CMPT 898 HOME EXERCISE 8
PROCESSING TWEETS VIA THE TWITTER SEARCH API & TWITTER4J

This exercise will build on class exposure to the Twitter search API, in light of learning regarding functional and imperative programming styles. A later exercise will focus on twitter streaming.

Recall that the Twitter search API allows for harvesting results using a *paging* model, in which received results are not all obtained at once, but are instead obtained in batches (one "page" at a time), similar to how a human user processes search queries in a browser. Among other factors, this allows twitter to avoid speculatively delivering all of the data at once to the user when only some of that data may be needed. This model also provides flexibility for the provider to update results for some pages if required.

Recall that the twitter4j version of the API covered calls search() on a Twitter object. This search method takes a Query object in as an argument, and returns a TwitterResponse. On TwitterResponse, the .hasNext method can be used to determine if there are further results available. If there are, the .nextQuery method can be called on a TwitterResponse to obtain a query that can be (in turn) processed through the search method on the Twitter object.

Please recall further that the Query object can be built up successively -- and in a functional manner – by allocating a Query object using new Query(), and then calling successive methods on that object (e.g., query, geoCode, since, count, resultType). Please be sure to always call "count" (specifying the number of results to obtain per page – 100 is suggesed). While we did not cover it in class, calling "resultType" with the argument Query.RECENT is also recommended, so as to return the most recent results.

Within this take-home exercise, you will provide two different implementations of a method called *identifyInterestingTweets*. This function will serve to harvest tweets deemed interesting from a search, moving as required through successive pages, and returning all the tweets judged to be of interest and metadata on the tweets obtained and those processed along the way. Both implementations of this method have identical externally observable behaviour – they take in as arguments a maximum count of tweets to examine (process), a predicate (a function) which takes as an argument a Status associated with a tweet which then returns a boolean indicating if that tweet (status) is of interest to harvest, and a Query to execute to obtain the tweets. This method *identifyInterestingTweets* returns a 4-tuple as a result. The elements of this 4-tuple consist of (respectively), the total number of tweets processed (whether judged interesting or not), the total number of "interesting" tweets found (as judged by the predicate returning true for such tweets), a collection (Seq[Status]) of the status associated with tweets that are interesting, and the earliest tweet Date encountered while processing the tweets needed to find the above. This method (regardless of impolementation) shoudl only look through the maximum number of tweets as specified by the user in the first parameter to the method.

One general implementation note: While – per good software engineering practice – the users of *identifyInterestingTweets* will know only that the 3rd element of the tuple returned by the function is a Seq[Status] (rather than it being an instance of any particular class), for both implementations below, you are requested to use geographia V*ector[Status]* (which is a subtype of and can thus be returned as a Seq*[Status]*). A motivation for using such a Vector as the implementation of Seq[Status] is the fact that a Vector allows nearly constant-time appends of one vector to another (using the "++" operator), which is how you will join together successive Sequences obtained from different pages.

Two additional points:

- Please remember when testing your method that the Twitter search API restricts the number of tweets provided within a 15 minute period. Particularly initially, you are advised to please test your *identifyInterestingTweets* with a smaller maximum count of tweets to examine (e.g., a few thousand at most per invocation). When it is working, and in a separate 15 minute period you can test it in retrieving a larger number of tweets.
- Please also note that, for reasons unclear to the instructor, the Twitter search API often appears to provide access to a rather longer date sequence when an actual query search term (e.g., "influenza") is specified, rather than when a search is made without any such search string (and only with other constraints, such as by geography or time period). You are thus advised to use a query search term (and one that occurs at least reasonably often) for testing.

a) Please provide an implementation of the *identifyInterestingTweets* method that is *imperative* in method, in that works using a do-while loop, going through each successive page (as required) to examine the tweets on the page, and identify those of possible interest. To cleanly implement this imperative version of the method, you will want to use variables (using Scala's "var" construct). Please note that while you will use a loop to go through the pages, you are not required to undertake all operations in an imperative manner; for example, you may wish to process tweets within a given page in a functional manner.

b) In the second implementation, please provide a recursive implementation of *identifyInterestingTweets.* (For this problem, you may assume that the standard stack size for recursive invocations is sufficient to handle the recusive calls required). Within this recursive method, the parameters passed to the *identifyInterestingTweets* method are identical to what is seen in the imperative style implementation, but when processing of further pages is required, instead of looping into such pages, a given invocation the method *identifyInterestingTweets* will instead *recusively call itself* with a query, and with the appropriate parameters, and then – following a return from that function – will return its own values, taking into account both the values returned from the recursive calls and the information gleaned from this page. Please note that – as is typical for recursive methods – there needs to be a "base case" where no recursive calls are performed. In this case, that base case will occur when no further pages are required for processing.