Assignment 7

Server Log Correlation

For this question, we will be imagining a situation where we have web server logs to analyze. Dealing with server logs is a fairly realistic task, but publicly available HTTP logs are hard to find because of privacy concerns. There are some old NASA web server logs available that are big enough to experiment with.

The data we will use is `nasa-logs-1` and `nasa-logs-2`. Neither is large, but we can get an idea of what dealing with this kind of data is like.

I have a theory: I think that in our web server logs, the number of bytes transferred to a host might be correlated with the number of requests the host makes.

In order to check this theory, I would like to calculate the correlation coefficient of **each host**'s number of requests and total bytes transferred. That is, each host making requests will contribute a data point (x, y) where x is the number of requests made by that host, and y is the total number of bytes transferred. Then the correlation coefficient can be calculated as,

$$r = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{\sqrt{n\sum x_i^2 - (\sum x_i)^2}\sqrt{n\sum y_i^2 - (\sum y_i)^2}}.$$

The formula looks bad, but note that there are really only six sums we need: $n = \sum 1$ (the number of data points), $\sum x_i$, $\sum y_i$, $\sum x_i^2$, $\sum y_i^2$, $\sum x_i y_i$.

That implies an algorithm like:

1. Get the data out of the files into a DataFrame where you have the hostname and number of bytes for each request. Do this using an RDD operation: see hints below.
2. Group by hostname; get the number of requests and sum of bytes transferred, to form a data point $(count\_requests, sum\_request\_bytes) = (x_i, y_i)$.
3. Produce six values: $\mathbf{1}, x_i, x_i^2, y_i, y_i^2, x_i y_i$. Add these to get the six sums.
4. Calculate the final value of r.

Write a Spark program `correlate_logs.py` that loads the logs from the input files, parses out useful info, and calculates the correlation coefficient of each requesting host's request count and total bytes transferred. Take arguments for the input files. Our output here is really just the value for r. We also care about r^2 because it's meaningful. For our output, we will simply **print** the values of r and r^2 like this (for the `nasa-logs-2` data set):

```
r = 0.928466
r^2 = 0.862048
```

You may also want to output the six sums for debugging (and feel free to leave them in your final version).

Your program should take only the input directory on the command line. The output will simply be printed.

Implementation

This is a case where the input is unstructured and it's easiest to start with an RDD and do some Python work. We can use a regular expression (with the Python re module) to disassemble the log lines.

```
import re
line_re = re.compile(r'^(\S+) - - \[(\S+) [+-]\d+\] \"[A-Z]+ (\S+) HTTP/\d\.\d\" \d+ (\d+)$')
```

The four captured groups are the host name, the datetime, the requested path, and the number of bytes transferred. There are a few lines that don't match this regex, and you'll have to ignore them.

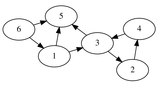
As soon as possible, convert an RDD of `Row` objects to a DataFrame and continue the calculation.

After you get the pairs of values (count and bytes for each host, or x and y as we'll start calling them), create a DataFrames of the six values you need to sum. Sum those tuples, collect, and calculate the correlation coefficient with the formula above.

There is a built-in function `corr` that you can **not** use (but you could use it to check your result).

Shortest Paths in Graphs

Let's have a look at the problem of finding the shortest path in a graph. To start with, this graph:



We will represent the graph by listing the outgoing edges of each node (and all of our nodes will be labelled with integers):

```
1: 3 5
2: 4
3: 2 5
4: 3
5:
6: 1 5
```

If we would are asked to find the shortest path from 1 to 4, we should return 1, 3, 2, 4.

You can do this problem with Spark DataFrames, **or** RDDs. (I will use a slightly more verbose description of the data below, to not bias toward one or the other.)

The Algorithm

We will use a modification of Dijkstra's algorithm that can be parallelized.

We will maintain an RDD/DataFrame containing the list of nodes we know how to reach with the shortest path, along with the node we reach them from (to reconstruct the path later) and their distance from the origin.

At the **end** of the algorithm, we hope to have found this:

```
node 1: source -, distance 0
node 3: source 1, distance 1
node 5: source 1, distance 1
node 2: source 3, distance 2
node 4: source 2, distance 3
```

That is, node 4 can be reached from node 2, and is 3 steps from the origin (1). We can start the algorithm with one entry we know: the origin is 0 steps away from itself: (Represent the "no source" however you like: we won't be using the value.)

```
node 1: source -, distance 0
```

After that, we can iterate: nodes in that list at distance n imply a path to their neighbours at distance $n + 1$. In this case, we have nodes of distance 0 (just node 1) and can now find all of their neighbours (3 and 5), so after one iteration, have:

```
node 1: source -, distance 0
node 3: source 1, distance 1
node 5: source 1, distance 1
```

Now we can look at the nodes at distance 1 and their outgoing edges: those can be reached at distance 2. After another iteration:

```
node 1: source -, distance 0
node 3: source 1, distance 1
node 5: source 1, distance 1
node 2: source 3, distance 2
node 5: source 3, distance 2
```

... and we see a problem. We already knew how to reach node 5 with a path of length 1: the newly-found path of length 2 doesn't help. We need to **keep only the shortest-length path** to each node:

```
node 1: source -, distance 0
node 3: source 1, distance 1
node 5: source 1, distance 1
node 2: source 3, distance 2
```

With one more iteration:

```
node 1: source -, distance 0
node 3: source 1, distance 1
node 5: source 1, distance 1
node 2: source 3, distance 2
node 4: source 2, distance 3
```

We can now use this to trace back the path used to get to 4: the last line says that we got to 4 from 2 at distance 3. We got to 2 from 3; and 3 from 1. That gives the correct path 1, 3, 2, 4.

Implementation

Your program should be `shortest_path.py` and the command line should take four arguments: the input directory, an output path where we will produce a few things, and the source and destination nodes (both integers). That is, the command line will end up like:

```
spark-submit shortest_path.py graph-1 output 1 4
```

The input path will contain a file `links-simple-sorted.txt`, and you should consider **only** that file in the directory. The input file is in the format above. Each line of input is a node, colon, and a space-separated list of nodes it's adjacent to.

For each iteration of the algorithm, output the RDD or DataFrame of the found paths (in some sensible format) in a directory `iter-<N>`: you can use this to debug or verify your algorithm. When you are done, produce an **output directory path containing the actual path** you found, with one node per line:

```
1
3
2
4
```

It would be very easy to create an infinite loop while implementing this algorithm. Please implement like this, so there is an upper-bound on the number of iterations of the algorithm (assuming RDD data here; adapt appropriately for DataFrames):

```
for i in range(6):
    ...
    paths.saveAsTextFile(output + "/iter-" + str(i))
    if we_seem_to_be_done:
        break

finalpath = ...
finalpath.saveAsTextFile(output + "/path")
```

[This implies we can't find paths of length >6, but we will live with that restriction.]

Sample Data

You will find `graph-1` in the usual locations, which is the 6-node graph from the examples above.

The `graph-2` data set is a randomly generated graph with 100 nodes and 200 edges (with a path from 53 to 91 and 92 to 19, but none of length ≤6 from 12 to 45). The `graph-3` data set is randomly generated with 1M nodes and 8M edges (with paths from 694568 to 4565 and 355931 to 381399; none from 5463 to 88512).

On the cluster, there is also a `graph-4` data set which is not random: it's data from Wikipedia. A description of the original analysis includes links to the data. The data format for this question was borrowed from that data set.

Hints

Pay attention to what you cache here: not caching data you need to iterate on will turn a perfectly reasonable algorithm into something completely insane.

Overall efficiency in your loop is going to matter as the input graph grows: don't create candidate paths blindly and then reduce/filter them out if you can avoid it.

There are certainly many ways to proceed with the problem, but I have done these...

Hints: Spark + RDDs

You should create a key-value RDD of the **graph edges**: the node as your key, and list of nodes connected by outgoing edges as the value. Cache this, because you'll be using it a lot.

You'll use it by joining it to your working list of **known paths** in a format something like `(node, (source, distance))`. Then for each node, produce a new RDD with all known paths (probably with `flatMap` and a function that will...). Produce all currently-known paths, as well as any new ones found by going one more step in the graph.

That will produce the longer-than-necessary paths: reduce by key to keep only the shortest paths.

Hints: Spark SQL + DataFrames

Create a DataFrame of the **graph edges**: source and destination nodes for each edge. Cache this, because you'll be using it a lot.

Create a DataFrame to represent **known paths** (node, source, distance), starting with only the origin node (at distance 0 from itself). With each step, you will need to create a DataFrame in the same format with one-step-longer paths and use `.unionAll` to add it to the collection of paths you have.

That will produce the longer-than-necessary paths: subtract/filter/groupby to keep only the shortest.

Hints: path reconstruction

Once you have the destination node and its "source", you can start working backwards. You can do a `.lookup` on an RDD or a `.where/.filter` on a DataFrame to find the source and *its* source. Iterate until you get back to the original origin.

Since we know we will be finding paths with length at most 6, you can build the path in a Python list (and `.parallelize` to write).

Questions

For once, I have no questions to ask here.

Submission

Submit your files to the CourSys activity Assignment 7.

Updated Wed Oct. 26 2022, 10:36 by ggbaker.