pyTweet

This module enables data scientists to build large Twitter datasets for network analytics. It is tedious to obtain large data sets for data analysis, along with developing architecture for processing and storage. With pyTweet, a user can easily implement a sampling method, or use a built-in method, and have the collection run unsupervised. Profile and timeline metadata are saved in either a JSON file format or in a PostgreSQL database with a graph-like schema.

Noteworthy functionality:

- Create a complete data set for graph analysis with Twitter data
- Wrapper for Official Twitter API: profiles, timelines, friends, follows, retweets, etc.
- Pausing for rate limits is built into all API wrapper functions
- Quickly dump collected .JSONs into a relational PostgreSQL data base

Contents of Documentation

1. **Getting Started**: This section describes the software requirements of pyTweet. It also provides instruction on how to create a Twitter application, which is needed for the REST API.
2. **API wrapper functions**: Use the API wrapper functions to create your own method of collecting tweets, profiles, trends and media
3. **Building a Network**: This section documents the available sampling methods in pyTweet as well as the API wrapper functions used in sampling.
4. **The pyTweet PostgreSQL Schema**: This section covers the PostgreSQL database schema on which pyTweet is dependent. Examples of how to create such a database are also included in this section.

Getting Started

Refer to this section to getting pyTweet's dependencies and accessing the Official Twitter API.

Python 2.7 and Installing pyTweet

The module pyTweet is compatible with Python version 2.7 and above. If installing Python for the first time, it's highly recommended that you download it though Anaconda: https://www.continuum.io/downloads. Anaconda is a freemium distribution of Python and R languages for large-scale data processing and computing, that also simplifies package management. In addition to the latest stable version of Python, Anaconda installs Numpy, Scipy, and other helpful modules that are not typically included in the regular Python installation [1]. Once your Python environment has been established, and have download pyTweet as well, install pyTweet locally with the command

```
$ python setup.py install
```

Or install the package with symlink, so that changes to the sources files will be immediately available to other users of the package on your system:

```
$ python setup.py develop
```

Additional Python Modules

The module pyTweet is dependent on several Python modules. Many of these come with the standard version of Python, but some are not included in the standard distribution. Nearly all are installed with Anaconda's version of Python.

Table 1: pyTweet is dependent on the following Python packages

Module	Description	Source
setuptools	Easily download, build, install, upgrade, and uninstall Python packages	https://pypi.python.org/pypi/setuptools
requests	Create and handle web requests	http://docs.python-requests.org/en/latest/
ujson	UltraJSON is an ultra fast JSON encoder and decoder written in pure C with bindings for Python 2.5+ and 3.	https://pypi.python.org/pypi/ujson
re	This module provides regular expression matching operations similar to those found in Perl.	https://docs.python.org/2/library/re.html
requests-oauthlib	OAuth authorization - dependent on oauthlib below	https://github.com/requests/requests-oauthlib
oauthlib	OAuth authorization	https://github.com/idan/oauthlib
psycopg2	Module to interface with PostgreSQL from Python. Only required for submodules json_to_database and depth_first_sampling.	http://initd.org/psycopg/
numpy	Scientific computing module	http://www.numpy.org/
nltk	Natural language toolkit. Only required for submodules json_to_database and depth_first_sampling.	http://www.nltk.org/
sklearn	Machine learning in Python. Only required for submodules json_to_database and depth_first_sampling	http://scikit-learn.org/stable/

Certificate for Official Twitter API

One of the requirements of the Twitter API is to use its certificate. However, this is not a concern while using pyTweet because the module will automatically download and save the certificate if necessary. Whenever a pyTweet command is called and the certificate cannot be found, it is downloaded from http://curl.haxx.se/ca/cacert.pem and saved as <dir to pyTweet>/pyTweet/api.twitter.cer.

Obtaining Twitter API Keys

Twitter's REST API requires API keys for access. To obtain keys you'll need an application in addition to a regular Twitter account. The following steps explain how to create an application and get them:

1. Create a standard Twitter account on https://twitter.com/
2. Go to https://apps.twitter.com/ and login. Click on the *Create New App* button to create an application.
3. Fill in the form, accept the developer agreement, and finish creating your application
4. Go to the 'Permissions' tab, and select read only access
5. Go to the 'Keys and Access Tokens' tab and click the button 'Create my access token'. You may need to refresh the page to view the new tokens. Use the Consumer Key (API Key), Consumer Secret (API Secret), Access Token, and Access Token Secret for authorization in pyTweet.

Configure Twitter API Keys for pyTweet

Once you've obtained at least one set of API keys, format them in JSON files so that pyTweet will be able to recognize the keys. Save the JSON file in the directory */path/to/pyTweet/twitter_api_keys*. Now whenever you run a network sampling script pyTweet will find the keys and authorize a connection with the Official Twitter API.

Notice that all API calls have rate limits. Typically about 15 calls can be executed during a window of fifteen minutes; however these rates can vary by the type of call. When a key has met its rate limit data collection will pause until the window is reset. This pause could last somewhere between fifteen to sixty minutes. Multiple API keys can be saved in separate files like the one below to make sampling faster. When the current key has reached the limit of calls during a window, it is swapped with another key. If there are multiple keys saved in */path/to/pyTweet/twitter_api_keys,* then either the best possible key or a fresh key is returned.

After you have configured the Twitter API keys, you can check if the keys work from time to time with the method *check_twitter_key_functionality* (see examples/test_twitter_api_keys.py). This will test the authentication of every saved key and indicate whether the authentication was successful. For unsuccessful keys, the error code and message are returned directly from the Twitter API. The following is output similar to what you should expect:

```
Test key C:\Anaconda\Lib\site-packages\pyTweet\twitter_api_keys\key3.json
   ERROR CODE 89: Invalid or expired token.
Test key C:\Anaconda\Lib\site-packages\pyTweet\twitter_api_keys\key5.json
   SUCCESSFUL AUTHENTICATION
Test key C:\Anaconda\Lib\site-packages\pyTweet\twitter_api_keys\key7.json
   ERROR CODE 32: Could not authenticate you.
```

Figure 1: Example of API Key JSON file, saved as *pyTweet/twitter_api_keys/example_key.json.* Each file should contain a single set of keys.

```
{
      "ACCESS_TOKEN":"your-access-token",
      "API_SECRET":"your-api-secret",
      "API_KEY":"your-api-key",
      "ACCESS_TOKEN_SECRET":"your-accesss-token-secret"
}
```

Basic API Wrapper Functions

They API wrappers of pyTweet can be used to create your own sampling methods. Generally these methods can be used to collect profile and timeline metadata, friends lists and followers lists. In order to scape data from the API, you must first authenticate with your credentials as shown below. View the full example in the file *pyTweet/examples/basic_command_examples.py*.

```
import pyTweet

# Enter proxy host and port information
host = 'my host'
port = 'my proxy'

# Create proxies dictionary
proxies = {'http': 'http://%s:%s' % (host, port), 'https': 'http://%s:%s' %
(host, port)}

# Load twitter keys
twitter_keys = pyTweet.load_twitter_api_key_set()

# API authorization
OAUTH = pyTweet.get_authorization(twitter_keys)
```

Once, you authenticate with the Official Twitter API you can proceed to use any of pyTweet's basic API wrapper functions. For example, collecting profile dictionary objects giving either a user ID or handle:

```
# Look up a list of user screen names
info_from_name = pyTweet.user_lookup_usernames(user_list=['username1',
'username2'], proxies=proxies, auth=OAUTH)
```

There are few details that one should know about the previous basic API wrapper commands. First, rate limit status checking is built into the wrappers. All of the related API calls have varying limits. When no more calls are available, pyTweet will either pause until there are available calls or authenticate with a new set of keys (if you have more than one API key saved). Second, some profile and timeline metadata are subject to change such as friends, favorites and retweets. It may be helpful to save the date of collection with each profile and tweet dictionary object. For compatibility with the pyTweet module, these dates should be saved as keys named 'DOC' in the dictionary object as datetime objects. This can be done as follows:

```
import datetime
profile_metadata['DOC'] = datetime.datetime.utcnow()
tweet_metadata['DOC'] = datetime.datetime.utcnow()
```

API Wrapper Functions in pyTweet

The following tables describe pyTweet's Twitter API wrapper functions. Note that pausing for the API's rate limits is built into all of these functions. Be sure to check out the directory pyTweet/examples because it contains implementations examples of all of the following functions.

Table 2: pyTweet Functions for API Authorization

Function	Description
load_twitter_api_key_set	This function loads a set of Twitter keys. Parameter: *Key_file*: Key file, not required Return: Dictionary object containing API login information 'API_KEY', 'API_SECRET', 'ACCESS_TOKEN', 'ACCESS_TOKEN_SECRET'
get_authorization	This function obtains an authorization object for accessing the Official Twitter API Parameter: *twitter_keys*: Dictionary object containing API login information Return: Authorization object required for remaining pyTweet collection functions
check_twitter_key_functionality	This function checks your saved Twitter API keys to ensure that they are functional. A message appears indicating each key's status. Parameters: *host*: proxy host *port*: proxy port

Table 3: API Wrappers for collecting Twitter profiles

Function	Description
lookup_users	Returns a list of user dictionaries, as specified by comma-separated values passed to the user_id and/or screen_name parameters. There are a few things to note when using this method. 1. You must be following a protected user to be able to see their most recent status update. If you don' follow a protected user their status will be removed. 2. The order of user IDs or screen names may not match the order of users in the returned array. 3. If a requested user is unknown, suspended, or deleted, then that user will not be returned in the results list. 4. If none of your lookup criteria can be satisfied by returning a user object, a HTTP 404 will be thrown. Parameters: *proxies*: proxy object, ex. {'http': 'http://%s:%s' % (HOST, PORT), 'https': 'http://%s:%s' % (HOST, PORT)} *auth*: Twitter application authentication, see the get_authorization method *screen_names*: List of screen names, or a single one - optional *user_ids*: List of user IDs, or a single one - optional *include_entities*: The entities node that may appear within embedded statuses will be disincluded when set to false.
search_users	Provides a simple, relevance-based search interface to public user accounts on Twitter. Try querying by topical interest, full name, company name, location, or other criteria. Exact match searches are not supported. Only the first 1,000 matching results are available. Parameters: *q*: Search term query, must be a string object or list of string objects

	exclusive: Boolean, if True, search with ORs rather than ANDs. Default is False *proxies*: proxy dictionary *auth*: Twitter application authentication, see the get_authorization method *limit*: limit to number of users to collect. Maximum and default values are 1000 *include_entities*: The entities node will be disincluded from embedded tweet objects when set to false. Return: List of profile dictionaries

Table 4: API Wrappers for collecting friends and followers

Function	Description
get_user_friends	Look up the IDs of all of a user's friends (people they follow), and return them in a list. Parameters: *user_id*: The ID of the user for whom to return results for – optional *screen_name*: The screen name of the user for whom to return results for - optional *limit*: limit to number of friends to collect. Set to None to get all friends. this is the default *proxies*: proxy object, ex. {'http': 'http://%s:%s' % (HOST, PORT), 'https': 'http://%s:%s' % (HOST, PORT)} *auth*: Twitter application authentication, see the get_authorization method
get_user_friend_profiles	Returns a list of user dictionaries for every user the specified user is following (otherwise known as their 'friends'). Parameters: *user_id*: Unique Twitter user ID, optional *screen_name*: The screen name of the user for whom to return results for - optional *limit:* limit to number of friends to collect. Set to None to get all friends. this is the default *proxies:* proxy object, ex. {'http': 'http://%s:%s' % (HOST, PORT), 'https': 'http://%s:%s' % (HOST, PORT)} *auth:* Twitter application authentication, see the get_authorization method
get_user_followers	Look up the IDs of all of a user's followers (people who follow them), and return them in a list of json objects. Parameters: *user_id*: The ID of the user for whom to return results for – optional *screen_name*: The screen name of the user for whom to return results for - optional *limit:* limit to number of friends to collect. Set to None to get all friends - this is the default *proxies:* proxy object, ex. {'http': 'http://%s:%s' % (HOST, PORT), 'https': 'http://%s:%s' % (HOST, PORT)} *auth:* Twitter application authentication, see the get_authorization method
get_user_follower_profiles	Returns a list of user dictionaries for users following the specified user. Parameters: *user_id*: Unique Twitter user ID, optional *screen_name*: The screen name of the user for whom to return results for - optional *limit*: limit to number of friends to collect. Set to None to get all friends - this is the default *proxies*: proxy object, ex. {'http': 'http://%s:%s' % (HOST, PORT), 'https': 'http://%s:%s' % (HOST, PORT)} *auth*: Twitter application authentication, see the get_authorization method

Table 5: API Wrappers for collecting timelines and searching for tweets

Function	Description
get_timeline	Find timeline of a user occurring after *start_date*. Enter either a screen name or user ID. User timelines belonging to protected users may only be requested when the authenticated user either 'owns' the timeline or is an approved follower of the owner. The timeline returned is the equivalent of the one seen when you view a user's profile on twitter.com. This method can only return up to 3,200 of a user's most recent Tweets. Parameters: *user_id*: The ID of the user for whom to return results for - optional *screen_name*: The screen name of the user for whom to return results for - optional

	trim_user: When set to true, each tweet returned in a timeline will include a user object including only the status authors numerical ID. Omit this parameter to receive the complete user object. *exclude_replies*: This boolean parameter will prevent replies from appearing in the returned timeline. Using exclude_replies will mean you will receive up-to count tweets *contributor_details*: This boolean parameter enhances the contributors element of the status response to include the screen_name of the contributor. By default only the user_id of the contributor is included. *include_rts*: When set to false, the timeline will strip any native retweets (though they will still count toward both the maximal length of the timeline and the slice selected by the count parameter). Note: If you're using the trim_user parameter in conjunction with include_rts, the retweets will still contain a full user object. *start_date*: start of timeline segment to collect, this is a datetime.date object. The default value is 52 weeks ago from today *proxies*: proxy dictionary, ex. {'http': 'http://%s:%s' % (HOST, PORT), 'https': 'http://%s:%s' % (HOST, PORT)} *auth*: Twitter application authentication, see the get_authorization method
get_tweets	Returns a list of tweet dictionaries, specified by the id parameter. The Tweet's author will also be embedded within the tweet. Parameters: *tweet_id*: Unique ID of tweet, or list of tweet IDs *include_entities*: The entities node that may appear within embedded statuses will be disincluded when set to false. *trim_user*: When set to either true, each tweet returned in a timeline will include a user object including only the status authors numerical ID. Omit this parameter to receive the complete user object. *keep_missing_twts*: When using the map parameter, tweets that do not exist or cannot be viewed by the current user will still have their key represented but with an explicitly null value paired with it *proxies*: proxy dictionary *auth*: Twitter application authentication, see the get_authorization method
search_for_tweets	This function searches for tweets based on a combination of string queries, geocode, language, date or result types. It returns a list of tweet dictionaries. Note that not all Tweets will be indexed or made available via the search interface. Parameters: *q*: A string or list of strings to query. This function searches for hashtags as well *exclusive*: Boolean, if True, search with ORs rather than ANDs. Default is False *geocode*: Returns tweets by users located within a given radius of the given latitude/longitude. The parameter value is specified by 'latitude,longitude,radius', where radius units must be specified as either 'mi' (miles) or 'km' (kilometers). *lang*: Restricts tweets to the given language, given by an ISO 639-1 code *result_type*: Specifies what type of search results you would prefer to receive. The default is 'mixed'. Valid values include 'mixed' (includes both popular and real time results in the response), 'recent' (return only the most recent results in the response) and 'popular' (return only the most popular results in the response) *limit*: Number of tweets to collect. Set to None to get all possible tweets. The default is 100 tweets. *until*: Returns tweets created before the given date, which should be formatted as YYYY-MM-DD. No tweets will be found for a date older than one week. *locale*: Specify the language of the query you are sending (only ja is currently effective). This is intended for language-specific consumers and the default should work in the majority of cases. *include_entities*: The entities node will be disincluded when set to false. *proxies*: proxy dictionary *auth*: Twitter application authentication, see the get_authorization method
get_retweets	Find all retweets of a given tweet, specified by a numerical tweet ID, and up to 100 per request Parameters: *tweet_id*: Unique ID of tweet *proxies*: proxy dictionary *auth*: Twitter application authentication, see the get_authorization method *trim_user*: When set to either true each tweet returned in a timeline will include a user object including

	only the status authors numerical ID. Omit this parameter to receive the complete user object. *limit*: limit to number of friends to collect. Max 100 (None also specifies 100)
get_tweets_with_hashtag	This function searches for tweets based on a combination of string hastag queries, geocode, langauge, date or result types. Note that not all Tweets will be indexed or made available via the search interface. Parameters: *q*: A string or list of strings to query. This function searches for hashtags as well *exclusive*: Boolean, if True, search with ORs rather than ANDs. Default is False *geocode*: Returns tweets by users located within a given radius of the given latitude/longitude. The parameter value is specified by 'latitude,longitude,radius', where radius units must be specified as either 'mi' (miles) or 'km' (kilometers). *lang*: Restricts tweets to the given language, given by an ISO 639-1 code *result_type*: Specifies what type of search results you would prefer to receive. The default is 'mixed'. Valid values include 'mixed' (includes both popular and real time results in the response), 'recent' (return only the most recent results in the response) and 'popular' (return only the most popular results in the response) *limit*: Number of tweets to collect. Set to None to get all possible tweets. The default is 100 tweets. *until*: Returns tweets created before the given date, which should be formatted as YYYY-MM-DD. No tweets will be found for a date older than one week. *locale*: Specify the language of the query you are sending (only ja is currently effective). This is intended for language-specific consumers and the default should work in the majority of cases. *include_entities*: The entities node will be disincluded when set to false. *proxies*: proxy dictionary *auth*: Twitter application authentication, see the get_authorization method
download_tweet_media	This function downloads a single link (or list) of Twitter media. The media is saved in save_dir, and must have one of the following extensions: 'gif', 'jpg', 'jpeg', 'jif', 'jfif', 'tif', 'tiff', 'png', 'pdf', 'mp4' Parameters: *link*: A single link string, or list of link strings *proxies*: proxy dictionary *save_dir*: Directory to save media files, default is current directory Return: Dictionary that corresponds links to filenames, if any exist

Table 6: API Wrappers for Twitter Geo-Searches

Function	Description
reverse_geocode	Searches for up to 20 places that can be used as a place_id when updating a status. This request is an informative call and will deliver generalized results about geography. Parameters: *lat*: The latitude to search around, which must be inside the range -90.0 to +90.0 *lon*: The longitude to search around, which must be inside the range -180.0 to +180.0 *accuracy*: The hint on the "region" in which to search, in meters, with a default value of 0m. If a number, then this is a radius in meters, but it can also take a string that is suffixed with ft to specify feet. *granularity*: This is the minimal granularity of place types: 'poi', 'neighborhood', 'city', 'admin' or 'country'. Default is 'neighborhood' limit: Number of places to return, returns up to 20 *proxies*: proxy object, ex. {'http': 'http://%s:%s' % (HOST, PORT), 'https': 'http://%s:%s' % (HOST, PORT)} *auth*: Twitter application authentication, see the get_authorization method Return: List of Twitter place dictionaries
lookup_place	Returns all the information about a known Twitter place, given its place ID. Twitter places are defined on the page https://dev.twitter.com/overview/api/places Parameters: *place_id:* Unique ID assigned to place by Twitter, can be looked up by reverse_geocode()

	proxies: proxy dictionary *auth*: Twitter application authentication, see the get_authorization method Return: Dictionary with place information
geocode_search	Search for places that can be attached to a statuses/update. Given a latitude and a longitude pair, an IP address, or a name, this request will return a list of all the valid places that can be used as the place_id when updating a status. Parameters: proxies: proxy dictionary auth: Twitter application authentication, see the get_authorization method lat: The latitude to search around, which must be inside the range -90.0 to +90.0 lon: The longitude to search around, which must be inside the range -180.0 to +180.0 q: Search term query, must be a string object or list of string objects exclusive: Boolean, if True, search with ORs rather than ANDs. Default is False ip: An IP address. Used when attempting to fix geolocation based off of the user's IP address. granularity: This is the minimal granularity of place types: 'poi', 'neighborhood', 'city', 'admin' or 'country'. Default is 'neighborhood' accuracy: The hint on the "region" in which to search, in meters, with a default value of 0m. If a number, then this is a radius in meters, but it can also take a string that is suffixed with ft to specify feet. max_results: A hint as to the number of results to return. This does not guarantee that the number of results returned will equal max_results, but instead informs how many 'nearby' results to return place_id: This is the place_id which you would like to restrict the search results to. Setting this value means only places within the given place_id will be found. Return: List of Twitter place dictionaries

Table 7: API Wrappers for searching for Twitter trends

Function	Description
find_trend_locations	Returns the locations that Twitter has trending topic information for. The response is an array of 'locations' that encode the location's WOEID and some other human-readable information. Parameters: *proxies*: proxy object, ex. {'http': 'http://%s:%s' % (HOST, PORT), 'https': 'http://%s:%s' % (HOST, PORT)} *auth*: Twitter application authentication, see the get_authorization method Return: List of Twitter place dictionaries
get_trends_for_place	Returns the top 50 trending topics for a specific WOEID, if trending information is available for it. The response is an array of 'trend' objects that encode the name of the trending topic, the query parameter that can be used to search for the topic on Twitter Search, and the Twitter Search URL. Use the function find_trend_places() to obtain a woeid. The information is cached for 5 minutes, and requesting more frequently than that will not return any more data. Parameters: *woeid*: The Yahoo! Where On Earth ID of the location to return trending information for. Global information is available by using 1 as the WOEID (default). *exclude*: Setting this equal to 'hashtags' will remove all hashtags from the trends list. *proxies*: proxy dictionary *auth*: Twitter application authentication, see the get_authorization method Return: List of Twitter topic dictionaries

Building a Network

This capability builds a data set based on an initial seed of users or topics. First a user loads a list of seed user names (Twitter @handles) into the sampling method. Then the sampling method *hops* to the next group of users based on a set of rules defined by the sampling parameters. There are two main categories of sampling Twitter graphs: breadth-first search and depth-first search. The breadth-first search function expands the graph based on profile and tweet information. There are two depth-first methods, cascade and causal sampling, that examine collection timelines with natural language processing techniques before selecting the seed of users to collect in the following hop. The depth-first methods require PostgreSQL to store and analyze data. See the section PostgreSQL Database Schema for information on the database schemas. However, the breadth-first search does not require PostgreSQL. It dumps profile and timeline JSON files in to specified directories. There are pyTweet extensions to load these JSON files into a PostgreSQL schema if the user desires.

As previously mentioned, sampling can take hours or days. Naturally you may not be able to run a script without interruption for such a long duration. To compensate for this pyTweet creates place saving files so that when sampling scripts are disrupted, they can pick up where they left off without any editing. This files, named *place_saver_v<{1,2}>.txt* and *growth_params_v<{1,2}>.txt* are stored in the same directory as the Twitter profile JSON files. Deleting them will cause the sampling to start from the beginning, or with the username seed.

Throughout the sampling loops, list specific to the hop iteration are saved. These lists contain user IDs collected on hop *k,* friends of these users, who they mention, and other similar information. The files created vary for each method, but all of them provide insight on how the Twitter graph is expanding. These files are saved in the Twitter profile JSON directory as well.

Breadth-first Search by Explicit Relationships

This capability builds a data set based on an initial seed of user screen names. After the names are read into the program, the sampling method them *hops* to the next group of users based on explicit relationships such as friendships, followers, mentions, or replies. The data collected is stored as one of two flavors of JSON files: user information JSON files and user timeline JSON files. The user information JSON files contain account information for a single user and are given the name **userInfo_*uuid*.json**. Similarly the timeline JSON files contain a timeline for a single user and are named **timeline_*uuid*.json**, where the UUID matches the one in **userInfo_*uuid*.json**. The directories to store profile and timeline JSON files must be specified before, otherwise default folders *profiles* and *timelines* are created in the current directory. For details about the metadata returned with the JSONs, see the Developer's site https://dev.twitter.com/overview/api/tweets and https://dev.twitter.com/overview/api/users. An example of the breadth-first search is implemented in the file *pyTweet/examples/breadth_first_search_example.py*.

While the breadth-first search completes iterations, it saves the following files as place savers, as well as means to analyze graph growth. These files are saved in the profile JSON directory, chosen by the user.

Table 8: Files generated by breadth-first search function

File Name	Description
place_saver_v<{1,2}>.txt	JSON file storing current and future hop collection IDs
Growth_paramsv<{1,2}>.txt	JSON file storing growth lists
h<hop *X*>_users.json	List of user IDs from hop *X*
h<hop X>_friends.json	User IDs of friends from hop X
h<hop X>_followers.json	Users IDs of followers from hop X
h<hop X>_user_mentions.json	User IDs of accounts mentioned by users from hop X
h<hop X>_replies.json	Users IDs of accounts replied to by users from hop X

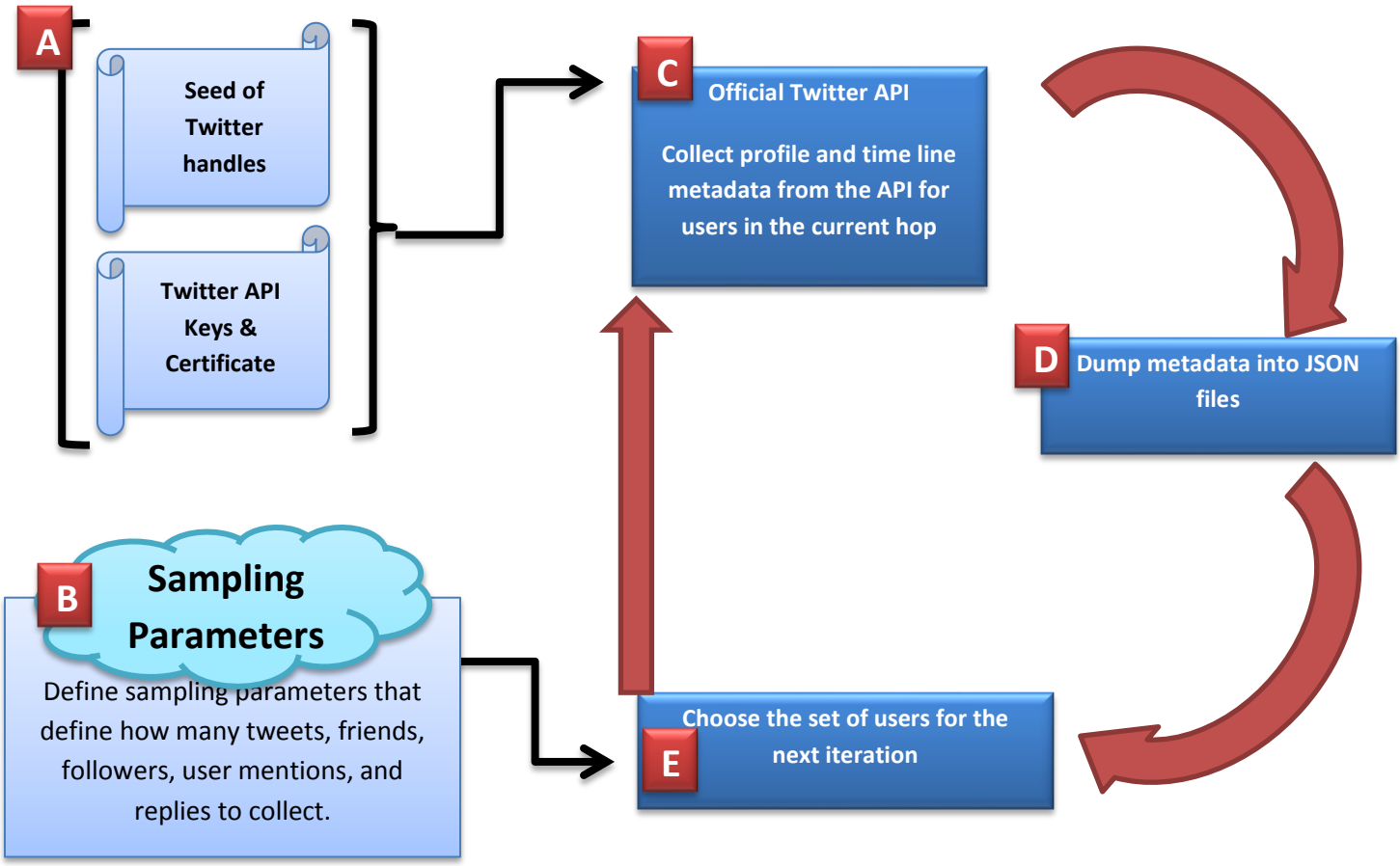
Note that sometimes the timeline JSON contain only the word 'null' if no timeline can be obtained. Keep the null timeline files so that you don't waste API calls looking for a timeline that may not exist. The existing timelines and user JSON files are routinely checked to avoid making redundant calls within the examples listed in the table. Also be aware that some attributes of timelines and profiles can change over time (i.e. retweet count, screenname). You may find it helpful to refer to the field *DOC*, which is the time the profile or timeline was collected.

The diagram in *Figure 2* illustrates the process of breadth-first sampling. Before sampling, the user must first complete **Steps A and B**:

A. Define the initial seed of Twitter handles and API keys. Refer to the *Getting Started* section to learn more.
B. Define the sampling parameters. There are three general objects used for sampling: the date timeline windows start, expansion limits. The expansion limits define how many friends, followers, replies, and user mentions to include in the next hop of users. They also define the stopping criteria of sampling: maximum number of hops and maximum amount of data (in GB). If a user doesn't want to impose a limit on one of these features they can simply define it as a *None* object.
C. Connect to the Official Twitter API and request data. We will receive mainly profile and timeline metadata.
D. Save the timeline and profile metadata as JSON files.
E. Based on the expansion limits, define the next set of users to collect metadata from. Redundant calls are avoided.

Steps C-E are iterative, and continue until the data or hop limit has been reached. Refer the file */pyTweet/examples/breadth_first_search_example.py* to see an implementation of this cod

Figure 2: Diagram of breadth-first search by explicit relationships



Depth-first Search with for Hashtag Cascades

This capability builds a data set based on an initial seed of user screen names. The graph is expanded based by user mentions in tweets, and friends and followers that use hashtags from the original seed of users. Users whose connections are expanded are limited to those who have less than one thousand friends and followers. As the profiles and tweets are collected, they are saved in a PostgreSQL database with the schema from section "PostgreSQL Database Schema." The depth-first cascade search also saves files that keep track of the iterations and growth of the sampling. These are saved in a directory selected by the user. Refer to the file *pyTweet/examples/cascade_search_example.py* for an example of implementing this search method.

Table 9: Files generated by depth-first cascade search function

File Name	Description
place_saver_v<{1,2}>.txt	JSON file storing current and future hop collection IDs
Growth_paramsv<{1,2}>.txt	JSON file storing growth lists
h<hop *X*>_users.json	List of user IDs from hop *X*
h<hop X>_missing.json	Users whose profiles should have been collected during hop X, but were unavailable for collection
h<hop X>_extendTRUE.json	Users whose user mentions and friends/followers will be collected

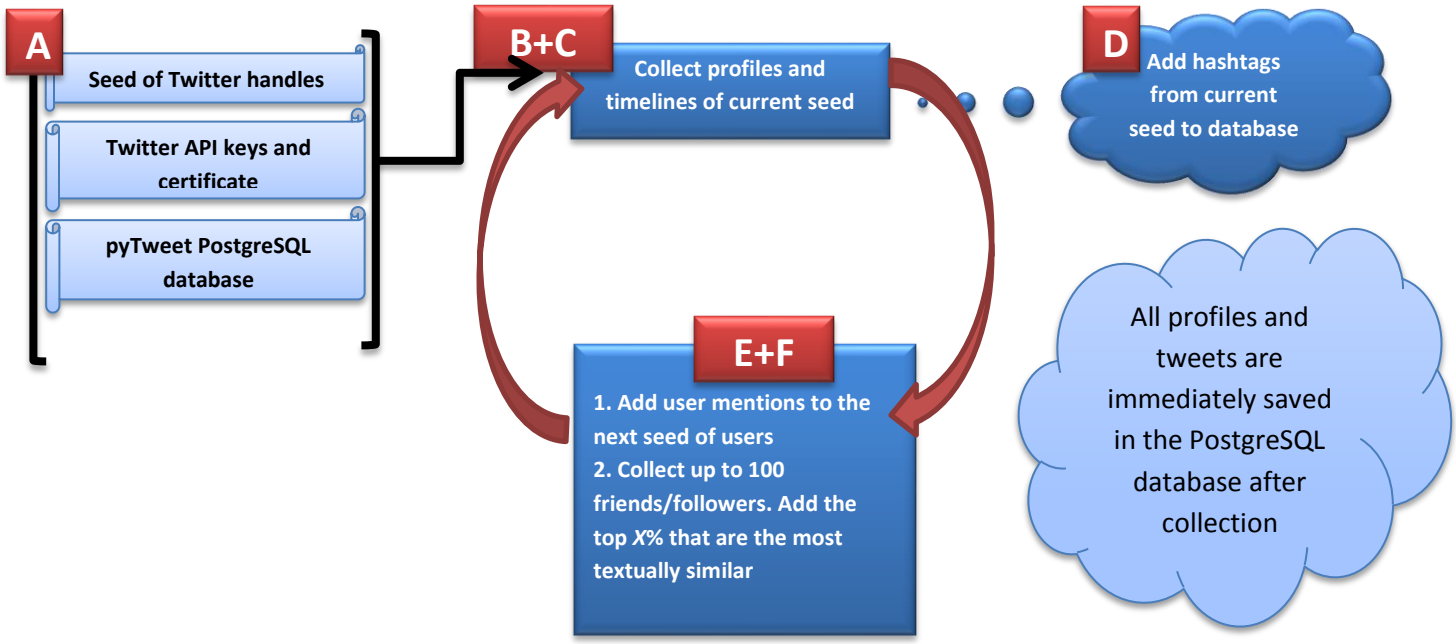
h<hop X>_extendFALSE.json	Users whose user mentions and friends/followers will not be collected
h<hop X>_extendNULL.json	Users whose user mentions and friends/followers will not be collected, but could be in another hop
h<hop X>_frfo_missing.json	Friends/Followers of the hop X seed whose profiles could not be collected
h<hop X>_frfo_extendTRUE.json	Friends/Followers of the hop X seed whose user mentions/friends/followers will be collected
h<hop X>_frfo_extendFALSE.json	Friends/Followers of the hop X seed whose user mentions/friends/followers will not be collected
h<hop X>_um_missing.json	User mentions of the hop X seed whose profiles could not be collected
h<hop X>_um_extendTRUE.json	User mentions of the hop X seed whose user mentions/friends/followers will be collected
h<hop X>_um_extendFALSE.json	User mentions of the hop X seed whose user mentions/friends/followers will not be collected
h<hop X>_um_extendNULL.json	User mentions of the hop X seed whose user mentions/friends/followers will be collected in the next hop, but could be in another
h<hop X>_relevant_extendTRUE.json	Friends/followers of the hop X seed who contained at least one common hashtag as the original seed, and whose user mentions/friends/followers will be collected
h<hop X>_relevant_extendNULL.json	Friends/followers of the hop X seed who contained at least one common hashtag as the original seed, and whose user mentions/friends/followers could be collected in another hop

The diagram in *Figure 3* illustrates the process of depth-first cascade sampling. Note that any user with more than one thousand friends and followers combined will not be expanded, AKA have their timelines, user mentions, friends, and followers collected.

A. Define the initial seed of Twitter handles and API keys. Refer to the *Getting Started* section for directions on this. Also be sure to create the PostgreSQL database that will store the data.
B. Collect profiles from the current seed, and eliminate some by the friend/follower rule
C. Collect timelines of current seed
D. If this is the first seed, add all hashtags to the database. From here on out collected friends' and followers' timelines will be compared to this list. Only those that have at least one hashtag in common will be expanded in the next hop.
E. Pull out user mentions from the current seeds' timelines. These users will be added to the next seed
F. Collect up to 1000 friends and 1000 followers from the current seed. Collect the profiles/timelines of these friends and followers. Only expand the ones that have at least one hashtag in common with the hashtag list. These friends/followers will be included in the next seed.

Steps B-F are iterative, and continue until the data or hop limit has been reached. Note that **Step D** is only implanted in the first iteration. Refer the file *pyTweet/examples/cascade_search_example.py* to see an implementation of this algorithm

Figure 3: Diagram of depth-first hashtag cascade search. Note that no timelines, user mentions, friends or followers are collected for users whose combined friend and follower count exceeds one thousand.



Depth-first Search for Causal Inference

This capability builds a data set based on an initial seed of user screen names. The graph is expanded based by user mentions in tweets, and the top friends and followers that are textually similar to the current hop. As the profiles and tweets are collected, they are saved in a PostgreSQL database with the schema from section "PostgreSQL Database Schema." The depth-first causal search also saves files that keep track of the iterations and growth of the sampling. These are saved in a directory selected by the user. Refer to the file *pyTweet/examples/causal_search_example.py* for an example of implementing this search method.

Table 10: Files generated by depth-first causal search function

File Name	Description
place_saver_v<{1,2}>.txt	JSON file storing current and future hop collection IDs
Growth_paramsv<{1,2}>.txt	JSON file storing growth lists
h<hop *X*>_users.json	List of user IDs from hop *X*
h<hop X>_friends.json	User IDs of friends from hop X
h<hop X>_followers.json	Users IDs of followers from hop X
h<hop X>_user_mentions.json	User IDs of accounts mentioned by users from hop X

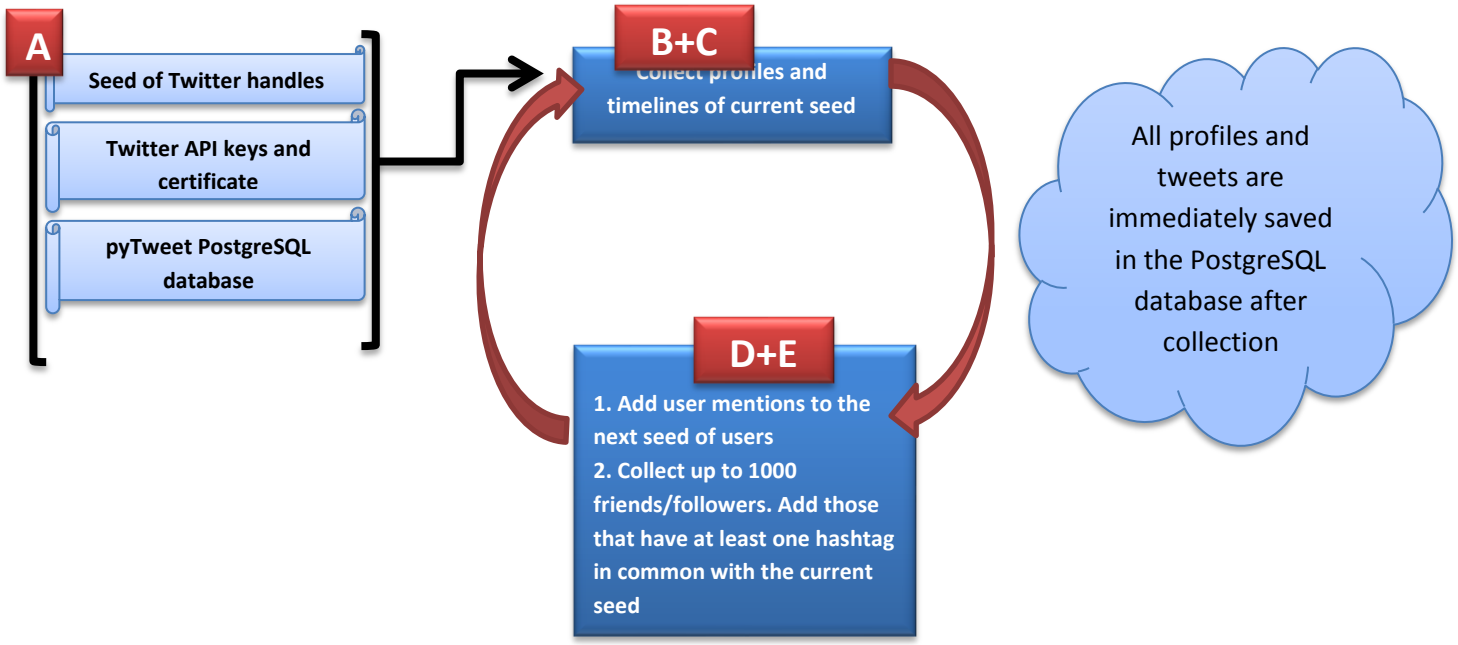
The diagram in *Figure 4* illustrates the process of depth-first cascade sampling. Note that to expand a user is to collect their user mentions, friends and followers.

A. Define the initial seed of Twitter handles and API keys. Refer to the *Getting Started* section for directions on this. Also be sure to create the PostgreSQL database that will store the data.
B. Collect profiles from the current seed
C. Collect timelines of current seed

D. Add user mentions from the current seed's timelines to the next seed
E. Collect up to 100 friends and 100 followers of the current seed. Calculate the CANDID TF-IDF similarity, and expand the top *X* percentage of them

Steps B-E are iterative, and continue until the data or hop limit has been reached. Refer the file *pyTweet/examples/causal_search_example.py* for an implementation of this algorithm

Figure 4: Diagram of depth-first hashtag causal search. Note that only uses with tweets occurring between the user specified dates can be expanded.



PostgreSQL Database Schema

Profile and timeline metadata can easily be formatted into a PostgreSQL database that resembles a graph. Tables are created for user profiles, tweets, and edges between users.

Load JSON files into pyTweet PostgreSQL database schema

The JSON files collected from the Twitter API can be directly converted into a PostgreSQL database with pyTweet's json_to_database submodule. This database's schema is very similar to a standard graph where there are vertex and edge-like tables. Tables exist for profiles, tweets and edges. Creating edges is explained in the following section. An example for loading JSON files into the profile and tweets tables can be found in the file *pyTweet/examples/populate_database_example.py*

Create edges in pyTweet database schema

In addition to populating profile and tweets tables, the submodule json_to_database offers a capability to populate and edge table. Overall, there are seven edge creation options through this package based on friendships, followers, and tweets. Edge types are assigned numbered values in the function, which are explained in Table 6. Additionally, the function load_hashtag_edges creates edges between users based on hashtag use. You can find an implementation of creating all of these edges in the file *pyTweet/examples/populate_database_example.py.*

Table 11: pyTweet edge options

Edge Type	Description
Friendship	*The user from_id and user to_id* follow each other. It's a 2-way relationship
Followers	*The user to_id follows user from_id. It's a 1-way relationship*
User mentions	The user *from_id* mentioned user *to_id* in the tweet *tweet_id*.
Replies	The user *from_id* replied to the tweet *tweet_id* that user *to_id* tweeted.
Co-mention	Co-occurrence edge. The users *from_id* and *to_id* are both mentioned in tweet *tweet_id*
Co-mention-reply	Co-occurrence edge. The user *from_id* is replied to the *tweet_id* that user *to_id* is mentioned
Hashtag	Links uses who use a specific hashtag

Recommendations for storing date of collection

When working with Twitter data, you should realize that some metadata is very likely to change after you collect it. Friends and followers of a user, favorites, and retweets are just some fields of metadata susceptible to change. Therefore you may want to save the date of collection with every profile and tweet. You can do this by adding a key to the profile or tweet dictionary called 'DOC' and setting it to the current time using a datetime object. For example,

```
profile_metadata['DOC'] = datetime.datetime.utcnow()
tweet_metadata['DOC'] = datetime.datetime.utcnow()
```

The breadth-first and depth-first sampling methods in pyTweet will save the date of collection automatically, and nothing needs to be done on the user's end to ensure that date of collection is saved. However, if you make your own JSON objects then the date of collection is not automatically stored. These objects will still be compatible, but the date of collection will be assumed to be the last time the JSON file was modified.

Database Schema

Table 12: Description of Users table in schema that contains metadata of profiles

Column	Data Type	Description
user_id	bigint	Unique identifying integer for user profiles
user_name	text	The name of a user, as they've defined it, but not necessarily a person's name. It's typically a maximum of 15 characters long, but some older accounts may have longer names.
screen_name	text	The screen name, handle, or alias that this user identifies themselves with. Screen_names are unique but subject to change. Use user_id as a user identifier whenever possible.
location	text	User's default location or hometown
friends_list	bigint[]	List of friends' profile IDs. Note that the count of friends in this list may differ from the actual friends_count if a limit was imposed on friend collection.
followers_list	bigint[]	List of followers' profile IDs. Note that the count of followers in this list may differ from the actual followers_count if a limit was imposed on the follower collection.
profile_background_image_url	text	Link to user's background profile image
profile_image_url	text	Link to user's profile image
profile_url	text	Link to user's Twitter profile
time_zone	text	Time zone string of user's location
date_of_collection	timestamp with time zone	Date when profile was collected
khop	integer	Hop number in collection
geo_enabled	boolean	TRUE if user has enabled geotagging for tweets
profile_langauge	character varying(10)	User's default language
friends_count	bigint	The number of friends of the user. Note that this count may differ from the friends_list if a limit was imposed on friend collection.
followers_count	bigint	The number of followers of the user. Note that this count may differ from the followers_list if a limit was imposed on follower collection.
has_timeline	boolean	Indicates whether the user's timeline has been collected. This column is present in cascade and causal sampling only.
expand_user	boolean	Indicates whether or not to expand a user. This column is present in cascade and causal sampling only.
timeline_is_relevant	boolean	Indicates whether or not the user's timeline contains at least one tweet using a phrase from the topics table. This column is present in cascade and causal sampling only.
has_timeline_filter	boolean	Indicates whether or not to expand a user. This column is present in cascade and causal sampling only.
decision_tfidf	double precision	The value of a user's TF-IDF value at the time of decision. This column is present in cascade sampling only.
timeline_document	text[]	Tokenization of user's timeline. This column is present in causal sampling only.
decision_candid_tfdf_score	double precision	TF-DF score computed for user's timeline. This column is present in causal sampling only.

Table 13: Description of Tweets table in schema that contains metadata of tweets

Column	Data Type	Description
tweet_id	bigint	Unique identifying integer for a tweet
user_id	bigint	Unique user ID for the author of the tweet
created_at	timestamp with time zone	Time of publication in UTC time
tweet	text	Tweet text
user_mentions	bigint[]	List of user ID's for users mentioned within the tweet
hashtag_entities	text[]	List of hashtags used in the tweet
url_entities	text[]	List of URLs mentioned in the tweet
in_reply_to_status_id	bigint	When the tweet is a reply to another tweet, this is the tweet ID of the tweet that is replied to
in_reply_to_user_id	bigint	When the tweet is a reply to another tweet, this is the user ID of the author who's tweet is replied to
latitude	double precision	Latitude coordinate of the tweet
longitude	double precision	Longitude coordinate of the tweet
retweet_count	integer	Number of times this Tweet has been retweeted. This field is no longer capped at 99 (since 01/31/2012) and will **not** turn into a String for "100+"
country	text	Tweet's country of origin
place_full_name	text	Name of tweet's origin location
place_type	text	Description of place the tweet originated from
place_url	text	If the place has a twitter profile, this is a link to it
favorite_count	bigint	The number of times the tweet has been favored.
date_of_collection	timestamp with time zone	Date when the timeline was collected.

Table 14: Description of Tweets table in schema that contains metadata of tweets

Column	Data Type	Description
edge_id	integer	Unique integer for an edge
from_id	bigint	Unique user ID of the user who the edge originates from
to_id	bigint	Unique user ID of the user who the edge ends with
edge_type	smallint	This is an integer {1,2,3,4,5,6,7} that represents the edge type: 1. *The user from_id and user to_id* follow each other. It's a 2-way relationship 2. *The user to_id follows user from_id. It's a 1-way relationship* 3. The user *from_id* mentioned user *to_id* in the tweet *tweet_id*. 4. The user *from_id* replied to the tweet *tweet_id* that user *to_id* tweeted. 5. Co-occurrence edge. The users *from_id* and *to_id* are both mentioned in tweet *tweet_id* 6. Co-occurrence edge. The user *from_id* is replied to the *tweet_id* that user *to_id* is mentioned in. 7. Hashtag edge. Both the user *from_id* and *to_id* used the hashtag in the *hashtag* column
tweet_id	bigint	If the edge comes from either a tweet reply (edge type 4) or user mention (edge type 3), this is the ID of the corresponding tweet
hashtag	text	If the edge comes from a shared hashtag (edge type ???), this is the hashtag

Table 15: Topics table used with causal sampling method

Column	Data Type	Description
topic_id	integer	Unique topic ID
topic	text	Phrase, URL, or hashtag
khop	integer	Indicates which sampling hop the phrase came from. A value of -1 indicates that the topic was added to the table before sampling began.
document_frequency	double precision	Found in causal sampling.

Table 16: Unavailable profiles table: This is a table of unavailable profiles because they are either private or have been deleted. This table is used in causal and cascade sampling.

Column	Data Type	Description
profile_id	integer	Unique identifier for users in this table
screen_name	text	Screen name of an unavailable profile
user_id	bigint	User ID of an unavailable profile

References

[1] "Anaconda." *Scientific Python Distribution*. Web. 26 June 2015.

[2] "Twitter Developers." *Twitter Developers*. Web. 26 June 2015.