DSSP5 Data Camp
July 7-8, 2017
"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. The particular task is focused on predicting the quality results of a query at the site of Home Depot. Past queries from users were evaluated on their quality from the users themselves. Thus, for each query we will have search query, the result product and the evaluation of that query (with values in the range 0-3). 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

This task is taken from a Kaggle competition[1]

We are given the following data:

- *train.csv* : Our training data. It contains the query term, the query relevance (quality) and the title of the product corresponding to the result of that query.
- *product_descriptions.csv*: It contains full text description of the products that are found in train.csv
- *attributes.csv :* It contains additional information about the aforementioned products in the form of product id, name of an attribute, value of that attribute
- You are also given the file *test.csv* which was used in the Kaggle evaluation process (it is similar to *train.csv* but it does not contain the quality of the query result).

[1] https://www.kaggle.com/c/home-depot-product-search-relevance

B2. Starting code

You are given examples on various tasks that you may need in order to complete this datacamp. The main goal is to produce vectors containing features extracted from the text of each query and the corresponding results. The following examples can be found in full in the files:

- *example1.py*: It contains simple feature extraction and regression examples
- *example2.py :* It contains a simple example on cleaning the data
- *check.py :* Contains helpful functions to clean the data

Loading the data into Spark

As some of the methods you might deploy might be computationally expensive, it is best to utilize Spark for distributed computations. The first step involves loading the data into RDDs. The following python code shows how to load a csv file as a Pyspark **DataFrame**. This data frame is very similar to the pandas data frame which you may have encountered in the past.

```
data=sqlContext.read.format("com.databricks.spark.csv").\
      option("header", "true").\
      option("inferSchema", "true").\
      load("/dssp/datacamp/train.csv").repartition(100)
```

Joining Data

In order to utilize the full potential of our data it would be best if we joined the information from the various data sources. It is a good idea to use dataframes for this task, since they provide very efficient methods for joining (in comparison with RDDs). Assume the previous dataframe and a dataframe that contains the attributes of each product aggregated by product id.

```
fulldata=data.join(atrDF,['product_uid'],'left_outer')
#atrDF is the data frame with the attributes per product id
```

As the data that holds the attributes might have more than one lines corresponding to a specific product id, it would be best to aggregate them first. Dataframes offer the functionality of user defined aggregation functions that can be applied to them. In this scenario, the user registers custom functions that will take as an input collected data and aggregate them together. In order to display how that happens and to show how we can transform a dataframe into an RDD and back again, we provide an illustration of the aggregation with the utilization of RDD methods.

```
#assume attributes is the dataframe
#fixEncoding returns the same data only the fields value and
name as concatenated together
#the data now are : (id,[attribute information])
attRDD=attributes.rdd.map(fixEncoding)
#group data by their key (id) and apply the lambda as the
aggregation
attAG=attRDD.reduceByKey(lambda x,y:x+y).\
map(lambda x:(x[0],' '.join(x[1])))
```

```
#convert the RDD back to a dataframe
atrDF=sqlContext.createDataFrame(attAG,
["product_uid", "attributes"])
```

You can see in more detail how we perform the above in the file *example1.py*.

Creating vector features with TF-IDF

Spark offers an automatic manner to compute the TF-IDF vectors of text data so we don't need to implement them. Here we use the title field of our data as the target; you may apply the same procedure in the other fields as well.

```
#Step 1: split text field into words and put results in to new field
tokenizer = Tokenizer(inputCol="product_title",
outputCol="words_title")
fulldata = tokenizer.transform(fulldata)

#Step 2: compute term frequencies (TF). Put the result in field "tf"
hashingTF = HashingTF(inputCol="words_title", outputCol="tf")
fulldata = hashingTF.transform(fulldata)
#Step 3: compute IDF. Put the result in field "tf-idf"
idf = IDF(inputCol="tf", outputCol="tf_idf")
idfModel = idf.fit(fulldata)
fulldata = idfModel.transform(fulldata)
```

Creating additional features

As you may realize, you will need to create additional features and combine fields in order to improve your model. For this reason we provide you with two examples on how to:

1. Add features to the existing TF-IDF ones.
2. Create a new field with new features.

Add features to the existing TF-IDF ones.

```
#new function to be applied to the RDD of the data frame
def addFeatureLen(row):
      vector=row['tf_idf']#get the Vector with the tf-idf values
      size=vector.size
      newVector={}#create new vector that we are going to enrich
      for i,v in enumerate(vector.indices):
            newVector[v]=vector.values[i]
      newVector[size]=len(vector.indices) #add another field
      size+=1
      #we cannot change the input Row so we need to create a new one
      data=row.asDict()
      data['tf_idf']= SparseVector(size,newVector)
      #new Row object with specified NEW fields
      newRow=Row(*data.keys())
      #fill in the values for the fields
      newRow=newRow(*data.values())
      return newRow
```

```
#apply the function to the RDD of the dataframe and create a new one
fulldata=sqlContext.createDataFrame(fulldata.rdd.map(addFeatureLen))
```

Create a new field with new features.

```
#Function to create a new field with features.
def newFeatures(row):
      vector=row['tf_idf']
      data=row.asDict()
       data['features']=DenseVector([len(vector.indices),#new field
      vector.values.min()])
      newRow=Row(*data.keys())
      newRow=newRow(*data.values())
      return newRow
```

A difference that you may notice between the two examples is that one use a sparse vector while the other a dense one. Sparse vectors do not store zero values and are thusly ideal for TF-IDF values. The sparse vector stores inly its' size and a dictionary of indexes and their corresponding values (for the indexes that have non-zero values).

Cleaning up the data

As the text might contain human errors we provide you with a simple set of functions that can be used to determine the likelihood of a word based on the frequency in the data.

The file *check.py* implements a simple dictionary building method based on a single column of the data. We call the function "*train*" at *example2.py* in order to get this dictionary and try to fix possible errors in the query words.

You are encouraged to edit that code in order to build a better dictionary (perhaps use a different field to train it).

Regression and MSE

As the general task of this data camp is a regression one, we provide you with a simple example on how to use a regression model from the Spark machine learning library. You are not limited to this model and you can explore other provide models.

```
#rename relevance column and select only features and label
fulldata=fulldata.withColumnRenamed('relevance',
'label').select(['label','features'])

#Simple evaluation : train and test split
(train,test)=fulldata.rdd.randomSplit([0.8,0.2])

#Initialize regression model
lr = LinearRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8)

# Fit the model
lrModel = lr.fit(sqlContext.createDataFrame(train))
```

```
#Apply model to test data
result=lrModel.transform(sqlContext.createDataFrame(test))
#Compute mean squared error metric
MSE = result.rdd.map(lambda r: (r['label'] -
r['prediction'])**2).mean()
```

Notice that the model is applied directly on a data frame and that it expects fields named *"features" and "label".*

As an evaluation metric here we assume the mean squared error between the predicted values and the original ones. **The end goal of this data camp is to compute features and utilize the best model which will provide the lowest error.**

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 to your personal laptops. In this camp, we are going to use PySpark which implemented a Python API to the Spark execution engine. By utilizing Spark, the tasks we described above (joining data, cleaning them) 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 --confspark.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 import SparkContext
sc = SparkContext(appName="Simple App")
```

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 regression results per team. Each team should have two or three persons. Each team will deliver only one solution.

- The performance of your model (MSE) 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 the approaches you have followed and tested (e.g., features, models, algorithms etc.)
  - *Optional*: You may use the test.csv file to see how well you would have done in the original Kaggle competition.

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:

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. A similar technique has been followed in the text mining lab of June 26.

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 regression algorithms

In the running example we have shown how to use a simple Linear regression approachto 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.

- **Similarity Feature** : Compute the similarity between the query and other fields

- **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. A simple way to use this is to compute the average vector of the top frequent terms and use it as a feature or combine it with other fields.
- **Improve the dictionary:** Apply the provided example to different fields to get a better dictionary for data cleaning

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 Regression 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. Remember that the main goal of the datacamp is to learn how to apply your ideas in a distributed environment and how to try different techniques that solve the same problem. Do not hesitate to try out your ideas.

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