09 NOV andrew:
set up a to-do list organized by the big categories of work to be done. [link]

10 NOV andrew:
spent the past couple of days figuring out how to customize the twitter streaming api, and how to get it to play well with python (and with spark, kind of). reading lots and lots on twitter streaming, spark streaming, and how/when python has a usable interface. wonder why python lags in the dev queue behind java and scala for spark?

just discovered in the spark fine print that, as of 1.5.1, spark streaming doesn't have a python implementation of its direct-to-tweet-stream ingestion service. bummer. thought that would be a major wrench in our plans...
but today i found this workaround, which uses the `stream=True` property on a `requests.get()` object to act as a generator, which can then feed into a Spark DStream. (blog writeup of that project here.)

that works, i tested it out and i can get tweets outputted from an RDD collect(). but it's a little hinky the way it's set up. it basically initializes empty RDDs and then uses a transform to pass them through the incoming stream from requests. it's not bad, but it's not great as an implementation. i think that's due to a combination of the author being creative but not totally savvy, mixed with pyspark just not really being designed to handle custom input streams.

11 NOV andrew:
pyspark **can** handle Kafka feeds, which i had been avoiding because it sounded scary. but today i took the dive and waded through all of the kafka docs to figure out how it all works. it's basically a fancy queue with good tolerance for distributed inputs and outputs, and it works well in the apache ecosystem. so i am still using the requests module to pull off tweets one by one from the streaming feed i have set up. but now that goes into a kafka producer. (there's a kafka-python wrapper.) next step is to hook up a spark DStream as a consumer.

- there's a lot on kafka-->spark (example in java),
- and on twitter-->kafka,
- and even some on twitter-->kafka-->spark,
- but almost nothing on **python(**twitter-->kafka-->spark**)**.
- this is python(kafka-->spark), which will be useful.
- (there's also python(twitter-->spark), as linked above, but we can do better than that, i think.)
- **python(twitter-->kafka-->spark)-->d3** seems to be totally undocumented (or un-googlable)
- there are a few examples of spark-->d3 out there (with an interesting use of spark-as-a-service (git repo here), which i hadn't seen before - that example is in scala though).

- This example (also Scala) goes kafka-->spark-->hbase-->d3, which is a nice reference. not sure if it's justified to add another middleman (ie hbase) into the mix for the level of data we're ingesting, but it's a good thought in case we need it. i also have never worked with hbase, which ups the intimidation factor a bit. i am just starting to feel good about kafka!)
- here are the official spark repo's examples for pyspark streaming use cases.
- we should be able to piece it all together.

Note: Kafka topics can be deleted with the terminal command:
`kafka-topics.sh --delete --zookeeper localhost:2181 --topic my-topic`
This only works if you have delete.topic.enabled=true in the server.properties config file.

Update: Apparently the .jar file for KafkaUtils is not included with the Spark 1.5.1 install, even though pyspark has a module that can use it. This took awhile to figure out. Now I have it working from spark-submit, but I still can't figure out how to add the jar to SparkConf() to make it work in ipython notebook. This is not a major issue, but a little bit annoying.

Update: We have a working pipeline up to spark output, sort of. Right now we have the Twitter streaming API feeding into `requests.get()`, which writes to a running Kafka producer (basically a streaming queue that listens for incoming data). Spark starts up as a separate script, and uses `KafkaUtils.createDirectStream()` to tap in directly to the Kafka stream. `createDirectStream` outputs a DStream, which can be operated on more-or-less like a normal Spark RDD. from there I apply some simple maps to filter down to the username and tweet in each entry, and then we print that output to stdout in the terminal. This works.

There are still some things to do before I feel confident saying that the pipeline is secure and functional up to this point:

1. **How much data can this Dstream/RDD thing see at any given moment?** Look into the BATCH_INTERVAL and BLOCKSIZE arguments that the hacky twitter-spark repo used. It's nice that we can print, but that just prints everything as it comes in. We want to collect aggregate measures per <time period> in order to get sentiment scores, etc. We also want to be able to pick the most-extreme-sentiment tweet in a given 1- or 5-minute window (this is an early idea but we should have the capability). So, figure out how much the stream RDD can get its hands on.
2. **How does partitioning work here? Also, is multi-thread concurrency helpful?** It's a bit cramped on my local machine because I think zookeeper/kafka take up one or two cores, and I only have 4. That means Spark only has 2-3 cores max to work with (and we probably want one free for system ops). But we should be able to assign 2-4 processes per core, if we're not multi-threading. So, figure out how much you have available in terms of resources, and how to allocate with this streaming spark regime.
3. **Look into using SQLcontext() for processing data**. This could make the sentiment scoring go a bit more smoothly. Also, how does **toPandas()** work?

4. **Figure out an intelligent way to stop the streaming context (shut down).** We currently have the python program that feeds the Kafka producer running for a fixed amount of time (set at 2 minutes right now). But when it stops giving new tweets, it's still giving timestamps (can we suppress this?), and so Spark Stream still has something coming in. Even if it didn't, I think it'd still just keep going until it received a command to stop. There's an `.awaitTermination()` method, which can take <number of seconds> as an argument. But if you don't pass it anything, how does it know when to stop? You can also just call `ssc.stop()` (ssc=spark streaming context), but you need something to trigger that at the right time. There may be some way to use `count()` to pay attention to how much new data are coming into the RDDs at any given time, as per this thread. I read it a couple of times and didn't really understand it, so there's some googling to be done there.
5. Once you have all this figured out, then would be a good time to do a quick check-in on AWS and make sure you can get all this up and running on an EMR cluster. might also want to bake an AMI that has all the stuff you need ready to go.

12 NOV andrew