CIS 545: Big Data Analytics

Spring 2018

Homework 2: Big Data, Graph Data

Due February 22, 2018 by 10pm

For this assignment, we will focus on *graph* data. You saw an instance of this with Homework 1 -- the airline flight network is actually a graph -- but we only did limited kinds of computation over the graph. However, many real-world datasets are, or can be modeled by, graphs (or trees which are special cases of graphs). Examples include:

- Networks (social networks, the Web, the connectome, the Internet, traffic networks, …)
- Sets of data in which some of the data is more closely connected than other parts of the data (edges may represent weighted similarity or affinity)
- Phylogenetic trees, grammars, etc.

For this assignment, we will be doing a few common operations on graphs. In the next assignment, when we have the power of matrices, we will do some further computation over the same graph data. (It's very common to encode graph connectivity through an *adjacency matrix* that we'll discuss in lecture.)

Step 1. Clone the Repo and Download the Datasets

Let's start by downloading the data files for this assignment. These will include flight data from HW1 as well as social network data (specifically, data about who answered questions from whom on Stack Overflow).

From Bitbucket, clone the HW2 repository:

```
git clone https://bitbucket.org/pennbigdataanalytics/hw2.git
```

As with HW1, you now need to figure out how to make the files visible to Jupyter within Docker.

- You are all set if you've cloned this into a **shared directory** between your main operating system's `Jupyter` folder and your Docker container.
- Otherwise, you can use `docker ps` to get the Docker all-spark container ID and `docker cp` to copy all of the data into the container, as with HW1.
- Or finally, you can go to Jupyter and Upload each of the files from **hw2**.

Now in Jupyter, go into **`hw2`** and open **`Retrieve.ipynb`**. Go to **Cell|Run All**. This will take a while, but will download several data files and, as necessary, decompress them. They also give you a model for how you can retrieve text or gzipped files from the Web.

Step 2. Get Started with Apache Spark (within Docker)

Apache Spark, which has become the de facto successor to Apache Hadoop, is a complex, cluster-based data processing system that was written in Scala. It leverages a wide variety of distributed tools and components used for big data processing. It interfaces "smoothly" to Python, but be forewarned that there are some rough edges. For those interested in why, there are a few reasons:

1. Scala has slightly different notions of types (especially things like Rows) and handles missing values (nulls) differently from Python.
2. The Scala-based Spark "engine" can't just run Python functions as it's doing da

you want to be careful to use Spark's library of functions, or the special mechanisms for inserting "user defined functions."
3. DataFrames on Spark are "sharded," so there is no single object corresponding to the DataFrame!

While Spark DataFrames try to emulate the same programming style as Pandas DataFrames, there are some differences in how you express things. Please refer to the Lecture Slides for our take on the differences. You may also find the following Web pages to be useful resources for understanding Spark vs Pandas DataFrames:

- https://lab.getbase.com/pandarize-spark-dataframes/
- https://ogirardot.wordpress.com/2015/07/31/from-pandas-to-apache-sparks-dataframe/

For this assignment, we are going to get familiar with Spark *without* worrying too much about sharding and distribution. We are going to run Spark on your Docker container. **This isn't really using it to its strengths** -- and in fact you might find Spark to be unexpectedly slow -- but it will get you comfortable with programming in Spark without worrying about distributed nodes, clusters, and spending real dollars on the cloud. Your code, if written properly, will "naturally scale" to clusters running on the Cloud. Later in the term we'll connect your Jupyter instance to Spark running on the cloud -- to handle "truly big data."

Step 2.1 Initializing a Connection to Spark

First, open the Jupyter notebook called "`Graphs`". Then fill in the appropriate Cell in the notebook ("# TODO: Connect to Spark as per Step 2.1") and establish a connection to the Spark engine as follows.

```
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F

spark = SparkSession.builder.appName('Graphs-HW2').getOrCreate()
```

Note: Spark has multiple interfaces -- SparkSession is the "most modern" one and we'll be using it for this course. From SparkSession, you can load data into DataFrames as well as RDDs.

Step 2.2 Load Our Graph Datasets.

For this assignment, we'll be looking at **graph data** downloaded from the Stack Overflow site. Many of you have probably ended up at Stack Overflow when Googling for answers about how to solve a programming task -- possibly even a programming task related to this very course! Stack Overflow has a variety of users who post questions, answers to questions, and comments on both the questions and the answers. Network science researchers at Stanford have extracted the network structure of these answers into a dataset for study.

A very brief review of graph theory. *Recall that a* ***graph G*** *is composed of a set of* ***vertices V*** *(also called nodes) and* ***edges E*** *(sometimes called links). Each vertex* $v \in V$ *has an* ***identity*** *(often represented in the real world as a string or numeric "node ID"). Each edge* $e \in E$ *is a tuple* (v_i, v_j) *where* v_i *represents the* ***source*** *or* ***origin*** *of the edge, and* v_j *represents the* ***target*** *or* ***destination****. In the simplest case, the edge tuple above is simply the pair* (v_i, v_j) *but in many cases we may have additional fields such as a label or a distance. Recall also that graphs may be* ***undirected*** *or* ***directed****; in undirected graphs, all edges are symmetric whereas in directed graphs, they are not. For instance, airline flights are directed, whereas Facebook friend relationships are undirected.*

Let's read the social graph data of Stack Overflow, which forms a directed graph. Here, the set of nodes is also not specified; the assumption is that the only nodes that matter are linked to other nodes, and thus their IDs will appear in the set of edges. To load the file into a Spark DataFrame, you can use the following lines.

```
# Read lines from the text file
answers_sdf = spark.read.load('sx-stackoverflow-a2q.txt', format="text")
```

We'll use the suffix `_sdf` to represent "Spark DataFrame," much as we used "`_df`" to denc

Homework 1. Repeat the load process to create Spark DataFrames for the other files, `sx-stackoverflow-c2a.txt`, which should go into `comments_answers_sdf`; and finally, `sx-stackoverflow-c2q.txt`, which should go into `comments_questions_sdf`.

Data Check for Step 2.2. Let's execute some Cells to get a quick look. You should already have the following Cells ready to do this, and should be able to step through them.

1. A Cell in Markdown with contents "Step 2.2 Results"
2. Consecutive Cells that run **answers_sdf.count()** and then **answers_sdf.show(10)**.
3. A Cell that runs **answers_sdf.printSchema().**
4. Two Cells that run **comments_answers_sdf.count()** and **comments_answers_sdf.show(10)**, respectively.
5. A Cell that runs **comments_answers_sdf.printSchema().**
6. Two Cells that run **comments_questions_sdf.count()** and **comments_questions_sdf.show(10)**, respectively.
7. A Cell that runs **comments_questions_sdf.printSchema().**

Step 2.3 Simple Wrangling in Spark DataFrames

Currently, the network data from the three source files is not very detailed; it consists of a string-valued attribute called **value** with three space-delimited columns (representing a social network *edge* and timestamp). We want to separate out each of these three columns into a Spark DataFrame column, and additionally we want to add a fourth field describing which type of edge is being represented.

Let's quickly enumerate some of the building blocks. (Look at the lecture slides for this part of the homework for more help.)

1. The **split** function in Spark, works similarly to the one in Python. It can be called directly from Spark SQL ("`select split(x,' ')` …") or by importing `pyspark.sql.functions` and referring to the function in Python.
2. You may need to **cast** your columns since they start off as strings. In Python, you can call `my_sdf.column.cast('type')` to convert data types. In SQL it's '`SELECT CAST(my_sdf.column AS type)`.'

To split, you can use a query such as:

```
SELECT split(col, ' ')[0] as column1, split(col, ' ')[1] as column2
```

And to cast, you can use a query like:

```
SELECT CAST(MyVarcharCol AS INT) FROM Table
```

Combining these, we get a command like (although note that it is incomplete and you need to finish it):

```
my_sdf.createOrReplaceTempView('my_sdf_view')
spark.sql('SELECT CAST(split(column, pattern)[0] AS double) AS my_col FROM my_sdf_view')
```

So, for instance, to convert a column to a **double**, you can do:

```
my_sdf.select(F.split(my_sdf.column, pattern)[0].alias("my_col").cast("double"))
```

Or

```
my_sdf.createOrReplaceTempView('my_sdf_view')
spark.sql('SELECT CAST(split(column, pattern)[0] AS double) AS my_col FROM my_sdf_view')
```

`createOrReplaceTempView` creates a temporary view (removed when the SparkSession ends) that allows us to run SQL queries against the contents of `my_sdf`; `my_sdf` isn't visible to Spark's SQL engine by itself.

Finally, you may need to add columns with literal values. You can do this by composing functions:

```
my_sdf.withColumn('new_col', F.lit('value'))
```

Which creates a column with a new literal value. Or you can use SQL:

```
spark.sql('select *, "value" AS new_col FROM my_sdf_view')
```

For this step, we want to regularize and combine our graph data from the three Spark DataFrames.

1. Convert each Spark DataFrame into a 4-ary (4-column) table with the columns
   Directly replace the original Spark DataFrame with the corresponding new one (

a single-column to a 4-column table).
- **from_node** and **to_node** should be **integers**.
- **edge_type** should be "`answer`" for **answers_sdf**, "`comment-on-question`" for **comments_questions_sdf**, and "`comment-on-answer`"for **comments_answers_sdf**.

2. Now use the Spark DataFrame **unionAll** function, or **spark.sql()** and the SQL `UNION ALL` operator, to create a single Spark DataFrame called **graph_sdf**, concatenating the results of all of the above, in the order **answers_sdf, comments_questions_sdf, comments_answers_sdf**. (If you use SQL, remember you need to call **createOrReplaceTempView** on the **_sdf** DataFrames to make them "visible" to SparkSQL.)

Data Check for Step 2.3. Let's now execute the output Cells that should be given to you:
1. A Cell in Markdown with contents "## Step 2.3 Results" and hit **Shift-Enter**
2. Two consecutive Cells that run (for the new versions of your Spark DataFrames) **answers_sdf.count()** and then **answers_sdf.show(5)**.
3. A Cell that runs **answers_sdf.printSchema().**
4. Two consecutive Cells that run **comments_answers_sdf.count()** and **comments_answers_sdf.show(5)**.
5. A Cell that runs **comments_answers_sdf.printSchema().**
6. Two consecutive Cells that run **comments_questions_sdf.count()** and **comments_questions_sdf.show(5)**.
7. A Cell that runs **comments_questions_sdf.printSchema().**
8. Two consecutive Cells that run **graph_sdf.count()** and **graph_sdf.show(5)**.
9. A Cell that runs **graph_sdf.printSchema().**

Step 2.4 Simple Analytics on the Social Data

In this section, we shall be executing Spark operations on the data given. Beyond simply executing the queries, the students are advised to think about how the query is executed. You may try using **.explain()** method to see more about the query execution. Also, please read the data description prior to attempting the following questions to understand the data.

2.4.1 Most Active users

Now, use the appropriate DataFrame to find the user who has answered the most (highest number of) questions (assume each user can answer each question only once).

"Query" for Step 2.4.1.
1. Find the Markdown Cell that says "Step 2.4.1 Results"
2. In the next cell, output the top 10 users by number of questions. Follow the schema (user, ansCounts)
3. In the next cell, output the top 10 users by number of answers to questions by distinct users .Follow the schema (user, userCounts)

2.4.2 Ignored users

Find out if there exists any users who have been ignored by the StackOverFlow community (user has no answers/comments to his/her questions).

"Query" for Step 2.4.2.
1. Find the Markdown Cell that says "Step 2.4.2 Results"
2. In the next cell, Output the number of users whose questions have never been answered or commented on. You may use **count()** to simply output the size.

2.4.3 Most helpful user pairs

Now, use the appropriate data frames to find the pairs of users who have helped out each other, i.e., users how have answered the most number of each other's questions (if A answered 10 questions by B, and B answered 5 questions by A, then ansCounts has to be 15).

"Query" for Step 2.4.3.
1. Find the Markdown Cell that says "Step 2.4.3 Results"
2. In the next cell, output the top 10 pairs of users along with the total count of the questions they have answered between them. Follow the schema (user1, user2, ansCounts)

Step 3.0 Computing Simple Graph Centrality

The study of networks has proposed a wide variety of measures for measuring *importance* of nodes. A popular metric that is easy to compute is the *degree centrality*. The degree centrality of a node is simply the number of connections to the node. In a directed graph such as ours, you will want to compute both the **indegree centrality** (number of nodes with edges coming **to** this node) and **outdegree centrality** (number of nodes with edges coming **from** this node).

Using the Spark DataFrame **groupBy(), orderBy()**, and **count()** functions, or their equivalents in SparkSQL using the **spark.sql()** function, compute a DataFrame with the **five vertices with highest indegree**. Similarly, you can get the **five vertices with highest outdegree**. Think about what these measures could tell you. (You can consult the lecture slides for more details.)

Data Check for Step 3. Let's now putput some results:
1. Find the Cell with contents "Step 3 Results"
2. Execute the Cell that runs **show()** on your Spark DataFrame for the 5 nodes with highest indegree. Include the fields **node** and the **indegree**.
3. Execute the Cell that runs **show()** on your Spark DataFrame for the 5 nodes with highest outdegree. Include the fields **node** and the **outdegree**.

Degree centrality is the simplest of the potential measures of graph node importance. We'll consider other measures in the Advanced part of this assignment, as well as in future assignments.

Step 4.0 "Traversing" a Graph

For our next tasks, we will be "walking" the graph and making connections.

4.1 Distributed Breadth-First Search

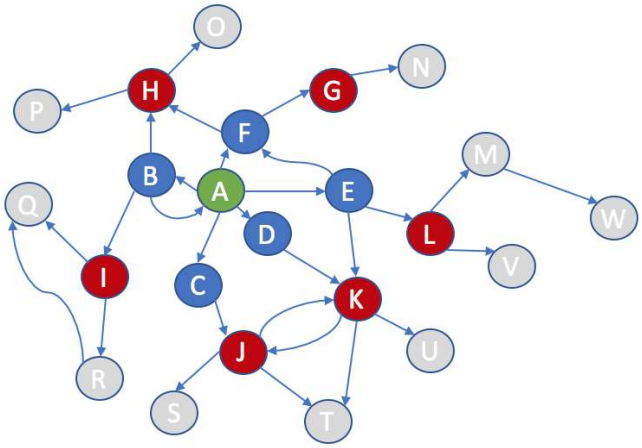
A *search* algorithm typically starts at a node or set of nodes, and "explores" or "walks" for some number of steps to find a match or a set of matches.

Let's implement a distributed version of a popular algorithm, **breadth-first-search (BFS)**. This algorithm is given a **graph *G***, a set of **origin nodes *N***, and a depth **d**. In each iteration or round up to depth **d**, it explores **the set of all new nodes directly connected to the nodes it already has seen, before going on to the nodes another "hop" away**. If we do this correctly, we will explore the graph in a way that (1) avoids getting caught in cycles or loops, and (2) visits each node in the fewest number of "hops" from the origin. BFS is commonly used in tasks such as friend recommendation in social networks.

How does distributed BFS in Spark work? Let's start with a brief sketch of standard BFS. During exploration "rounds", we can divide the graph into three categories:

1. **unexplored** nodes. These are nodes we have not yet visited. You don't necessarily need to track these separately from the graph.
2. **visited** nodes. We have already reached these nodes in a previous "round".
3. **frontier** nodes. These are nodes we have visited in this round. We have not yet checked whether they have out-edges connecting to unexplored nodes.

We can illustrate these with a figure and an example.



Let's look at the figure. The green node A represents the origin.

1. *In the first round, the origin A is the sole frontier node. We find all nodes reachable directly from A, namely B-F; then we remove all nodes we have already visited (there are none) or that are in the frontier (the node A itself). This leaves the blue nodes B-F, which are all reachable in (at most) 1 hop from A.*
2. *In the second round, we move A to the visited set and B-F to the frontier. Now we explore all nodes connected directly to frontier nodes, namely A (from B), F (from E), and the red nodes G-L. We eliminate the nodes already contained in the frontier and visited sets from the next round's frontier set, leaving the red nodes only.*
3. *In the third round, we will move B-F to the visited set, G-L to the frontier set, and explore the next round of neighbors N-V. This process continues up to some maximum depth (or until there are no more unexplored nodes).*

Assume we create data structures (we can make them DataFrames) for the visited and frontier nodes. Consider (1) how to initialize the different sets at the start of computation [note: unexplored nodes are already in the graph], and (2) how to use the graph edges and the existing data structures to update state for the next iteration "round".

*You might possibly have seen how to create a breadth-first-search algorithm in a single-CPU programming language, using a **queue** to capture the frontier nodes. With Spark we don't need a queue -- we just need the three sets above.*

Create a function **spark_bfs(G, origins, max_depth)** that takes a Spark DataFrame with a graph **G** (following the schema for **comments_questions_sdf** described above), a Python list of maps **origins** of the form `[{'node': nid1}, {'node': nid2}, …]`, and a nonnegative integer "exploration depth" **max_depth** (to only run BFS on a tractable portion of the graph). The function should return a DataFrame containing **pairs of the form *(n, x)*** where *n* is the node or vertex ID, and *x* is the depth at which *n* was first encountered. *Note that the origin nodes should also be returned in this Spark DataFrame (with depth 0)!*

You can create a new DataFrame with an integer **node** column from the above list of maps **origins**, as follows. This will give you a DataFrame of the nodes to start the BFS at

```
schema = StructType([
            StructField("node", IntegerType(), True)
        ])

    my_sdf = spark.createDataFrame(my_list_of_maps, schema)
```

A few hints that should keep Spark from getting too bogged down:

1. In this algorithm, do not traverse edges back to previously visited nodes (this is

reach the same node through multiple edges during a single step of the traversal, you should keep the duplicate entries (including parallel edges) as they will be useful for 4.2 for finding "friends of multiple friends." However, don't keep duplicate paths that started in previous layers of the BFS (i.e. your **frontier** DataFrame should be a set before the start of each iteration).
2. For performance reasons, as you compute DataFrames that are referenced more than once -- you should call the **.cache()** function: e.g., `my_sdf = dataframe_expression.cache()`. You should definitely do this with your graph, visited node set, etc.
3. For performance reasons, you may want to divide your DataFrames into many shards, likely on the **from_node** or **node**. You can do this by running `my_sdf.repartition(100, my_column)`.

An important note: PySpark's `subtract` function may seem useful in this problem. However, one thing to know about it is it treats the inputs as sets; that is, if we run the `subtract` function it will automatically remove duplicate rows from the input DataFrames. For example, if we call `sdf1.subtract(sdf2)`, any duplicate rows in `sdf1` and `sdf2` will be dropped when it performs the computation. This means that if you are relying on one of your DataFrames to store nodes seen multiple times at a depth level, there is a good chance these duplicate rows are being dropped if you are using the `subtract` function. One way around this is to use what Spark calls a *Left Anti-Join*. This is syntactic sugar for the following SQL query:

```
SELECT *
FROM table1 AS t1
LEFT JOIN table2 AS t2
ON t1.col = t2.col
WHERE t2.col IS NULL
```

To run a Left Anti-Join in Spark, you can use a join with the 'how' parameter set to 'leftanti', e.g. `...sdf1.join(sdf2, sdf1.col==sdf2.col, 'leftanti')`. When running this, you may or may not get a Spark error that includes the message: "`AnalysisException: 'Both sides of this join are outside the broadcasting threshold and computing it could be prohibitively expensive. To explicitly enable it, please set spark.sql.crossJoin.enabled = true;'`".
If you do, try running the `join` on its own line instead of chaining it with other commands: e.g. instead of

```
my_awesome_sdf = my_awesome_sdf.computation1().computation2().join(...)
```

Try running:

```
my_awesome_sdf = my_awesome_sdf.computation1().computation2()
my_awesome_sdf = my_awesome_sdf.join(...)
```

Data Check for Step 4.1. Let's output some results:
1. Compute a Spark DataFrame called **bfs_sdf** with the results of **spark_bfs(comments_questions_sdf, [{'node': 4550}, {'node': 242}], 2)**. (We are not asking for the computation over **graph_sdf** simply because it takes a long time to complete.)
2. Find the Cell with contents "Step 4.1 Results"
3. Run two consecutive Cells that call **count()** and then **show()** on the above Spark DataFrame.

Note that in a data-parallel setting like this, the BFS algorithm you've implemented above actually is a form of (partial) "transitive closure."

Step 4.2. Friend Recommendation

Now create a function **friend_rec** that takes in two arguments: **filtered_bfs_sdf** and **graph_sdf** (note: not necessarily the **graph_sdf** you created earlier). **filtered_bfs_sdf** is a subset of the rows of **bfs_sdf**. **graph_sdf**, in this case, will be **comments_questions_sdf**. **friend_rec** should return the set of recommendations for nodes that are **not adjacent to one another**. For each pair (n1, n2), where both n1 and n2 are non-adjacent nodes in **filtered_bfs_sdf**, the output DataFrame should have an edge (in both directions) between n1 and n2 to recommend the two nodes become friends. This is the notion of **triadic closure** (completion of triangles) that is commonly used in social networks like Facebook.
Hint: Since the graph is directed, in order to ensure that the nodes aren't already connected to one another, you have to check that there is no edge in either direction in the graph.

Data Check for Step 4.2. Let's now output some results:
1. Compute a new Spark DataFrame **filtered_bfs_sdf**: take the **bfs_sdf** DataFra

and choose all items at **depth 2** that are seen **more than once** at that depth.
2. Compute a Spark DataFrame with the results of **friend_rec()** on this DataFrame.
3. Find the Cell with contents "## Step 4.2 Results"
4. Create two consecutive Cells that call **count()** and then **show()** on the friend_rec() output Spark DataFrame from above.

Step 4.3. Friend Visualization

We'll now make use of the **networkx** graph visualization tool, which lets us see what the graph actually looks like. Make sure you have run the line at the top of the supplied Graphs.ipynb that says "!pip install networkx". (You only need to run this one time for your notebook.)

Data Check for Step 4.3. Let's now create some Cells:
1. In the TODO area for Step 4.3, fill out the code to convert from **friend_recommendations_sdf** (Spark DataFrame) to a Pandas DataFrame. [Hint: there is a Spark DataFrame method called **toPandas()**].
2. Call **nx.from_pandas_dataframe** with two parameters: (1) the Pandas dataframe from the step above, (2) the source and destination fields in the dataframe (e.g., from_node, to_node). Assign the results to a variable called **friend_graph**.
3. Find the Cell with contents "Step 4.3 Results"
4. Run the Cell that prints the properties of the graph
5. Run the Cell that draws the graph

Now, save your Jupyter Notebook (again, as `Graphs.ipynb`) for submission. If you like, you may go on to the Advanced steps, or you may submit the basic assignment as per Step 6.0.

Step 5.0 Advanced - PageRank

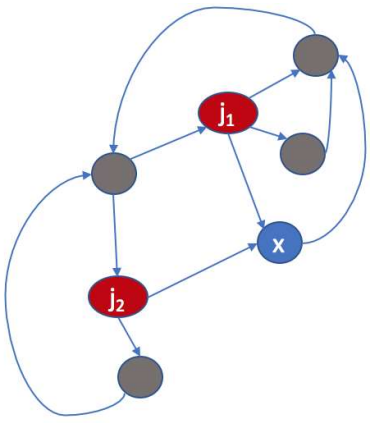
Many of you may be familiar with the **PageRank** computation, which is used to measure the importance of a Web page. (Contrary to popular belief, PageRank is named after Larry Page, not Web pages…) PageRank is actually a tweaked version of a centrality measure called **eigenvector centrality**. One way to implement PageRank is as an **iterative** computation. We take each graph node *x* and in iteration 0 assign it a corresponding PageRank p_x:

$$p_x^0 = 1 / N$$

where N is the total number of nodes.

Now in each iteration *i* we recompute:

$$p_x^{(i)} = \alpha * \sum_{j \in B(x)} (1/N_j)\, p_j^{(i-1)} + \beta$$



Where *B(x)* is the set of nodes linking to node *x*, and N_j is the outdegree of each such node *j*. Typically, repeating the PageRank computation for a number of iterations (15 or so) results in convergence within assignment we'll assume β = 0.15 and α = 0.85 (we'll discuss these in more details in the

Example. *In the figure to the right, nodes* j_1 *and* j_2 *represent the back-link set* B(x) *for node* x. N_{j1} *is 3 and* N_{j2} *is 2. Thus in each iteration* i, *we recompute the PageRank score for* x *by adding half of the PageRank score for* j_2 *and a third of the PageRank score of* j_3 *(both from the previous iteration* i-1*).*

For your solution to this step:

- Create a new Jupyter Notebook, called **Advanced**. Connect to Spark as for the basic part of this assignment.
- Import the file **`pr_graph.txt`** as a graph DataFrame called **pr_sdf**, following a similar process to the one we used with **answers_sdf**. We have left a 3rd column in the timestamp position to make it consistent with the other graphs -- however, this is always set to 0 and won't be used. (You don't need the **edge_type** column here.)
- Write the function **pagerank(G, num_iter)** which takes a graph DataFrame **G** corresponding to your graph, and runs for **num_iter** steps. It should return a DataFrame with columns **(**node_id, pagerank).

Hint. Build some "helper" DataFrames. We suggest at least 2 DataFrames, where the first is used the build the second, and the second is used in your solution:

1. a DataFrame with each **from_node** and the **proportion of weight** it transfers to each outgoing edge. For instance, if the **from_node** is node *j* then the proportion of weight should be $1/N_j$.
2. a DataFrame, again with the **from_node**, each node it transfers weight to, and the proportion of weight computed in (1). For instance, if the **from_node** is *j* and the **to_node** is *x*, then the tuple should be (j, x, $1/N_j$).

Initialize your PageRank values for each node in the "base case". Then, in each iteration, use the helper DataFrames to compute PageRank scores for each node in the next iteration.

You will likely find it easier to express some of the computations in SparkSQL. If you want to use **spark.select**, you may find it useful to use the Spark **F.udf** function to create functions that can be called over each row in the DataFrame. You can create a function that returns a **double** as follows:

```
my_fn = F.udf(lambda x: f(x), DoubleType())
```

Then you can call it like:

```
my_sdf.select(my_fn(my_arg)).alias('col_name')
```

Data Check for Step 5. Let's now create some Cells:

1. Find the Cell with contents "Step 5 Results"
2. In consecutive Cells, write the **count**() of **pr_sdf** and then call **pr_sdf.show()**.
3. In a Cell, call **show()** on the output DataFrame for **pagerank(pr_sdf, 5)**.

Save your Jupyter notebook as **Advanced.ipynb**.

6.0 Submitting Homework 2

Please sanity-check that your Jupyter notebooks contain both code and corresponding data. Then copy you the files listed below, using the same technique you used in Homework 1. Add the notebook files to **hw2.zip** using the **zip** command at the Terminal, much as you did for HW0 and HW1. The notebooks should be:

- `Graphs.ipynb`
- `Advanced.ipynb` (if applicable)

Next, go to the submission site, and if necessary click on the Google icon and log in using your Google@SEAS or GMail account. At this point the system should know you are in the appropriate course. Select CIS 545 **Homework 2** and upload `hw2.zip` from your Jupyter folder, typically found under `/Users/`*{myid}*.

If you check on the submission site after a few minutes, you should see whether your submission passed validation. You may resubmit as necessary.