approach requires to define a similarity metric between instances or between instances and labels:
        i. **Variation A:** For each for the "unknown" data find the top-*K* most similar instances from the "known" data and assign the most frequent labels.
        ii. **Variation B:** Cluster the known data into many clusters and maintain their centroids. Assign labels per centroid. Find the closest centroid for each of the unknown instances.
        iii. **Variation C:** Find a similarity metric between labels and posts on the training data and use it to find the "closest" labels for each of the test data.
3. **Final Evaluation Metric:** Regardless of the way you model the task, the final evaluation will be conducted with a variation of the F1-measure for multi-label classification. This variation in a macro-average over all instances: $\boldsymbol{F1 - score = \frac{1}{N}\sum_{1}^{N}\frac{2*|P_i \cap Y_i|}{|P_i|+|Y_i|}}$

    Where:
    a. *N*: the number of instances
    b. P_i : the set of predicted labels for instance *i*
    c. Y_i : the set of the real labels for instance *i*

B4. Tasks to be performed during this data camp

During this data camp, you optimize all steps of the classification task so that you may reach the heist F1-score possible **on the test data.** In short, you have the following is the list of phases that you need to optimize to achieve the optimal score. For more details and ideas you may have a look at the APPENDIX:

1. Data Loading, Cleaning and Pre-processing
2. Feature Extraction and Selection
3. Better Model and Hyper-parameter Optimization

Notice: It is a good practice to optimize your model and general approach on the training data and utilize the test data as the final evaluation. In real world tasks, the labels would not be available for the "unknown" test data. You need to maintain this assumption otherwise you risk over fitting on the test data.

C. RUNNING THE TASK ON THE CLUSTER

Even though the data are not exactly "Big", the procedures involved will require a lot of computations and potentially create intermediate data matrices too big to fit in the main memory of your personal computers. In this data camp, we are going to use PySpark which implements a Python API to the Spark execution engine. By utilizing Spark, the tasks we described above (joining data, cleaning data, etc.) and tasks you will probably require (e.g. computing similarities among fields) will benefit from the distributed computational model of Spark.

You may use spark-submit to send a piece of code to Spark or use the command line interface for quick interactive tests. We cover both approaches in the following.

C1. How to Run PySpark Interactively

Starting a command line interface reserves resources even if we don't execute anything, but it is handy to test unfinished ideas.

```
pyspark --master yarn --num-executors 8 --py-files tosend.zip --driver-memory 2g --conf spark.ui.port=6660
```

When starting a command line interface you have access to the spark context with a predefined variable **sc:**

```
>>> temp=range(10000)                    # local variable : list with 10000 entries

>>> temp_rdd=sc.parallelize(temp)        #variable that refers to the distributed
                                         #version of temp
```

You have also access to *sqlContext* which is useful for data frame related tasks.

```
>>> df=sqlContext.createDataFrame(temp_rdd,["numbers"])
```

C2. How to Use Spark-Submit with Python

If we have a more "complete" code (e.g., code.py) we may send the code to run on the cluster using **spark-submit**:

```
spark-submit --master yarn --num-executors 8 --py-files tosend.zip
--driver-memory 2g --conf spark.ui.port=6660 /path_to_code/code.py
```

When submitting code you have to create a variable that will contain the spark context. Assuming you have already passed the master parameters in the "spark-submit" command:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('example').getOrCreate()
sc = spark.sparkContext
```

Parameter	Exaplanation
--master	Where (url location) to find the master. yarn= look at the local yarn configuration
--num-executors	How many executors to reserve. Each executor reserves a predefined amount of vCPUs and RAM

--py-files	Path to local zip file that contains custom code. This code is available to all executors if we want them to use it. If we don't pass the files in a zip the executors will not have access to the code
--driver-memory	How much local memory (max) should the client use (default is 512mb and usually runs out)
--conf	Additional spark/yarn configurations. E.g the port we want to connect to. Another example : memory per executor/worker : spark.executor.memory. Two pass multiple yarn parameters you have to define --conf multiple times. E.g. ***--conf*** *spark.ui.port=6660* ***--conf*** spark.executor.memory=*2g*

D. EVALUATION OF DATA CAMP

As part of this task you are required to form teams and deliver your final classification results per team. Each team should have at least two persons. Each team will deliver only one solution.

- The performance of your model (F1-score on the test data) as well as the completeness of your solution will determine the evaluation of the members of your team.
- **You will deliver:**
  - The code that produces the feature space and optimizes your regression model
  - A small report on approaches (features, models etc.)

Team formation is necessary to cover the multitude of topics under this data camp and to limit the amount of the required resources for the cluster (the more people working on the task the more resources will be reserved). In order to increase the performance of your model you may choose to apply several techniques. These techniques are already known to you from previous labs. You may freely reuse any code fragment you find useful. In particular, the techniques you may use to increase the accuracy of the results are summarized as follows:

E. APPENDIX

Stemming

The use of stemming is mandatory! Therefore, you should use it in your solution. For example, we may apply it like this:

```
stemmer = PorterStemmer()
doc = [stemmer.stem(w) for w in doc]
```

Feature Selection

If you utilize raw tf-idf features, each term is considered a *feature*. Based on what you have learned in previous lectures, some features may be more important than others. In order to increase the accuracy of the results you may select a limited number of features using the **chi-squared test**. This way, you may decrease the number of dimensions by keeping the most important once, eliminating features that make your data noisy.

Stopword Removal

Some terms cannot contribute in discriminating between documents, simply because they appear in the vast majority of the documents. For example, words like "the", "a", "of", etc can be eliminated from the document collection. Stopword elimination has been proven effective in increasing the accuracy of the results, and you may use it. On way to apply stopword removal is to

parameterize the tf-idf vectorizer. Another way to do it is to apply a "preprocessing" and remove any word that is found in the set of stopwords. Note that the list of stopwords may be user-defined and may contain any word the use thinks that it can be eliminated from the text.

Using *N*-grams

In many cases, n-grams may help decrease the error of your regression task. The default technique is to use each term as a single "gram". An *n*-gram, is generated by applying a sliding window of size *n* over the sequence of terms of the document. For example, assume that we have the document: "this is a simple text document". The unigrams of this document are simply the words in order, i.e., "this", "is", "a", "simple", "text", "document". The bigrams or 2-grams of the document are: "this is", "is a", "a simple", "simple text", "text document". *N*-grams can be computed in the same way, by collecting words that appear together in the document and shifting the sliding window from left to right. Note that if you use *n*-grams, each document is represented as a vector in the n-gram space and not in the term space. Essentially, each *n*-gram is like a "complex term".

Different rClassification Algorithms

In the running example we have shown how to use a simple Logistic regression approach to provide the results. It is up to you to test other algorithms as well and compare them with respect to the accuracy of results. For example, you may use Random Forest or SVM. Moreover, as your final feature space might not be too big (depending on your approaches), you can also collect locally those features and utilize function provided by the popular scikit library.

More complex features

Here we provide you with a list easy tasks which might improve your model.

- **Matrix Factorization**: Try to utilize dimensionality reduction techniques and use as features the first k principal components.
- **Word2Vec** : Word2Vec provides a new vector representation space where every word is a vector. Simple ways to use this is:
    - Compute the average vector of the top frequent terms and use it as a feature or combine it with other fields.
    - Compute similarit between word vector of labels and text (mean/max similarity over all words.)
    -

Note that, different combination of techniques may give very different results. Therefore, it is important to test as many combinations as possible, to maximize the accuracy of the results. Don't be surprised if Linear Regresion with a good feature selection and stopword removal is better than "more sophisticated" techniques without feature selection and stopword removal. Thus, you should spend a significant amount of time in evaluating the classification models you are going to use.

RESOURCES

http://spark.apache.org/docs/latest/api/python/

https://spark.apache.org/docs/latest/programming-guide.html

https://spark.apache.org/examples.html

https://www.youtube.com/watch?v=xc7Lc8RA8wE