Documentation of User Location Dataset (ULDS) Code

Christopher Marks

June 7, 2017

1 Overview

This paper serves as abbreviated documentation of the Python code used to generate a set of Twitter users in a specific location. This code is an implementation of the user location dataset collection process detailed in Chapter 4 of Marks [2017].

The code consists of two python scripts:

- `get_ULDS.py`. This file executes the expand—classify method described in Marks [2017], Chapter 4. Data is saved in an `SQLite` database file.
- `get_tweets.py`. This file collects all available tweets from the users identified in the target location by the expand—classify method. These tweets are appended in additional tables in the `SQLite` database file.

Requirements

Execution of these scripts requires the following:

- Python 2.7 or higher, and the following modules.
  - `os, sys, datetime, sqlite3` and `time` (included with most installations).
  - `matplotlib`.
  - `sklearn`.
  - `numpy`.
  - `gmplot` for plotting maps & geocoding location.
  - `random`.

  - `networkx`.
  - `twython`.

- Twitter account. Sign up at Twitter.com
- Twitter API authentication credentials. The code requires the user to initialize an authenticated `twython.Twython` object. Note that the `oauth1setup.py` script available at https://zlisto.scripts.mit.edu/informs_tutorial/oauth1setup.py will create these credentials for an authenticated Twitter user and write a Python script to import the authenticated `twython.Twython` object.

2 get_ULDS.py

This script executes the expand—classify method described in Marks [2017], Chapter 4.

2.1 Required User Inputs

- `twitter`. An authenticated `twython.Twython` object.
- `runtime_hours`. The number of hours to run the expand—classify iterations.
- `db_name`. The name of the database file. The "`.db`" extension will be appended automatically. **This should be the name of the population center**.
- `geocode_name`. The name of the target location. The `gmplot` module will retrieve latitude-longitude coordinates from the Google Maps API for this location.
- `label_radius`. The radius around the latitude-longitude coordinates that identifies the target location area. Users known to be inside this radius through geo-coded Tweets will be labeled as being in the target location.
- `working_dir`. The path to the directory where the `SQLite` database file will be created and the plots and other output will be saved. This directory must exists; the Python script will not create it.
- `seed_user_ids`. A list of seed users presumed to be in the target location. Because the algorithm collects friends and followers of seed users, selecting users that appear to be locally well-connected will improve performance. Note that a short, commented script is included that will user the Twitter user search API to produce a seed user set, but for smaller locations this method is unreliable.
- `location_terms`. A set of terms or phrases that will serve as useful features in classifying a user as in or out of the target location. For example, if the target location is Boston, MA, then this list can include `''Boston, MA''`, `''Boston''`, `''Cambridge''`, etc.

- `languages`. A list of two-letter language abbreviations associated with the target location. If the target location is Boston, MA, this list should contain the string `''en''`.
- `utc_offsets`. A list of time zones, encoded as integer numbers of seconds offset from Coordinated Universal Time (UTC), that can be used to associate a user account with the target location. Typically this list will contain one or two time zones.

2.2 Other Parameters & Methods

The code will typically execute if the user provides a reasonable set of inputs as described above. This section will briefly describe the remainder of the code.

2.2.1 User Profile Probability Model

The classify method requires a method or set of methods that inputs a social media profile object and returns a log-odds ratio. The current implementation extracts features from the social media profile based on the location terms provided by the user *and* based on a subset of cities from the world cities database [MaxMind.com, 2017]. In order for the features derived from the world cities to be of value in the classification, this list and the set of location terms should be mutually exclusive. Currently, the script removes `db_name` from the world cities list if it is present. For example, if the target location is Los Angeles, CA and `db_name` is set to "Los Angeles", then the string "Los Angeles" will be removed from the world cities list. The remaining cities are stored in the variable `outside_location_terms`.

The `features` method extracts features from profile and also returns a list of feature types. The list of feature types is useful for creating the `SQLite` tables.

The `X` method inputs a user profile and returns a vector of features.

The `fit_LR` method inputs a UserData object (described below) and label radius. This method fits a logistic regression model, plots the ROC, and returns a function that takes a vector of features and returns the log-odds ratio based on the logistic regression fit. The output of this method serves as the user profile energy function, ϕ, as described in Marks [2017].

2.2.2 Link probability model

This section of the code defines a method, ψ, as described in Marks [2017]. The parameters can by adjusted as desired by the user, even after the collection is completed in order to check for sensitivity. The table below gives the parameter names from Marks [2017] and the corresponding names in the file `get_ULDS.py`.

Marks [2017]	`get_ULDS.py`
λ	`alambda`
γ	`alpha[0]`
α_1	`alpha[1]`
α_2	`alpha[2]`

2.2.3 User location data class

This section defines an object class that contains a database connection and a set of methods used to execute the expand—classify method. The following list is not exhaustive, but describes some of the methods and parameters that might be of interest:

- `lldist`. This method returns the distance in miles between two points given in lattitude-longitude.
- `create_tables`. This method creates the database structure used for collection, if it doesn't exist already in the tables. If the tables already exist, they are not overwritten. **This could be a problem if the features in the user profile probability model have changed.**
- `add_users`. This method takes a set of profiles and inserts them into the database.
- `add_geo_user`. This method inputs a user ID and an authenticated `twython.Twython` object. It then searches the identified social media user's most recent 200 Tweets for geo-located content. If it finds geo-located content, it records the location of the geo-located content that is closest to the latitude-longitude obtained for the target location.
- `expand_step`. This method inputs an authenticated `twython.Twython` object and a profile features function and executes the *expand* step in the expand—classify method. The optional `sample_size` parameter indicates how many queries will be executed for each of the four API methods employed. The default `sample_size` is 15, which is the maximum number of queries per 15-minute window allowed by the Twitter APIs. The four API methods used are
  - `GET friends/list` (up to 200 results).
  - `GET followers/list` (up to 200 results).
  - `GET friends/ids` (up to 5000 results).
  - `GET followers/ids` (up to 5000 results).

Users are selected randomly for these methods, but the "list" queries are run on users with not more than 200 relationships of the type being queried.

- `collect_geo_users`. This method inputs an authenticated `twython.Twython` object and some optional stopping criteria. The method iteratively calls the `add_geo_user` method for social media users who have geo-location enabled in their profiles, populating the set of users in the database with known locations.

- `classify_step`. This method inputs the two energy functions described in Marks [2017] and executes the *classify* step of the expand—classify method. The method uses Python's `networkx` module to build the energy graph and find the minimum cut. See Marks [2017], Chapter 4 for details. Label classifications are updated in the database. Additionally, social media users with low link energy totals (summed over all their neighbors) are "pruned" from future classification steps because we can assume their classifications rely solely on the user profile energy ϕ values and not on the links. This pruning helps future iterations of the classify step run faster.

- `update_phi_Optim0`. This method inputs a profile energy function and uses it to update the profile energy for all pruned users in the database.

- `make_map_ROC`. This method inputs a filename string and a radius and generates a color-coded HTML map of the geo-located users, using the Python `gmplot` module and the Google Maps API. The method also produces an ROC curve based on the output of the classification model. A description of how this plot is generated is given in Marks [2017], Chapter 4. **Note: these plots include data coded as "test" data only, i.e., data that is not used in training or validating the logistic regression model.**

2.3 Execution Script

2.3.1 Preparation

The script

- Changes into the working directory.

- Calls the `features` method on a test profile.

- Establishes the database connection.

- Gets the geocode latitude-longitude for the target location using `gmplot` to access the Google Maps API.

- Initializes the `UserData` object with the database connection and the target geocode location.

- Creates the database tables.
- Adds the seed users to the database tables.

2.3.2 First Iteration

- Executes the *expand* step.
- Collects geolocation information on geo-enabled users until either
  - 500 geo-located users are identified, OR
  - All geo-enabled users' timelines have been queried.

  Note: it can take a long time to find 500 users with geo-location information. However, this data is very important as it provides a labeled dataset which we will use in the logistic regression model.
- Fits a logistic regression model predicting whether or not a geo-located users is in the target location based on the profile features. Function `phi` applies this model on a set of features to obtain a log odds ratio.
- Executes the *classify* step using the `phi` function from the previous step and the fixed `psi` function.
- Creates and saves a geo-located user classification map and an ROC plot for iteration 1.
- Collects geolocation information on geo-enabled users until the Twitter API rate limit reset time (15 minutes) has passed or there are no unqueried geo-enabled users.
- Re-fits the logistic regression model and updates `phi` based on the results of additional geo-located data.
- Updates the values for `phi` in the database for users that were "pruned" in the previous classify step.
- Waits for rate limit reset, if necessary.

2.3.3 Follow-on Iterations

The following loop iterates until `runtime_hours` hours have elapsed since the start of the collection.

- Executes the *expand* step.
- Executes the *classify* step.

- Creates and saves a geo-located user classification map and an ROC plot for iteration 1.
- Collects geolocation information on geo-enabled users until the Twitter API rate limit reset time (15 minutes) has passed or there are no unqueried geo-enabled users.
- Re-fits the logistic regression model and updates `phi` based on the results of additional geo-located data.
- Updates the values for `phi` in the database for users that were "pruned" in the previous classify step.
- Waits for rate limit reset, if necessary.

2.3.4 Post-collection

After the total collection time has elapsed, no more new users are collected. However, the following steps are executed to "complete the record."

- Gather geo-location data for all remaining unqueried geo-enabled users. **Note: this often takes several days to complete.**
- Re-fit the logistic regression model with the updated geo-location data.
- Update values for `phi` in the database for users that have been "pruned" from the classification steps.
- Execute the classify step with the new user profile probability model.
- Make final HTML map and ROC plot and save.

2.4 `SQLite` Data Model

The `SQLite` database created and used in this script contains six tables:

- `collections`. The purpose of this table is to keep summary statistics on each iteration of the expand—classify method. It has the following fields.
  - `collection_no`. The iteration number. The value 0 corresponds to the insertion of the seed users into the database.
  - `start_collect`. The the date and time the expand step started.
  - `end_collect`. The date and time the expand step ended.

  - `pos_label_count`. The number of users classified in the target location in this iteration.
  - `neg_label_count`. The number of users classified out of the target location in this iteration.

- `user`. This table contains pertinent profile and classification information for each user collected in the expand steps. In contains the following fields.
  - `user_id`. The user's Twitter user ID.
  - `friends_count`.
  - `followers_count`.
  - `protected`. Takes value 1 if the user account is protected, otherwise 0.
  - `verified`. Takes value 1 if the user account is verified by Twitter, otherwise 0.
  - `geo_enabled`. Takes value 1 if the user account is geo-enabled, otherwise 0.
  - `statuses_count`. The number of tweets posted by the user account.
  - `tweet_rate`. The number of tweets divided by the age of the account.
  - `collection_no`. The iteration of the expand step in which the user was first collected. For seed users, this is set to 0. For users found in the first iteration of the expand step, this value is set to 1, etc. This field references `collections.collection_no`.
  - `queried_friends`. Set to `''Y''` if the user's friends have been queried in an expand step, `''N''` if the user's friends have not been queried in an expand step, or `''P''` if some of the users friends have been queried, but some have not.
  - `queried_followers`. Same as `queried_friends` but for followers.
  - `friend_query_page`. For user's that have had some, but not all of their friends queried (i.e., `queried_friends = ''P''`), the page cursor for the next friend query.
  - `follower_query_page`. Same as `friend_query_page`, but for followers.
  - `Label`. Takes value 1 if the user is classified as being in the target location, otherwise 0.
  - `Prob`. The local classification probability (see Marks [2017]).
  - `psi_local`. The sum of the link energy terms in the exponent in the local classification probability (see Marks [2017]).
  - `phi_local`. The sum of the profile energy terms, including adjustments for $\psi(\mathbf{z}_{ij}, 0, 0)$, in the exponent in the local classification probability (see Marks [2017]).
  - `A00_contrib`. The sum of the profile energy adjustments for $\psi(\mathbf{z}_{ij}, 0, 0)$ in the exponent in the local classification probability (see Marks [2017]). Note that adding this to `phi_local` should recover profile energy function `phi`.

  - `Res`. Residual flow from the max flow—min cut algorithm. This field is not used in the current implementation.
  - `Optim`. Takes value 0 if this user has been pruned in the classification step because of the lack of substantial link energy. Otherwise, set to 1.
  - `ML_set`. Set to `''TRN''` if user is assigned to training set, `''VAL''` if assigned to the validation set, and `''TST''` if assigned to the test set. Currently user's are randomly assigned when they are collected in the expand step: 50% training, 20% validation, and 30% test.

- `link`. This table contains data on all collected social network connections. It has the following fields.
  - `left_user_id`. The lesser of the two Twitter user IDs that constitute the social media relationship. This field references `user.user_id`.
  - `right_user_id`. The greater of the two Twitter user IDs that constitute the social media relationship. This field references `user.user_id`.
  - `left_follows_right`. Takes value 1 if the account identified by `left_user_id` is following the account identified by `right_user_id`.
  - `right_follows_left`. Takes value 1 if the account identified by `right_user_id` is following the account identified by `left_user_id`.

- `geo_user`. This table contains geo-location data for geo-enabled users. It has the following fields.
  - `user_id`. The user ID of the account. This field references `user.user_id`.
  - `geo_tweet`. This field takes value 1 if a geo-located tweet was found in the user's most recent 200 tweets. Otherwise, it takes value 0.
  - `tweet_id`. This field contains the tweet ID for the geo-located tweet posted by the user that is closest to the target location geo-coordinates.
  - `lat`. The latitude of the closest geo-located tweet.
  - `lon`. The longitude of the closest geo-located tweet.
  - `dist`. The distance from the closest geo-located tweet to the target location geo-coordinates.
  - `name`. The geo-user's Twitter name (only inserted if a geo-located tweet was found).
  - `screen_name`. The geo-users' Twitter screen name (only inserted if a geo-located tweet was found).
  - `location`. The geo-users' declared location (only inserted if a geo-located tweet was found).

- `features`. This table contains the user profile features that serve as the input vector for the phi function. It has a `user_id` field that references `user.user_id` and a field for each feature in the probability model.

- `collection_queries`. This table keeps track of which users' friends and followers are queried in each expand step. It has the following fields.
  - `collection_no`. The expand step iteration number. This field references `collections.collect`
  - `user_id`. The user's Twitter user ID. This field references `user.user_id`.
  - `query`. The query type. This field is set to ``` ``friend'' ``` (or ``` ``follower'' ```) if the user `user_id`'s friends (or followers) were queried in iteration `collection_no`.

3 `get_tweets.py`

The `get_tweets.py` script collects and stores the tweets belonging to the users classified in a target location using the expand—classify approach.

3.1 Required User Inputs

- `twitter`. An authenticated `twython.Twython` object.
- `db_name`. The name of the database file. In contrast to the `get_ULDS.py` script, this string should include the ``` ``.db'' ``` file extension. The database should already exist and contain a set of location-classified users.
- `max_queries_per_user`. The maximum number of timeline queries to execute on each location user. Each query returns up to 200 tweets. Twitter limits results to a user's most recent 3200 tweets, so this value should not be set to anything higher than 16.
- `working_dir`. The path to the directory where the `SQLite` database file is found.

3.2 Other Parameters & Methods

This script defines a `TweetData` class that maintains a connection to the `SQLite` database and contains methods for creating tables, and collecting and inserting tweets. The methods are fairly straightforward SQL queries.

3.3 Execution Script

The script

- Changes into the working directory.

- Establishes the database connection.
- Creates tables for storing tweet data **in the existing database**.
- Gets a list of users that have been classified in the target location.
- Gets a list of users that have already had their timelines queried (if any).
- Uses set difference to obtain a list of location users that have not had their timelines queried (`remaining_users`).
- Randomizes the order of `remaining_users`.
- For each user in `remaining_users`,
  - Obtains the full user profile through the Twitter API and inserts it into the `user_profile` table.
  - If the user has posted tweets and the profile is not protected,
    - Iteratively query the user's twitter timeline until either a query returns no results or the `max_queries_per_user` is reached.
- Disconnects from the database.

3.4 `SQLite` Data Model

The `TweetData` class creates additional tables in the `SQLite` database created and populated by the `UserData` class. These tables closely follow the Twitter data model contained in the slides appended to this documentation. However, the `user` table from the Twitter data model depicted in the slides is replaced by the the `user` table created in the expand—classify method.

In addition to the tables described in the slides, the `TweetData` class creates and maintains a `collected` table, which tracks which users' timelines have been collected. The `collected` table consists of the following fields.

- `user_id`. The Twitter user ID. This field references `user.user_id`.
- `date_collected`. The date the user's timeline was collected.
- `first_tweet`. The earliest tweet collected from the user. This field references `tweet.tweet_id`.
- `last_tweet`. The latest tweet collected from the user. This field references `tweet.tweet_id`.

References

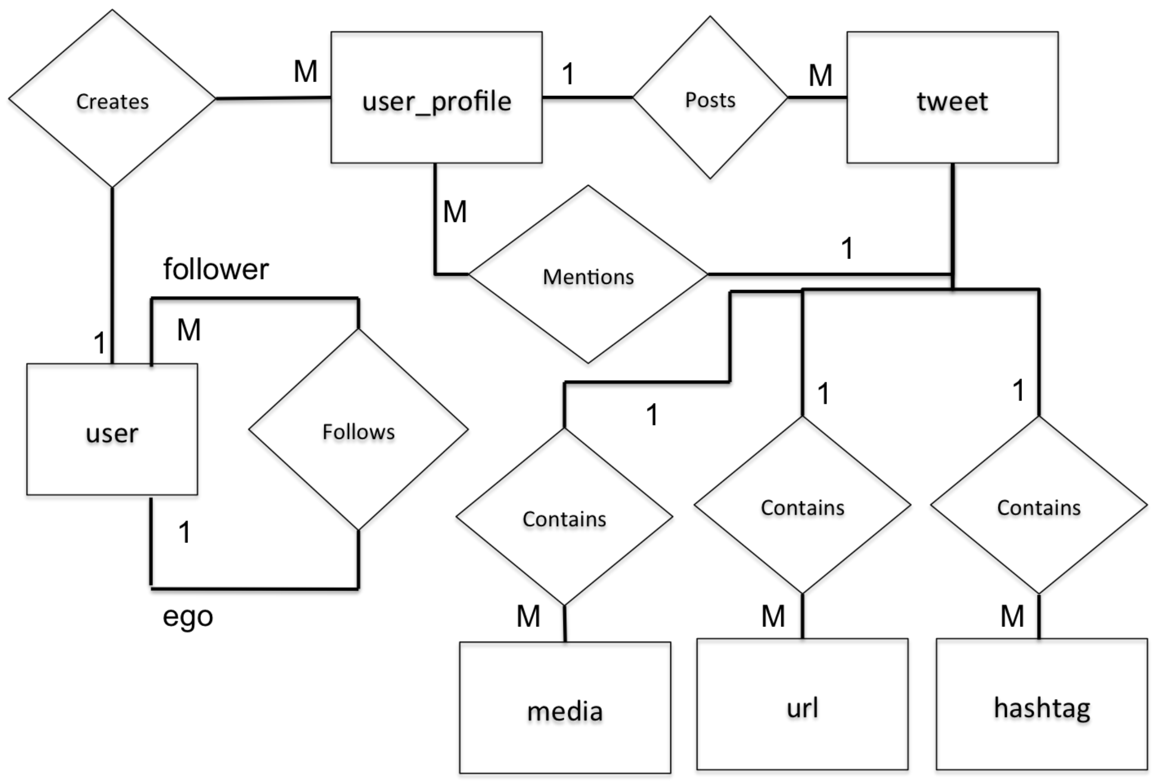
Christopher Marks. *Analytic search methods in online social networks.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, June 2017.

MaxMind.com. World cities database, 2017. URL `http://www.maxmind.com/`.

A Twitter Data Model Slides

The slides on the following pages explain the Twitter data model that is employed by the TwitterData class.

Relational Data Model



Relational Data Model Implementation

- Online SQL database with eight tables.
  - `user`.
  - `user_profile`.
  - `tweet`.
  - `tweet_media`.
  - `followers`.
  - `tweet_usermentions`.
  - `tweet_hashtags`.
  - `tweet_url`.

Attributes for **user** Table

- `user_ID`. A unique ID number assigned to each user that also serves as the primary key for this table.
- `name`. The name of the user associated with the account. This column is set to `NULL` by default, but we can use it if we can positively identify a user with an account.
- `location`. The location of the user associated with the account. This column is set to `NULL` by default, but we can use it if we know the user's location.

Attributes for **user_profile** Table (1 of 3)

- `user_ID`. A unique ID number assigned to each user in Twitter. This value is constrained to be a `user_ID` entity from the**user** Table.
- `screen_name`. A unique screen name selected by the user for the account.
- `name`. The account user's name, as inputted by the user. Unlike the `screen_name`, this value does not have to be uniquely identify the user.
- `created_at`. The date and time the user account was created.
- `description`. A free text self-description inputted by the user.
- `status_id`. The `tweet_id` identifying the user's current tweet status. If the tweet is available, a corresponding entity will be in the **tweet** entity set. However, this value is *not* constrained to be a `tweet_id` from the **tweet** entity set.
- `geo_enabled`. A binary (true/false) variable indicating whether the account has geo-location of content posts enabled.
- `protected`. A binary (true/false) variable indicating whether the media attributed to the account is publicly available, as determined by the account's privacy settings.
- `friends_count`. The number of other users the account is following.

Attributes for **user_profile** Table (2 of 3)

- `followers_count`. The number of other users following the account.
- `statuses_count`. The number of social media posts attributed to the account.
- `lang`. The account language.
- `location`. The location registered to the account.
- `verified`. A binary (true/false) variable indicating whether the user identification information registered to the account has been verified by the social media provider.
- `profile_url`. An external url registered to the account, inputted by the user. These external urls often point to accounts in different social media applications.
- `default_image`. A binary (true/false) variable indicating whether the user is using the default profile image, or has uploaded an image.
- `time_zone`. The time zone location registered with the account.

Attributes for **user_profile** Table (3 of 3)

- `UTC_offset`. An alternative specification of the account time zone as the number of minutes ahead of Greenwich Mean Time. Time zones in the western hemisphere are behind Greenwich Mean Time and therefore have negative `UTC_offset` values.
- `profile_pic_hash`. An eight-digit hexadecimal average hash generated by the user's uploaded profile picture.
- `profile_pic_filename`. The filename where the profile picture has been stored, if downloaded.
- `profile_banner_hash`. An eight-digit hexadecimal average hash generated by the user's uploaded profile banner picture.
- `profile_banner_filename`. The filename where the banner image has been stored, if downloaded.
- `obtained`. The currency date of the account profile information. This is the date the profile information was recorded.
- `last_accessed`. The last date the profile was checked for activity, but not necessarily recorded.
- `active`. A binary (true/false) variable indicating whether the account is currently active.

Attributes for **tweet** Table (1 of 2)

- `tweet_id`. A unique ID number that also serves as the primary key.
- `user_id`. The `user_id` for the **user_profile** entity that posted the content. This variable is constrained to come from **user_profile** entity set.
- `screen_name`. The `screen_name` for the **user_profile** entity that posted the content, also constrained to come from the **user_profile** entity set.
- `created_at`. The date and time the content was posted.
- `text`. The text posted by the user. Twitter generally restricts this field to 140 characters.
- `geo_lat`. The latitude (if provided) of the user's location when the content was posted.
- `geo_long`. The longitude (if provided) of the user's location when the content was posted.
- `place_type`. The type of "place" (if provided) used to describe the user's location when the content was uploaded, e.g., "region," or "city."
- `place_name`. The name of the "place" (if provided) used to describe the user's location when the content was uploaded, e.g., "Sidney, Australia."

Attributes for **tweet** Table (2 of 2)

- `lang`. The language registered to the tweet.
- `source`. The application employed by the user to post the tweet.
- `retweet_count`. The number of times the tweet has been retweeted.
- `retweet_status_id`. The `tweet_id` of the original content, if the user identifies the content as a re-post of another user's content. This variable is constrained to be a `tweet_id` from the **tweet** entity set.
- `reply_to_status_id`. The `tweet_id` of the original content that the post is "in reply to." This variable is *not* constrained to reference another `tweet_id` in the **tweet** entity set.
- `reply_to_user_id`. The `user_id` associated with the original content that the post is "in reply to." This variable is *not* constrained to reference a `user_id` in the **user** or **user_profile** entity sets.
- `reply_to_screen_name`. The `screen_name` associated with the original content that the post is "in reply to." This variable is *not* constrained to reference a `screen_name` in the **user_profile** entity set.

Attributes for **tweet_media** Table (1 of 2)

- `tweet_id`. The `tweet_id` referencing the **tweet** entity containing the image. This variable is constrained to be a `tweet_id` in the **tweet** entity set.
- `pic_hash`. An eight-digit hexadecimal average hash generated by the uploaded image.
- `pic_id`. A unique media identification number assigned to the uploaded image by the social media application. If different users independently upload the same image, different numbers will be assigned. However, if a user reposts an image uploaded by another user, the `pic_id` remains the same.
- `pic_source_user_id`. The `user_id` for the user that originally uploaded the image. This variable is *not* constrained to be a value in the **user** or **user_profile** entity set.

Attributes for **tweet_media** Table (2 of 2)

- `pic_source_status_id`. The `tweet_id` for the content post that originally contained the image. This variable is *not* constrained to be a value in the **tweet** entity set.
- `pic_filename`. The filename where the image has been stored, if downloaded.
- `url`. The url that typically is included in the text of the tweet containing the image.
- `media_url`. The expanded url for the image's location on the Twitter server.
- `display_url`. The url that displays when a WWW user points to a hyperlink to the URL.

Attributes for **followers** Table

- `ego_id`. The `user_id` for the user account is being followed by another user. This variable is constrained to come from **user** entity set.
- `follower_id`. The `user_id` for the user account following the user. This variable is *not* constrained to come from **user** entity set.

Attributes for **tweet_usermentions** Table

- `tweet_id`. The `tweet_id` referencing the **tweet** entity containing the user mention. This variable is constrained to be a `tweet_id` in the **tweet** entity set.
- `user_mention_id`. The `user_id` associated with the mentioned user's account. This variable is *not* constrained to be a `user_id` in the **user** or **user_profile** entity set.
- `user_mention_screen_name`. The `screen_name` associated with the mentioned user's account.
- `user_mention_name`. The `name` associated with the mentioned user's account.

Attributes for **tweet_hashtags** Table

- `tweet_id`. The `tweet_id` referencing the **tweet** entity containing the hashtag. This variable is constrained to be a `tweet_id` in the **tweet** entity set.
- `hashtag`. The hashtag contained in the referenced tweet.

Attributes for **tweet_url** Table

- `tweet_id`. The `tweet_id` referencing the **tweet** entity containing the url. This variable is constrained to be a `tweet_id` in the **tweet** entity set.
- `url`. The compressed url contained in the tweet.
- `expanded_url`. The full url (if available) linked in the tweet.