

Dealing Data

A blog dedicated to learning data science techniques by applying them to real world data.

[Home](#)[About](#)[Archive](#)

Subscribe to our mailing list

Email Address

Subscribe



© 2016. All rights reserved.

Dealing Data

PoGo Series: Collecting Tweets with Tweepy

23 Jul 2016

Table of Contents:

1. [Introduction to the Series](#)
2. [The Twitter API](#)
3. [The Tweepy Library](#)
4. [Naive Bayes Classifiers](#)
5. [Training a Sentiment Analyzer](#)
6. [Statistical Analysis of the Data](#)
7. [Visualizing the Data with Choropleth Maps](#)

This is the third post in an on-going Pokemon Go analysis series. [Last time](#), we discussed Twitter's Search and Streaming APIs. In this post, we'll

discuss how to use Python's Tweepy library to interface with Twitter's API and start collecting tweets about the Pokemon Go teams.

We'll cover the following topics:

1. [The Tweepy Library](#)
2. [Using the Search API to collect historical tweets](#)
3. [Using the Streaming API to collect new tweets](#)

The Tweepy Library

There are a number of Python libraries that connect to the Twitter API. Twitter provides a [full list of API libraries](#) for both Python and other languages. Some of the most popular among these include [python-twitter](#), [Twython](#), and [Tweepy](#). I'm most familiar with Tweepy, so we'll be using that for our analysis. In practice, Tweepy is a simple wrapper for the Twitter API. It simplifies the interaction between Python and Twitter's API by handling complications such as rate limiting behind the scenes. Installing Tweepy is as simple as typing `pip install tweepy` into a command line. For more information about Tweepy, read the [developer's introduction to Tweepy](#).

Using the Search API to collect historical tweets

In this section, we'll look at how to collect historical tweets using Tweepy and Twitter's Search API. The first thing we need to do is import the `sys`, `os`, `jsonpickle`, and `tweepy` libraries into python.

```
#Importing libraries
import sys
import os
import jsonpickle
import tweepy
```

Recall that our goal is to make a map of the United States which displays the dominance of each team in each state. Since we aren't interested in tweets from outside of the United States, we'll want to add a location filter when we use the Search API. This requires us to use the [Geo Search REST API](#) to find Twitter's `place_id` for the United States. Note from the [geo_search documentation](#) that we have to use user-authentication for such a query. We set up user authentication in Tweepy with the following commands:

```
#Pass our consumer key and consumer secret to Tweepy's user authentic
auth = tweepy.OAuthHandler(consumer_key, consumer_secret)

#Pass our access token and access secret to Tweepy's user authenticat
auth.set_access_token(access_token, access_secret)

#Creating a twitter API wrapper using tweepy
```

```
#Details here http://docs.tweepy.org/en/v3.5.0/api.html
api = tweepy.API(auth)

#Error handling
if (not api):
    print ("Problem connecting to API")
```

Make sure to replace

`consumer_key` , `consumer_secret` , `access_token` , and `access_secret` with your personal keys and secrets. Once we have the API object instantiated we can use Tweepy's `geo_search` method to query the Geo Search API:

```
#Getting Geo ID for USA
places = api.geo_search(query="USA", granularity="country")

#Copy USA id
place_id = places[0].id
print('USA id is: ',place_id)
```

```
USA id is: 96683cc9126741d1
```

Great, now we know which `place_id` to use as a filter in the Search API. Next, we'll use the Search API to collect historical tweets. Recall:

- The search API only collects a random sample of tweets
- There is no way to get more than the last week of tweets without purchasing them from a third-party provider

From the [Search API documentation](#), note that we can use either user-authentication or application-only authentication to access the Search API. User-authentication allows for 180 queries per access token every 15 minutes, and application-only authentication allows for 450 queries every 15 minutes. Since we only have one access token to work with, we'll want to switch to application-only authentication for the higher rate limit. Tweepy has a separate `AppAuthHandler` method for this purpose:

```
#Switching to application authentication
auth = tweepy.AppAuthHandler(consumer_key, consumer_secret)

#Setting up new api wrapper, using authentication only
api = tweepy.API(auth, wait_on_rate_limit=True,wait_on_rate_limit_no

#Error handling
if (not api):
    print ("Problem Connecting to API")
```

You may have noticed that we added two arguments to the API wrapper this time around. The Search API returns a maximum of 100 tweets per query. We have 450 queries available to us every 15 minutes using application authentication, so we can collect 45,000 tweets every 15

minutes. If we exceed this rate, the `wait_on_rate_limit` argument tells tweepy to pause until we are no longer rate limited. If

`wait_on_rate_limit` is set to the default value of `False`, Tweepy would just return an error and stop searching once we reached our rate limit. If we do end up being rate limited, the

`wait_on_rate_limit_notify` argument tells Tweepy to display a notification that lets us know it is waiting for the 15 minute rate limit window to refresh. These arguments were also available when we were using user-authentication with the Geo Search API, but we did not need to use them since we were only sending one query. As a side note, if you ever want to check the status of your rate limits you can use Tweepy's `rate_limit_status` method, as shown below:

```
#You can check how many queries you have left using rate_limit_status
api.rate_limit_status() ['resources'] ['search']
```



```
{ '/search/tweets': { 'limit': 450, 'remaining': 450, 'reset': 1470850
```



Now that we have an API object using application-only authentication, we're ready to query the Search API for tweets about each Pokemon Go team. First, we define the search query that we'll send to the API. In this case, we'll want to search for any phrase that is likely to identify tweets about each Pokemon Go team, but will not grab generic tweets about Pokemon, Pokemon Go, or other topics. After looking through trending hashtags and tweets about Pokemon Go, I chose the following list of phrases:

- #teammystic
- #teaminstant
- #teamvalor
- #teambblue
- #teamyellow
- #teamred
- #mystic
- #valor
- #instinct
- "team mystic"
- "team valor"
- "team instinct"

Note that the search phrases are not case sensitive. We can search for all of these terms at once by separating them with `OR` in our query. I've also included a filter that requires the tweet originate within the United States by including `place:96683cc9126741d1` at the beginning of the query, where `96683cc9126741d1` is the US `place_id` we found with the Geo Search API.

```
#This is what we are searching for
#We can restrict the location of tweets using place:id
#We can search for multiple phrases using OR
searchQuery = 'place:96683cc9126741d1 #teammystic OR #teaminstinct OF
               '#teambblue OR #teamyellow OR #teamred OR' \
               '#mystic OR #valor OR #instinct OR' \
               '"team mystic" OR "team valor" OR "team instinct"'
```

We also need to define the maximum number of tweets that we want to collect, and the maximum number of tweets that we want to receive from each query that we send:

```
#Maximum number of tweets we want to collect
maxTweets = 1000000

#The twitter Search API allows up to 100 tweets per query
tweetsPerQry = 100
```

Note that the Search API limits us to 100 tweets per query, and that we may not collect the maximum number of tweets that we specified since the Search API only collects the past week of tweets. The easiest way to send our query to the Search API is through [Tweepy's Cursor method](#). The Cursor will automatically send queries to the Search API until we have collected the maximum number of tweets that we specified, or until we reach the end of the Search API database. We can iterate over each tweet that the Cursor gathers using a `for` loop, and write the JSON formatted tweet to a text file using the `jsonpickle` library:

```
tweetCount = 0

#Open a text file to save the tweets to
with open('PoGo_USA_Tutorial.json', 'w') as f:

    #Tell the Cursor method that we want to use the Search API (api.s
    #Also tell Cursor our query, and the maximum number of tweets to
    for tweet in tweepy.Cursor(api.search,q=searchQuery).items(maxTw

        #Verify the tweet has place info before writing (It should, i
        if tweet.place is not None:

            #Write the JSON format to the text file, and add one to t
            f.write(jsonpickle.encode(tweet._json, unpicklable=False
            tweetCount += 1

    #Display how many tweets we have collected
    print("Downloaded {0} tweets".format(tweetCount))
```

Downloaded 18850 tweets

Let's take a closer look at the information we are storing in the text file. The Twitter API returns the tweet in [JSON format](#), and Tweepy parses the JSON text into a tweet class. First, let's look at the raw JSON text that

Twitter returns. This is what we are saving to our text file in the `for` loop:

```
#Let's look at what we are actually printing to text (if you don't k
tweet._json
```

```
{'contributors': None,
 'coordinates': None,
 'created_at': 'Tue Aug 02 19:51:58 +0000 2016',
 'entities': {'hashtags': [],
 'symbols': [],
 'urls': [],
 'user_mentions': [{'id': 873491544,
 'id_str': '873491544',
 'indices': [0, 13],
 'name': 'Kenel M',
 'screen_name': 'KxSweaters13'}]},
 'favorite_count': 1,
 'favorited': False,
 'geo': None,
 'id': 760563814450491392,
 'id_str': '760563814450491392',
 'in_reply_to_screen_name': 'KxSweaters13',
 'in_reply_to_status_id': None,
 'in_reply_to_status_id_str': None,
 'in_reply_to_user_id': 873491544,
 'in_reply_to_user_id_str': '873491544',
 'is_quote_status': False,
 'lang': 'en',
 'metadata': {'iso_language_code': 'en', 'result_type': 'recent'},
 'place': {'attributes': {}},
 'bounding_box': {'coordinates': [[[-71.813501, 42.4762],
 [-71.702186, 42.4762],
 [-71.702186, 42.573956],
 [-71.813501, 42.573956]]],
 'type': 'Polygon'},
 'contained_within': [],
 'country': 'United States',
 'country_code': 'US',
 'full_name': 'Leominster, MA',
 'id': 'c4f1830ea4b8caaf',
 'name': 'Leominster',
 'place_type': 'city',
 'url': 'https://api.twitter.com/1.1/geo/id/c4f1830ea4b8caaf.js
'retweet_count': 0,
'retweeted': False,
'source': '<a href="http://twitter.com/download/android" rel="nc
'text': '@KxSweaters13 are you the kenelx13 I see owning leomins
'truncated': False,
'user': {'contributors_enabled': False,
 'created_at': 'Thu Apr 21 17:09:52 +0000 2011',
 'default_profile': False,
 'default_profile_image': False,
 'description': 'Arbys when it's cold. Kimballs when it's warm.',
 'entities': {'description': {'urls': []}},
 'favourites_count': 1106,
 'follow_request_sent': None,
 'followers_count': 167,
 'following': None,
 'friends_count': 171,
 'geo_enabled': True,
 'has_extended_profile': False,
 'id': 285715182,
 'id_str': '285715182',
```

```
'is_translation_enabled': False,
'is_translator': False,
'lang': 'en',
'listed_count': 2,
'location': 'MA',
'name': 'Steve',
'notifications': None,
'profile_background_color': '131516',
'profile_background_image_url': 'http://abs.twimg.com/images/tl
'profile_background_image_url_https': 'https://abs.twimg.com/i
'profile_background_tile': True,
'profile_banner_url': 'https://pbs.twimg.com/profile_banners/28
'profile_image_url': 'http://pbs.twimg.com/profile_images/7272
'profile_image_url_https': 'https://pbs.twimg.com/profile_imag
'profile_link_color': '4A913C',
'profile_sidebar_border_color': 'FFFFFF',
'profile_sidebar_fill_color': 'EFEFEF',
'profile_text_color': '333333',
'profile_use_background_image': True,
'protected': False,
'screen_name': 'StephenBurke_',
'statuses_count': 5913,
'time_zone': 'Eastern Time (US & Canada)',
'url': None,
'utc_offset': -14400,
'verified': False}}
```

Each one of the JSON attributes is turned into an individual class member in Tweepy's tweet object. I've included a function below that will list all of these class members.

```
#A function to clean up and print all of the JSON attributes
def PrintMembers(obj):
    for attribute in dir(obj):

        #We don't want to show built in methods of the class
        if not attribute.startswith('__'):
            print(attribute)

PrintMembers(tweet)
```

```
_api
_json
author
contributors
coordinates
created_at
destroy
entities
favorite
favorite_count
favorited
geo
id
id_str
in_reply_to_screen_name
in_reply_to_status_id
in_reply_to_status_id_str
in_reply_to_user_id
in_reply_to_user_id_str
is_quote_status
lang
metadata
```

```

parse
parse_list
place
retweet
retweet_count
retweeted
retweets
source
source_url
text
truncated
user

```

Note that each additional layer within a JSON attribute will be parsed as an additional class member within the tweet object:

```

#Can dig into each member as well
PrintMembers(tweet.place)

```

```

_api
attributes
bounding_box
contained_within
country
country_code
full_name
id
name
parse
parse_list
place_type
url

```

As a closing note, I wanted to share the “old” way of using Tweepy. Before the Cursor method was introduced, the process of sending multiple queries until you collected your maximum number of tweets had to be handled manually. You will often see old implementations online which do so, but when possible you should use the Cursor instead. I’ve included an example of our search query using the “old” way below. I hope the example will help you recognize and update any out of date implementations you may come across in your own work.

```

#Old way of doing things - which is what you will see a lot of people
max_id = -1
tweetCount = 0
with open('PoGo_USA_Tutorial.json', 'w') as f:
    #While we still want to collect more tweets
    while tweetCount < maxTweets:
        try:
            #Look for more tweets, resuming where we left off
            if max_id <= 0:
                new_tweets = api.search(q=searchQuery, count=tweetsP
            else:
                new_tweets = api.search(q=searchQuery, count=tweetsP

            #If we didn't find any exit the loop
            if not new_tweets:
                print("No more tweets found")
                break

```



```

#Write the JSON output of any new tweets we found to the
for tweet in new_tweets:

    #Make sure the tweet has place info before writing
    if (tweet.place is not None) and (tweetCount < maxTw
        f.write(jsonpickle.encode(tweet._json, unpicklab
            '\n')
        tweetCount += 1

#Display how many tweets we have collected
print("Downloaded {0} tweets".format(tweetCount))

#Record the id of the last tweet we looked at
max_id = new_tweets[-1].id

except tweepy.TweepError as e:

    #Print the error and continue searching
    print("some error : " + str(e))

print ("Downloaded {0} tweets, Saved to {1}" .format(tweetCount, fNam

```

Using the Streaming API to collect new tweets

If we want to collect new tweets in real-time, we'll need to use Twitter's Streaming API. Since we don't need to worry about rate limits, we'll switch back to using Tweepy's user-authentication handler for this.

```

#Connecting to the twitter streaming API
auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_token, access_secret)

api = tweepy.API(auth)

```

The [Tweepy documentation page](#) tells us how to set up a Streaming API interface in three steps:

1. Create a class inheriting from `StreamListener`
2. Use that class to create a `Stream` object
3. Connect to the Twitter API using the `Stream`.

First, we'll discuss how to inherit and modify the `StreamListener` class. There are two ways to interact with the streaming data - by overloading the `on_data` method or by overloading the `on_status` method. If you want detailed information about the tweets you need to override the `on_data` method. This will allow us to interact with the data at the highest level and opens up information such as replies to statuses, deletions, direct messages, friends, etc. The default `on_data` method from the `StreamListener` class passes data from statuses to the `on_status` method. If we're only concerned with reading the JSON output of tweet statuses, can override the `on_status` method instead of the `on_data` method. This process is shown in the code below:

```
#Inherit from the StreamListener object
class MyStreamListener(tweepy.StreamListener):

    #Overload the on_status method
    def on_status(self, status):
        try:

            #Open a text file to save tweets to
            with open('PoGo.json', 'a') as f:

                #Check if the tweet has coordinates, if so write it t
                if (status.coordinates is not None):
                    f.write(status)
                return True

        #Error handling
        except BaseException as e:
            print("Error on_status: %s" % str(e))

        return True

    #Error handling
    def on_error(self, status):
        print(status)
        return True

    #Timeout handling
    def on_timeout(self):
        return True
```

Now that we have our `MyStreamListener` class, we can use it to create a class object.

```
#Create a stream object
twitter_stream = tweepy.Stream(auth, MyStreamListener())
```

The [Tweepy documentation](#) discusses a number of ways to start the Twitter stream. In our case, we'll want to use the `filter` method to track particular phrases:

```
twitter_stream.filter(track=['#teammystic', 'teaminstinct', '#teamvalo',
                             '#teambblue', '#teamyellow', '#teamred', \
                             '#mystic', '#instinct', '#valor', \
                             'team mystic', 'team instinct', 'team val
```

Note that, we could filter on [other parameters](#), such as locations, instead:

```
twitter_stream.filter(locations=[-122.75, 36.8, -121.75, 37.8])
```

It is important to note that some filters don't interact as expected with each other. For instance, look at the information about the location parameter in the previous link. It states that "Bounding boxes do not act as filters for other filter parameters. For example `track=twitter&locations=-122.75,36.8,-121.75,37.8` would match any

tweets containing the term Twitter (even non-geo tweets) OR coming from the San Francisco area.” Therefore, if we tried to filter on both location and phrases at the same time, we would not get tweets that contain the phrase AND that are from the location. Instead, we’d get tweets that contain the phrase OR are from the location. This is why we need the

```
if (status.coordinates is not None):
```

condition in our class definition.

Closing remarks

That was a lot of content to digest for one blog post! We covered how to use Tweepy to collect historical tweets through Twitter’s Search API, and how to use Tweepy to collect new tweets in real-time through Twitter’s Streaming API. Now that we have a text file full of tweets, we’ll lay the ground work for classifying the tweets as positive or negative in our [next post](#).

Share this:



0 Comments

DealingData

 Login ▾

♥ Recommend

🔗 Share

Sort by Best ▾



Start the discussion...

Be the first to comment.

ALSO ON DEALINGDATA

Second test post

1 comment • 5 months ago •

Richard Knoche — second test comment

Fantasy Football Predictions — A First Look


8 comments • 4 months ago •

Richard Knoche — I used Python's nflldb library to collect stats on previous seasons. You can find details

Test post

1 comment • 5 months ago •

Richard Knoche — Test comment

 Subscribe  Add Disqus to your site Add Disqus Add  Privacy

