Praktikumsbericht
DSSN Tweet Text Corpus

Robert Rößling
Universität Leipzig

2. Februar 2016

Inhaltsverzeichnis

1 Einleitung und Aufgabenstellung

Im Rahmen des diesjährigen DSSN-Praktikums sollten in einem Team von anfänglich 5 und später 3 Studenten verschiedene Programme entworfen werden, welche das dezentrale semantische soziale Netzwerk Xodx[1] testen sollen, um statistische und technisch relevante Beobachtungen erheben zu können.
Neben dem Konfigurieren und Erstellen von Docker[2] Containern war eine der Kernaspekte in dem Praktikum das Erstellen eines logisch aufgebauten Textkorpus, welcher von anderen Programmen verwendet werden kann, um Xodx einem Stresstest unterziehen zu können. Die Daten aus dem Textkorpus dienen schlussendlich dazu, einen Agenten zu instantiieren, um ein typisches Nutzerverhalten auf dem Netzwerk zu simulieren.
Dieser Praktikumsbericht erläutert näher den Aufbau und die Funktionsweise des Programms, welches den Textkorpus erstellt, sowie die einzelnen Beweggründe, verschiedene APIs Dritter zu nutzen und warum diese notwendig sind.
Aufgrund des Umstandes, dass der Textkorpus für das Testen von Xodx genutzt werden soll, ergibt sich, dass die erhobenen Datensätze leicht auf Strukturen von sozialen Netzwerken anwendbar sein sollen. Diese Strukturen beinhalten unter anderem den Nutzernamen, eine eindeutige ID, Personen die einer anderen Person folgen, usw.. Um eine möglichst glaubwürdige und realistische Distribution der Datensätze zu erhalten, sollten Realdaten verwendet werden.
Ein weiterer wichtiger Aspekt ist, die Ordnung der Datensätze, so dass jedem Nutzer ein bestimmter Teil der Daten zugewiesen werden kann.

2 Recherche

Wie in Kapitel 1 erwähnt, sollen Realdaten verwendet werden, welche eine logische Struktur besitzen, die auf soziale Netzwerke angewendet werden kann. Es liegt daher nahe, Datensätze von einem bereits existierenden sozialen Netzwerk zu verwenden, welches eine ähnliche Struktur wie Xodx aufweist.
Aufgrund der dezentralen Natur von Xodx gibt es keine konkreten Freundschaftsanfragen eines Nutzers zu einem anderen Nutzer, sondern nur die Möglichkeit, einem Nutzer zu folgen. Dies ähnelt stark dem Follower-Prinzip von Twitter[3]. Twitter selbst besitzt gut dokumentierte und umfangreiche REST APIs[4], die es ermöglichen, verschiedene Metadaten von realen Nutzern mittels eines Programms über die Rückantworten, welche in JSON[5] vormuliert sind, der REST APIs zu erhalten. Ein weiterer wichtiger Punkt ist auch die gegebene Sicherheit, welche durch OAuth[6] erreicht wird, ohne dass die Passwörter an Anwendungen Dritter übergeben werden müssen. Auch wird die Twitter API von vielen verschiedenen Libraries in den unterschiedlichsten Programmiersprachen

[1] http://aksw.org/Projects/Xodx.html
[2] https://www.docker.com/
[3] https://twitter.com
[4] https://dev.twitter.com/rest/public
[5] http://json.org/
[6] https://dev.twitter.com/oauth

unterstützt[7].
Eine besonders praktische Programmiersprache, insbesondere für das Erstellen und Handhaben von langen Strings bzw. XML / HTML Dateien ist Python, welche ebenfalls mehrere Libraries besitzt, die die Twitter API unterstützten. Nach näherem betrachten der vorhandenen Libraries, welche die Twitter API für Python unterstützen, erschien „Python Twitter Tools"[8] als die beste Wahl. Für die Erstellung eines Textkorpus sind nur einfache Anfragen an die REST APIs notwendig, folglich ist eine möglichst leicht gehaltene Library, welche ebenfalls schnell und einfach installiert werden kann und die eine einfach zu bedienende Schnittstelle besitzt, zu bevorzugen. Durch den Umstand, dass Python Twitter Tools im Vergleich zu den anderen Libraries erfüllt, fiel die Wahl auf diese Library.
Schlussendlich müssen, da die Datensätze stringent und strukturiert dargestellt bzw. gespeichert werden sollen, die erhaltenen Daten in XML Dateien transformiert und gespeichert werden. Hierfür stellen sowohl die Standardlibrary Element Tree[9] von Python als auch Beautifoul Soup 4[10] eine gute Möglichkeit dar, dies umzusetzen. Aufgrund von Schwierigkeiten, die im Kapitel 7 näher erläutert werden, wurden beide Libraries genutzt.

3 Twitter REST APIs

Wie im Kapitel 2 zu erkennen ist, stellt eine der wichtigsten Komponenten die Twitter API dar. In diesem Kapitel soll nur auf die Anfragen eingegangen werden, welche auch von dem Programm genutzt werden und auf andere wichtige Gegebenheiten der Twitter API, welche beachtet werden müssen.

3.1 `GET statuses/user_timeline`

Diese Anfrage[11] gibt eine Datensammlung der neusten Tweets eines spezifischen Nutzers zurück. Der Nutzer wird entweder durch seinen `screen_name`, also Anzeigename, oder seiner `user_id` deklariert.
Die Anfrage kann allerdings nicht auf alle Tweets eines Nutzers zugreifen, sondern nur auf die letzten 3200. Dabei ist es irrelevant, ob es sich bei einem Tweet um einen so genannten Retweet, also eine Weiterverbreitung eines bereits abgesendeten Tweet[12], handelt oder nicht. Falls keine Retweets in der Ergebnismenge erwünscht sind, kann dies mittels des boolschen Parameters `include_rts` erreicht werden.

[7] `https://dev.twitter.com/overview/api/twitter-libraries`
[8] `http://mike.verdone.ca/twitter/`
[9] `https://docs.python.org/2/library/xml.etree.elementtree.html`
[10] `http://www.crummy.com/software/BeautifulSoup/`
[11] `https://dev.twitter.com/rest/reference/get/statuses/user_timeline`
[12] `http://www.twitter-leitfaden.de/fragen/was-ist-ein-retweet`

Es soll nun kurz auf alle Parameter der Anfrage eingegangen werden.

- `user_id`
  Die ID des Nutzers, dessen Timeline ausgegeben werden soll, z.B. 12345.

- `screen_name`
  Der Anzeigename des Nutzers, dessen Timeline ausgegeben werden soll, z.B. noradio.

- `since_id`
  Dieser Parameter legt eine ID fest, welche als untere Schranke dient, denn es werden nur Tweets zurückgegeben, welche eine ID besitzen, die größer oder gleich des Parameters ist, wie z.B. 12345.

- `count`
  Gibt die Anzahl der Tweets an, welche versucht werden zurückzugeben, bis zu einem Maximum von 200 pro Anfrage. Dabei werden auch Retweets mit angerechnet, selbst wenn `include_rts` nicht gegeben ist.

- `max_id`
  Analog zu dem Parameter `since_id` wird hier eine obere Schranke festgelegt.

- `trim_user`
  Wenn dieser boolsche Parameter auf `true` gesetzt wird, beinhaltet die Ausgabe ein Nutzerobjekt, welches nur aus der Nutzer ID besteht.

- `exclude_replies`
  Wie der Name schon suggeriert, werden alle Tweets, welche einfach nur Antworten auf andere Tweets sind, ausgeschlossen und nicht mit angegeben.

- `contributor_details`
  Sollte dieser Parameter `true` sein, so wird neben der `user_id` eines Antworttweets auch noch der `screen_name` übergeben.

- `include_rts`
  Wie schon näher erläutert, können über diesen Parameter Retweets aus der Ergebnismenge entfernt oder hinzugefügt werden. Dabei sei anzumerken, dass Retweets ein komplettes Nutzerobjekt beinhalten.

3.2 GET followers/list

Diese Anfrage[13] gibt eine Datensammlung von Nutzerobjekten zurück, welche einem spezifischen Nutzer folgen.
Dabei werden die Ergebnisse chronologisch geordnet, beginnend mit dem Nutzer, der den angegebenen Nutzer am kürzesten folgt. Die Daten werden in Gruppen von 20 Nutzern und über mehreren Seiten hinweg ausgegeben; Mittels `next_cursor` kann durch diesen mehrseitigen Datensatz navigiert werden.
Es wird nun, analog zu dem Kapitel 3.1, auf die jeweiligen möglichen Parameter der Anfrage eingegangen.

- `user_id` **und** `screen_name`
  Verhält sich identisch zu den gleichnamigen Parametern aus Kapitel 3.1.

- `cursor`
  Dieser Parameter ist semi-optional und definiert, welche Seite angezeigt werden soll. Dabei stellt -1 den Standardwert dar und gibt die erste Seite zurück.

- `count`
  Analog zu der Anfrage aus Kapitel 3.1, beschreibt dieser Parameter die Anzahl der Nutzer, welche pro Seite ausgegeben werden sollen. Der Standardwert liegt bei 20, während das Maximum erneut bei 200 liegt.

- `skip_status`
  Falls dieser Parameter `true` gesetzt wird, wird der Status der einzelnen Nutzer, wie der Name andeutet, nicht in die Ergebnismenge inkludiert. Ansonsten wird der Status eines Nutzers stets übergeben.

- `include_user_entities`
  Der Nutzerobjektknoten `entities`, auf den hier nicht weiter eingegangen werden soll, wird nicht übergeben, wenn dieser Parameter auf `false` gesetzt wird.

3.3 Rate Limits

Um zu verhindern, dass zu viele Anfragen gesendet werden, besitzen die Twitter REST APIs verschiedene Rate Limits[14]. Diese Rate Limits beziehen sich stets auf einen Zeitrahmen von 15 Minuten und werden nach Erreichen der Limitation wieder zurückgesetzt. Wird ein Rate Limit erreicht und dennoch eine Anfrage gestellt, so wird ein HTTP 429 zurückgegeben mit der Nachricht „Too Many Requests".
Aufgrund des Umstandes, dass nur 2 Endpunkte der REST APIs genutzt werden, sind auch nur deren Limitationen für dieses Programm relevant. Während bei `GET statuses/user_timeline` die Grenze bei 180 Anfragen pro 15 Minuten liegt, fällt die Grenze für `GET followers/list` mit 15 Anfragen pro 15

[13] `https://dev.twitter.com/rest/reference/get/followers/list`
[14] `https://dev.twitter.com/rest/public/rate-limits`

Minuten deutlich geringer aus, was nicht weiter kritisch ist. Schlussendlich wird die Anfrage `GET statuses/user_timeline` um einiges häufiger gestellt als `GET followers/list`.
Es sei hier noch angemerkt, dass zwischen den nutzerauthentifizierten Rate Limits und den anwendungauthentifizierten Rate Limits unterschieden werden muss. Während die oben genannten Limitationen die der Nutzerauthentifikation waren, wären die Limitationen der Anwendung selbst, knapp um das Doppelte mit jeweils 30 bzw. 300 Anfragen pro 15 Minuten höher. Dies bezeichnet jedoch ein Rate Limit, welches von allen Instanzen, die die selbe Anwendungsauthentifikation besitzen, zusammen erreicht werden kann. Das Programm selbst, welches hier genutzt werden soll, stellt aber nur eine Instanz dar, weshalb auch die Limitationen der Nutzerauthentifikation betrachtet werden müssen.

3.4 Anmeldung bei Twitter

Wie das Kapitel 3.3 schon erwähnt, sind Authentifizierungen notwendig, um Anfragen an die REST APIs von Twitter zu stellen. Dies setzt einen Account bei Twitter voraus, über welchen eine Anwendung registriert werden kann.
Die hier beschriebene Anwendung wurde über den Twitter Account ,,mam09cqz" unter den Namen ,,DSSNTweetCorpus"[15] registriert. Nach dieser Registrierung werden die jeweils eindeutigen Schlüssel und Token übermittelt, welche notwendig sind, um den Zugriff der Anwendung auf die Twitter API zu ermöglichen. Diese Schlüssel werden nicht in dem Quellcode selbst eingebunden, sondern separat in einer Datei ,,keys.txt" gespeichert, auf welche das Programm zugreift, um die Authentifizierung durchzuführen.

4 Python Twitter Tools

In Kapitel 2 wurde die Library Python Twitter Tools kurz bzgl. der Recherche beschrieben. In diesem Kapitel soll näher auf die Installation und Nutzung dieser Library eingegangen werden, welche einen sicheren, schnellen und einfachen Zugriff auf die Twitter REST APIs ermöglicht.

4.1 Installation

Es gibt 2 Möglichkeiten, diese Library zu installieren. Es können die Basispakete herunter geladen werden, um diese manuell zu installieren. Dies kann allerdings aus unterschiedlichsten Gründen recht komplex werden und sollte, wenn möglich, vermieden werden. Stattdessen kann dieses Paket über das Modul `easy_install`[16] einfach installiert werden.

[15] `https://apps.twitter.com/app/7167263`
[16] `https://pythonhosted.org/setuptools/easy_install.html`

Abbildung 1: oAuthDance Methode

```
def oAuthDance(self, keys):
    """OAuth Dance with Twitter"""
    MY_TWITTER_CREDS = os.path.expanduser('~/.my_app_credentials')
    CONSUMER_KEY = keys[0]
    CONSUMER_SECRET = keys[1]
    oauth_token = keys[2]
    oauth_token_secret = keys[3]
    if not os.path.exists(MY_TWITTER_CREDS):
        oauth_dance("DSSNTweetCorpus", CONSUMER_KEY, CONSUMER_SECRET,
            MY_TWITTER_CREDS)
    auth = OAuth(oauth_token, oauth_token_secret, CONSUMER_KEY,
        CONSUMER_SECRET)
    return auth
```

4.2 Nutzung

Die Library kann mittels `from twitter import *` eingebunden werden. Besonders praktisch ist der Umstand, dass die einzelnen Methoden der Twitter API nachempfunden sind. So kann die Anfrage `GET statuses/user_timeline` mit folgendem Befehl gestellt werden:

```
twitter.statuses.user\_timelien(screen\_name='billybob')
```

Es ist auch ersichtlich, dass die einzelnen, bereits erwähnten Parameter der Anfragen gleichnamig sind und einfach mit übergeben werden können. Auch zu erkennen ist, dass ein Twitterobjekt benötigt wird, welches mittels den einzelnen Schlüsseln und Tokens der registrierten Anwendung erzeugt werden kann.
Die von der Anfrage erhaltenen Daten sind in einer geordneten Matrix enthalten. Auf diese kann einfach, mittels der Namen der einzelnen Listen, zugegriffen werden. So erhält man den Anzeigenamen des Nutzers, welcher den ersten Tweet geschrieben hat, mit folgendem Befehl:

```
timeline[0]['user']['screen\_name']
```

4.3 OAuth Dance

Der so genannte Tanz mit OAuth ist notwendig, um die Anwendung, welche genutzt werden soll, authentifizieren zu können. In Abbildung 1 steht die Methode, welche diesen OAuth Dance ausführt. Das zurückgegebene Objekt ,,`auth`" ist notwendig, um das im Kapitel 3.4 beschriebene Twitterobjekt zu erstellen. Auch ist erkenntlich, dass eine Liste namens ,,`keys`" initial übergeben werden muss. Diese Liste enthält alle Schlüssel und Token, welche für die Authentifizierung

notwendig sind. Diese Liste wird mittels einer anderen Methode aus der oben genannten Datei ,,keys.txt" ausgelesen. Die Datei ist demnach elementar und wird benötigt, soll allerdings aus Sicherheitsgründen nicht online gelagert werden.

5 Aufbau

Es soll nun auf den allgemeinen Aufbau des Programms eingegangen und die einzelnen Aufgaben und Hintergründe der jeweiligen Klassen näher erläutert werden. Der Quellcode und diverse Dokumentation ist auf GitHub unter dem Namen ,,DSSNTweetCorpus"[17] zu finden. Es wird mit den beiden Objekten, die durchweg in diesem Programm genutzt werden, begonnen.

5.1 `User` und `Tweet`

Die wichtigsten Variablen des Nutzerobjektes sind die `uid`, die ID des Nutzers, welche allerdings keine interne zugewiesene ID ist, sondern die tatsächliche, von Twitter vergebene, ID. Der `screen_name` ist selbsterklärend der Anzeigename des Nutzers; Dahingegen ist `name` der von dem Nutzer angegebene vollständige Name, `tweets` beinhaltet eine Liste der letzten Tweets des jeweiligen Nutzers, `followers` respektiv eine Liste mit den Nutzer IDs aller Nutzer, welche diesen Nutzer folgen, `deltaTime` gibt die Differenzzeit in Sekunden zwischen der Initiierung des Agenten auf dem Docker Container und der Registrierung an der Plattform an und `protected` sagt aus, ob das Profil des Nutzers geschützt ist oder nicht, was wichtig ist, um dessen Follower bzw. Tweets zu erhalten.
Die anderen Variablen werden entweder im Moment nicht genutzt oder sind selbsterklärend und nicht weiter wichtig.
Analog zu dem Nutzerobjekt existiert das Tweetobjekt, auch hier geht der Sinn und Zweck der jeweiligen Variablen aus dem Namen hervor, es sollen dennoch die wichtigsten benannt und kurz erläutert werden.
Dabei stellt `tid` die von Twitter vergebene Tweet ID dar, welche einzigartig für jeden Tweet ist. `text` beinhaltet logischerweise den Inhalt des Tweets, `isReply` gibt an, ob der Tweet eine Antwort auf einen anderen war, während `replyTo` die Nutzer ID von, dem Nutzer beinhaltet dem diese Antwort galt. Wichtig ist auch `isFollowRequest`, was angibt, ob mittels diesem Tweet der Nutzer nun einem anderen Nutzer folgen möchte, dabei gibt `followRequestToId` die Nutzer ID an, der der Nutzer nun folgt. Schlussendlich gibt es noch `deltaTime`, diese gibt erneut die Zeit in Sekunden an, die zwischen der Erstellung des Nutzers und dem Erstellen dieses Tweets vergangen sind.

5.2 `Corpus`

Die Klasse `Corpus` beinhaltet, wie der Name suggeriert, den eigentlich Textkorpus, welcher schlussendlich ausgegeben werden soll. Dieser Textkorpus stellt die Klassenvariable `users` dar; Sie enthält also jeden einzelnen Nutzer in einer

[17] https://github.com/DSSN-Practical/DSSNTweetCorpus

Liste.
Die Klasse selbst wird mit `keys` initialisiert, dass ist der genaue Dateipfad der Textdatei, welche die Schlüssel und Token von der registrierten Twitter Anwendung beinhaltet.
Es soll nun kurz auf die einzelnen Methoden eingegangen und deren Nutzen erläutert werden:

- `readKeys`
  Der Name ist selbsterklärend. Diese Methode liest mittels der Klassenvariable `keys` die einzelnen Schlüssel aus und gibt diese in Form einer Liste zurück.

- `oAuthDance`
  Diese Methode wurde schon im Kapitel 4.3 näher analysiert.

- `getUserTimeline` **und** `getUserFollowers`
  Diese beiden Methoden führen respektiv die Anfragen an die Twitter APIs aus, um entweder die Timeline, also die neusten Tweets, oder die einzelnen Followers eines Nutzers zu erhalten und schlussendlich zurückzugeben.

- `halt`
  Sollte das Rate Limit erreicht werden, greift diese Methode und hält das Programm für 15 Minuten an.

5.3 `Friender`

Diese Klasse wird mittels einem Corpusobjekt initialisiert, um alle Nutzer und deren Tweets bearbeiten zu können.
Die einzige Aufgabe dieser Klasse ist es, die einzelnen Nutzer sich untereinander folgen zu lassen, da die tatsächlichen Tweets leider kein konkretes follower Attribut besitzen, welches hier Anwendung gefunden hätte.
Die beiden Methoden `setUserFollowers` und `setUserFollowerrequests` dienen dazu, einerseits eine Liste mit Nutzern für einen spezifischen Nutzer zu erzeugen, die eben diesem folgen und andererseits einzelne Tweets mit jeweils einem Element aus dieser erzeugten Liste als eine Followeranfrage zu designieren.
Die Klasse gibt keine Variable zurück, sondern verändert einfach nur den bereits existierenden Textkorpus, der bereits in dem Corpusobjekt liegt.

5.4 `Handler`

Nicht nur beinhaltet diese Klasse die `main` Methode, sie dient ebenfalls dazu, die gesamten Funktionsaufrufe der anderen und der eigenen Klasse zu koordinieren.
Initial wird ein Corpusobjekt erstellt, dann das Twitterobjekt mittels dem Authentifizierungsobjekt, welches von der Methode oAuthDance übergeben wird.
Danach folgen die anfänglichen 3 Anfragen, die näher im Kapitel 6 erläutert werden.

Es wird nun kurz auf die einzelnen Methoden eingegangen:

- `addUsers` **und** `addUserTweets`
  Während `addUsers` das Ergebnis der `GET followers/list` Anfrage bearbeitet und ein neues Nutzerobjekt für jeden Follower erzeugt, so erzeugt `addUserTweets` ein Tweetobjekt für jeden einzelnen Tweet, welche mittels der `GET statuses/user_timeline` Anfrage erhalten werden und weist diese dem Nutzer zu.

- `createDeltaTimes`
  Diese Methode übergibt jedem Tweet eines Nutzers eine Differenzzeit in Sekunden, basierend auf den tatsächlichen Zeitabstand der Tweets untereinander. Dabei wird die von dem Nutzer festgelegte `maxtime` nie überschritten.

- `createDir, createOutputFile` **und** `createUserEntry`
  Während die Methode `createUserEntry` für jeden Nutzer einen XML basierten Eintrag mittels `ElementTree` und `BeautifulSoup` generiert, erzeugt `createDir` ein Ausgabeverzeichnis und `createOutputFile` eine Ausgabedatei, welche in eben diesem Verzeichnis gespeichert wird, für den jeweiligen Nutzer.

- `startHandling`
  Schlussendlich dient diese Methode für das allgemeine und koordinierte Ausführen aller in dieser Klasse befindlichen Methoden, es wird weiterhin gefragt, ob die Daten gespeichert werden sollen oder nicht.

6 Nutzung, Eingaben und Ausgaben

Wie im Kapitel 4.4 beschrieben, werden in der Klasse `Handler` 3 initiale Eingaben von dem Nutzer erwartet.
Zunächst wird nach dem Anzeigenamen des initialen Twitter Nutzers gefragt. Von diesem Nutzer werden alle Nutzer, die ihm folgen, ermittelt, damit die erste Iteration, um neue Nutzer erstellen zu können, eingeleitet werden kann.
Als nächstes wird nach der Anzahl der Iterationen gefragt, das bedeutet, wie oft nochmals die Anfrage `GET followers/list` durchgeführt werden soll, um neue Nutzer hinzuzufügen.
Schlussendlich wird nach der maximalen Differenzzeit gefragt, hier jedoch in Tagen. Diese Variable wird genutzt, um die einzelnen `delta` Werte für die Tweets oder Nutzer übergeben zu können. Falls eine Zeitspanne von 10 Tagen eingegeben wird, werden die einzelnen Tweets anhand des tatsächlichen Abstandes der Tweets auf einer Zeitspanne von 10 Tagen verteilt.
Der allgemeine Ablauf des Programms ist relativ simpel. Es werden zunächst alle Eingaben vom Nutzer entgegen genommen und danach sofort via der Anfrage `GET followers/list` neue Nutzerobjekte erstellt, bis die Anzahl der angegebenen Iterationen erreicht ist. Im Anschluss werden die Timelines aller Nutzer abgefragt und für jeden Tweet ein Tweetobjekt erstellt, welches in die Liste

`tweets` von jedem Nutzer gespeichert wird.
Schlussendlich erfolgt die Abfrage, ob das Programm terminieren soll, die Daten einfach in der Konsole ausgegeben oder ob die Daten stringent als XML ausgegeben und gespeichert werden sollen.
Die Ausgabe erfolgt in einem neuen Ordner, welcher das momentane Datum und die Uhrzeit im Namen trägt. In diesem Ordner finden sich XML Dateien für jeden Nutzer. In Abbildung 2 ist eine Beispieldatei zu sehen. Deren Aufbau ist selbsterklärend und die Variablen wurden im Kapitel 3 schon näher erläutert.

7 Zusammenfassung

Im Laufe des DSSN Praktikums musste eine konkrete Lösung erstellt werden, um einen Textkorpus zu erstellen, welcher von Agenten einfach genutzt werden kann. Dabei ergaben sich neben gruppeninternen Problemen, wie dem Ausstieg von 2 Teilnehmern, auch spezifische technische Probleme bzgl. des hier beschriebenen Programms.
Während zunächst unklar war, wie die Ausgabe genau aussehen sollte und welche Kriterien der Textkorpus erfüllen muss, ergab sich eine Problematik bzgl. der übermäßigen RAM Nutzung mit der nativen Python Library `ElementTree`, falls zu viele Elemente hinzugefügt werden. Um dieses Problem zu umgehen, wurde für jeden Nutzer eine eigene XML Datei angelegt.
Ein weiteres Problem war die Ausgabe selbst. Während `ElementTree` einige spezielle Zeichen in der Ausgabedatei nicht erlaubt, so existiert diese Problematik in `BeautifulSoup` nicht. Aus diesem Grunde werden die XML Dateien von `BeautifulSoup` erstellt, um fehlerfreie Ausgaben zu ermöglichen.
Ein anderes Problem war ebenfalls, wie schon im Kapitel 5.3 erwähnt, zu ermöglichen, dass Nutzer sich untereinander folgen. Dieses Problem hatte sowohl einen technischen Aspekt bzgl. der Umsetzung als auch einen theoretischen Aspekt in Bezug auf die statistische Verteilung, welche je nach Belieben nachträglich verändert werden kann, um andere Szenarien simulieren zu können.
Nachdem diese Probleme überwunden wurden, konnte an dem Projekt mit klar definierten Aufgaben schnell weitergearbeitet werden, was im momentanen Stand des Programms resultiert. Aufgrund der einzelnen Nutzer- und Tweetobjekte und der objektorientierten Umsetzung des Programms können weitere, feingranularere Aspekte hinzugefügt werden, falls weitere Anforderungen bzgl. des Textkorpus aufkommen.

Abbildung 2: Beispiel XML Datei

```
<?xml version="1.0" encoding="utf-8"?>
<file>
 <user>
  <id>
   831175088
  </id>
  <name>
   Juerg Werner
  </name>
  <screen_name>
   werjo01
  </screen_name>
  <description/>
  <delta>
   0
  </delta>
  <protected>
   False
  </protected>
  <followers>
   <id>
    113595107
   </id>
  </followers>
  <timeline>
   <tweet>
    <id>
     251016005948698624
    </id>
    <text>
     http://t.co/La0xlcWn
    </text>
    <delta>
     2592000
    </delta>
    <followrequest_to_id>
     113595107
    </followrequest_to_id>
   </tweet>
  </timeline>
 </user>
</file>
```