COMS W4121: Compute Systems for Data Science

Assignment #4: MLlib, Spark Streaming, and SparkSQL

Due April 20th, 2015 at 11:55PM

Update April 15th:

- Changed part 3 to be an extra credit part
- An additional output in part 2
- Extended deadline

Overview

In this assignment, you will build a simple topic-based document search engine using the Machine Learning Library of Apache Spark. The tasks include computing the **term frequency–inverse document frequency (tf-idf)**, cluster the pages based on their TFIDF scores using K-Means clustering algorithm and find topic words of each cluster, stream the input file using Spark Streaming, and finally query your database using SparkSQL.

The dataset you will be dealing with is a set of pre-processed Wikipedia pages. Read more about the dataset in the "Dataset" section.

Dataset

Pre-processed English Wikipedia Page:

The dataset consists of a single file. Each line in the file represents a English Wikipedia page. The contents of the pages are tokenized such that each word is separated by a single space. A word contains only letters and hyphens. You **do not have to do any data cleaning on this dataset**. The format of a line in the dataset is as follows:

1. Text before the first space is the ID of the page
2. Text after the first space and before the first tab ('\t') is the title of the page.
3. Text after the first tab is the content of the page.
4. All words in the content are deliminated by a single space (' ')

```
id1 title1    word1 word2 word3 ...
id2 title2    word1 word2 word3 ...
id3 title3    word1 word2 word3 ...
...
```

Spark Machine Learning Library

MLlib is Spark's scalable machine learning library consisting of common learning algorithms and utilities, including classification, regression, clustering, collaborative filtering, dimensionality reduction, as well as underlying optimization primitives.

We will be using MLlib's K-means and TFIDF modules.

Machine Learning Library Here

Spark Programming Guide: Here

Spark Python API: Here

1. Compute TFIDF using MLlib (10 points)

Please refer to HW #3 for a tutorial on TFIDF

Here is a piece of sample code on how to use MLlib's TFIDF module. I encourage you to run this piece of code in a PySpark or on the DataBrick cluster. Carefully inspect each step's output and the structure of the output.

TF and IDF are implemented in HashingTF and IDF. HashingTF takes an RDD of list as the input. Each record could be an iterable of strings or other types.

```
from pyspark.mllib.feature import IDF
from pyspark.mllib.feature import HashingTF

#create some fake data
doc1 = ['word1', 'word1', 'word4', 'word2'] #document 1
doc2 = ['word1', 'word1', 'word3', 'word2'] #document 2
docs_rdd = sc.parallelize([doc1, doc2]) #transform to an RDD
```

Build a HasingTF model and compute the TF scores of the **entire corpus**

```
htf = HashingTF()
tf = htf.transform(docs_rdd)
```

While applying HashingTF only needs a single pass to the data, applying IDF needs two passes: first to compute the IDF vector and second to scale the term frequencies by IDF.

```
idf = IDF().fit(tf) #compute the IDF vector
tfidf = idf.transform(tf) #scale the term frequencies by IDF

print tfidf.collect()

#find the index of a word on the resulting feature vector
print htf.indexOf('word1')
```

Task:

Write a Spark program that uses MLlib's TFIDF module to compute TF-IDF of a set of pre-processed Wikipedia pages. For each page, output the word that has the highest TFIDF value and its correspounding page title and ID.

Notice that due to the nature of HashingTF, words might be mapped to the same index in the feature vector. So there can be 0 or more words correspound to the index of the highest TFIDF score. In this case, just output all words that are mapped to the index that has the highest TFIDF score. In the case that no word is mapped to the highest TFIDF score, find the next index that has the highest TFIDF score and has word(s) mapped to it.

Implementation details:

Input: `hdfs:/datasets/hw4/data.txt` Please refer to the "Dataset" section for more information about the format of this dataset.

Output: Word(s) that has the highest TF-IDF of each page and its tf-idf value and the correspounding page title and id.

What to submit:

1. A Spark program (**p1_tf_idf.py**) that computes the TF-IDF of a input document/directory. The program should take the first command line argument as the path of the input file/directory and the second argument as the path of the output directory.
2. A file (**p1_output.txt**) that contains the output. The output should be in the following format:

```
document_id document_title word tf_idf
document_id document_title word tf_idf
document_id document_title word tf_idf
...
```

2. K-means Clustering using TFIDF scores (60 points)

Clustering is an unsupervised learning problem whereby we aim to group subsets of entities with one another based on some notion of similarity. Clustering is often used for exploratory analysis and/or as a component of a hierarchical supervised learning pipeline (in which distinct classifiers or regression models are trained for each cluster).

MLlib supports k-means clustering, one of the most commonly used clustering algorithms that clusters the data points into predefined number of clusters. The MLlib implementation includes a parallelized variant of the k-means++ method called kmeansII. The implementation in MLlib has the following parameters:

- k is the number of desired clusters.
- maxIterations is the maximum number of iterations to run.
- initializationMode specifies either random initialization or initialization via k-meansII.
- runs is the number of times to run the k-means algorithm (k-means is not guaranteed to find a globally optimal solution, and when run multiple times on a given dataset, the algorithm returns the best clustering result).
- initializationSteps determines the number of steps in the k-meansII algorithm.
- epsilon determines the distance threshold within which we consider k-means to have converged.

The following code snippets can be executed in spark-shell.

In the following example after loading and parsing data, we use the KMeans object to cluster the data into two clusters. The number of desired clusters is passed to the algorithm.

```
data = [(1, 1.1), (1, 1.2), (1, 1.3), (2, 1.1), (2, 1.2), (2, 1.3)]
data_rdd = sc.parallelize(data)

clusters = KMeans.train(data_rdd, k=2, maxIterations=10, runs=1, initializationMode="
random") #run K-means clustering

print clusters.centers #centers of each cluster
```

Notice that KMeans can operate on both an RDD of list and an RDD of MLlib's Sparse Vector, which is the data structure of the output of the TFIDF module.

Tasks:

In this part, you will cluster the pages based on their TFIDF scores, find which page belongs to which cluster, and get the topic words of each cluster.

Step 1

Use MLlib's K-Means module combined with your code from Part 1 to perform K-means clustering on a set of pre-processed Wikipedia pages. This will accelerate the computation in step 2 and step 3.

Use `k=50, maxIterations=10, runs=2, initializationMode="random"` for KMeans module.

Step 2

Find the **indices** of the **topic words** of each cluster.

The topic words of a cluster is defined as the words that have the top **100** TFIDF values in the **center vector** of the cluster. In other words, sort the center vector in descending order, and get the indices of the first 100 terms. Notice that you don't have to actually find the words that correspound to these indice until the very end of this part.

Again, due to the nature of HashingTF, words might be mapped to the same index on a feature vector, which means that there can be 0 or more words correspounds to an index of the center feature vector. Hence the number of topic words of each cluster is not necessarily 100.

Step 3

Find which page belongs to which cluster.

Due to the high dimensionality of our feature vector, using K-Mean's predict() function would take more than two hours to go through each page (**So don't use it!**). Instead, we will use the topic words from step #2 to find which page belongs to which cluster.

To do so, for each page and each cluster, count the number of times that each topic word of a cluster appears in a page. A page is considered to belong to a cluster if the sum of the number of times that the topic words of the cluster appear in the page is greater or equal to some threshold value `t`. Notice that a page can belong to more than one cluster.

For example, say we have cluster c1, c2, and page p1, p2

c1's topic words are: `'word1', 'word2', 'word3'`

c2's topic words are: `'word2', 'word5', 'word6'`

p1 contains: `'word1', 'word1', 'word5'`

p2 contains: `'word5', 'word6', 'word1', 'word3`

Then p1 has count=2 for c1 and count=1 for c2, p2 has count=2 for c1 and count=2 for c2.

If the threshold t=2, then p1 belongs to c1. p2 belongs to both c1 and c2.

Step 4

Write out a list of (topic word, page title) mappings from the clusters that **have less than 50 but more than 1 pages**. You **don't have to submit this output file**. You will be reading from this file in Part 3. The suggested format of this output file is:

```
topic1 title1
topic2 title1
topic3 title2
...
```

, where the part before the first space is the word and everything after the first space is the title.

Implementation details:

Input: `hdfs:/datasets/hw4/data.txt` Please refer to the "Dataset" section for more information about the format of this dataset.

Use threshold value t = 30

Output:

1. For each cluster that has **less than 50 pages and more than 1 page**, write out the **5** topic words that have the hightest TFIDF value and the titles of all pages that belong to the cluster.
2. (Don't submit this) Write out a list of (topic word, page title) mappings from the clusters that **have less than 50 but more than 1 pages**. You **don't have to submit this output file**. You will be reading from this file in Part 3.

IMPORTANT: Unfortunately there is not a way of designating a random seed for the Kmeans module in the current release of Spark, which means that each run of your code will potentially be different, but not by very much. So I won't grade the output based on how well it matches the output of the solution, but based on if the it seems to be a reasonable clustering.

What to submit:

1. A Spark program (**p2_kmeans.py**). The program should take the first command line argument as the path of the input file/directory, second argument as the path of the first output file, third argument as the path of the second output file, and the fourth argument as the threshold value.
2. A file (**p2_output.txt**) that contains the output. The output should be in the following format (roughly):

```
[topic1, topic2, topic3, topic4, topic5], [title1, title2, ...]
[topic1, topic2, topic3, topic4, topic5], [title1, title2, ...]
...
```

---

3. SparkSQL & Streaming (30 points)

In this part, you will transform the result from part 3 to a DataFrame table, and query the database using SparkSQL SparkSQL Tutorial. The query keywords (user input) come from SparkStreaming.

Important: When submitting the Spark job for part 3, set the master to be `local[2]`, i.e.,

```
spark-submit --master local[2] p3_sql.py
```

or when initalizing SparkContext in your script, do

```
sc = SparkContext("local[2]", "p3_sql")
```

Otherwise the program will not able to receive streaming from a local shell.

Tasks:

1. Read the output from part2 as an RDD, and transform the RDD into a DataFrame.
2. When creating the DataFrame, name the field that correspounds to the topic word as "topic", and the field that correspounds to the page title as "title".
3. Register the DataFrame as a table.
4. Start a StreamingContext with batch interval = 3 second.
5. Run Spark SQL on the table from step 3 to find all documents of the particular topic indicated by the word obtained from the Streaming. Print out the result in the console.
6. Test it with some topic words and see if the output is reasonable. I suggest testing the topic words from the output of part 2.

What to submit:

1. A Spark program (**p3_sql.py**). The program should take the first command line argument as the path of the input file/directory and the second argument as the port number.

---

4. Extra Credit (30 points)

4.1. Do Part 1 and Part 2 in SparkR (20 points)

To start SparkR, run R shell and type `library(SparkR)`

Tutorial: http://amplab-extras.github.io/SparkR-pkg/

4.2. Stream the dataset (10 points)

In this part, you will change the input of your program from reading from HDFS to reading from a text stream.

Refer to this for a what is Spark Streaming and a simple Spark Streaming example. I strongly suggest that you run through the word count example.

Tasks:

1. Copy your code from part 2 and name it streaming.py. Create a StreamingContext, and set the batch size to be **15** seconds.
2. Set the input port number to be a very special number that is >= 10000 such that you can avoid port conflicts with other users and/or program on the cluster. I suggest using 1+youruni. For example, if your uni is ab1234, the port to use is 11234.
3. Use the RDD obtained from StreamingContext as the dataset of your clustering algoithm.
4. Start the streaming
5. We will simulate a text stream by sending the dataset via netcat in a local shell. To stream the data, open up another terminal and SSH on to the cluster, do `cat data.txt | nc -lk your_port_number`
6. For each cluster that has **less than 50 pages and more than 1 page**, **print out** the **5** topic words that have the hightest TFIDF value and the titles of all pages that belong to the cluster.

Use threshold value t = 30

What to submit:

1. A Spark program (**streaming.py**). The program should take the first command line argument as the port number, second argument as the path of the output directory, and the third argument as the threshold value.