Projektreport Programmierung II

Aufgabe: Author Profiling – TwiSty-Korpus und Myers-Briggs Typenindikator (MBTI)

1 Aufgabe

Ich habe das Thema *author profiling* mit dem TwiSty-Korpus (Verhoeven et al, 2016) als Datensatz gewählt. Die Aufgabenstellung war, eine Person anhand ihrer Tweets als einen der 16 Persönlichkeitstypen nach Myers-Briggs zu kategorisieren. Es handelt sich also um ein Klassifizierungsproblem mit 16 Klassen; das Geschlecht wurde bei der Klassifizierung außer Acht gelassen.

Die Aufgabe fand ich interessant, da ich zuvor noch nie mit Twitter-Daten gearbeitet hatte. Das von mir entwickelte Programm könnte z.B. für personalisierte Werbung benutzt werden.

Das Projekt ist einfach erweiterbar – für anderssprachige Daten müsste lediglich ein anderes spaCy-Modell geladen werden und ggfs. das Verhältnis im Datensatz-Split an die Datenmenge angepasst werden. Die Features können in den entsprechenden Funktionen angepasst werden. Beim Download der Twitter-Daten werden auch die Beschreibungen (Bios) aller Accounts heruntergeladen, hieraus könnte man bspw. ebenfalls Features extrahieren.

Die ausgewählten Features enthalten sowohl Twitter-Metadaten (die auf die Beliebtheit/Extraversion einer Person rückschließen sollen) als auch linguistische Features (die auf die Persönlichkeit rückschließen sollen, z.B. könnte eine *thinking* Person mehr Zahlen benutzen als eine *feeling* Person, eine *judging* Person mehr Emoticons als eine *perceiving* Person usw.)

User-Ebene
- Follower-Freunde-Verhältnis
- Verifiziertes Profil ja/nein
- Profil-URL ja/nein

Twitter-Ebene
- Durchschnittliche Anzahl Hashtags (#) pro Tweet
- Durchschnittliche Anzahl Mentions (@) pro Tweet
- Durchschnittliche Anzahl Likes pro Tweet
- Durchschnittliche Anzahl Retweets pro Tweet
- Likelihood für Fotos im Tweet
- Likelihood für URLs im Tweet
- Likelihood, dass ein Tweet eine Antwort auf einen anderen Tweet ist

Text-Ebene
- (Alle Features wurden über die Anzahl heruntergeladener Tweets pro Account normalisiert)
- Wortlänge
- Satzlänge
- Tweet-Länge
- Größe d. Vokabulars

- Anzahl Tokens
- Named Entities
- Fragezeichen
- Ausrufezeichen
- Zahlen bzw. Zahl-ähnliche Wörter
- Adjektive
- Emoticons
- Sonderzeichen

2 Programm

2.1 Aufbau

Das Programm ist in drei Dateien aufgeteilt: `main.py`, `mbti_classifier.py` und `twitter_classes.py` (und `test_module.py` für die Unittests).

Die main-Datei übernimmt die Überprüfung des User-Inputs und den Aufruf des entsprechenden Programm-Modus' (Training vs. Inferenz; Näheres hierzu s. README). `twitter_classes.py` enthält die Klassen `User` und `Tweet`, in die die Twitter-Daten nach dem Download gespeichert werden. `mbti_classifier.py` übernimmt die inhaltliche Arbeit und wird im Folgenden näher beschrieben:

Der Konstruktor übernimmt keine weitere Arbeit, außer eine Verbindung zur Twitter-API herzustellen und das spaCy-Sprachmodell zu laden. D.h. alle Funktionen werden in der main-Datei direkt aufgerufen; so gestaltet sich die Aufteilung in Trainings-/Inferenzmodus am einfachsten.

Im Trainingsmodus kann optional der Eingabekorpus zuerst in Trainings-, Validierungs- und Testdaten aufgesplittet werden.[1] Danach werden die Trainingsdaten heruntergeladen, verarbeitet und auf Klassenbasis aggregiert (ungewichtetes Modell). Das Modell wird in eine tsv-Datei geschrieben. Die Testdaten werden ebenfalls vorverarbeitet und anhand des nun trainierten Modells werden die Vorhersagen getroffen. Als Evaluationsmaß wurde die *accuracy* gewählt.

Im Inferenzmodus wird nur die *predict*-Funktion mit den Eingabedaten und dem trainierten Modell aufgerufen.

Aus dem TwiSty-Korpus werden lediglich die User-IDs und die MBTI-Typen benutzt, nicht die Liste mit den Tweets. Das liegt daran, dass teilweise pro Account nur ein einziger Tweet enthalten war, dass einige Tweets aus dem Korpus inzwischen gelöscht sind und dass es länger dauert, Tweets nach ID herunterzuladen als einfach 120 Tweets vor dem 23.08.2020.[2] Aus diesen werden dann Retweets entfernt, die ja nicht von der Person selbst verfasst wurden, sodass pro User ca. 80-100 Tweets übrig bleiben.

[1] Weil das Programm auf dem deutschen TwiSty-Korpus entwickelt wurde, musste ich hier das Verhältnis von 70:10:20 zu 60:19:21 anpassen, da sonst nicht jede Klasse in jedem Datensatz enthalten gewesen wäre.

[2] Es wurde ein beliebiger Tweet als „Höchstalter" gewählt anstatt die 120 neuesten Tweets zu laden, damit die Daten reproduzierbar bleiben.

Um die Laufzeit zu verringern, wurde Concurrency implementiert. Der größte Zeitfresser war das Herunterladen der User-Daten, also eine I/O-Operation, weshalb ich mich für Threading statt Multiprocessing entschieden habe. Das stellte keine große Herausforderung dar, denn die gethreadeten Funktionen sind *stateless*, d.h. sie liefern für gleichen Input (User-ID) immer denselben Output (Tweet-/User-Objekte) und sind daher *thread-safe*. Die heruntergeladenen Daten werden erst im Anschluss in einen gemeinsamen Pandas DataFrame gespeichert.

2.2 Module

Standardmodule
- collections.namedtuple – Datenstruktur für die Features
- concurrent.futures – Threading
- logging – Logging
- os – um die Twitter-Credentials aus einer `.env`-Datei lesen zu können und sie nicht hartkodieren zu müssen

Externe Module
- dotenv – um die Twitter-Credentials aus einer `.env`-Datei lesen zu können und sie nicht hartkodieren zu müssen
- pandas – DateFrame als primäre Datenstruktur
- sklearn.model_selection.train_test_split – Aufsplitten des Korpus'
- spacy – für die linguistischen Features
- tweepy – Twitter-API

Eigene Module
- Tweet – eigene Klasse zur Repräsentation von Tweets
- User – eigene Klasse zur Repräsentation von User_innen

3 Reflektion

3.1 Workflow

Ich habe zum ersten Mal intensiver mit Git/GitHub gearbeitet, was sich als sehr angenehm herausgestellt hat. Manchmal habe ich aus Versehen zu früh/zu spät committed, aber da ich alleine an dem Projekt gearbeitet habe, hab ich mir darüber nicht zu viele Gedanken gemacht. Sehr nützlich fand ich das Issues-Feature auf GitHub, wo ich mir verschiedene To-Do-Listen angelegt habe. Auch das Reviewen war sehr einfach über GitHub.
Ich habe meist den Tag über immer wieder lokal committed und dann zum Feierabend alles gepusht. Teilweise habe ich auch mit verschiedenen Branches gearbeitet.

Wie empfohlen habe ich zuerst eine Minimalversion des Programms erstellt und das dann nach und nach um mehrere Features, Logging, Threading, etc. erweitert. So hatte ich auf jeden Fall gute Kontrolle über den Zeitplan und konnte auch mal einen Tag lang anspruchslosere Arbeiten wie Docstrings schreiben machen.

3.2 Leichtes und Schweres

3.2.1 Leichtes

Prinzipiell fiel mir die Arbeit relativ leicht, v.a. war ich motivierter als bei anderen Programmierprojekten, weil es eine klare Erwartungshaltung gab (Aufgabenbeschreibung und Bewertungsprotokoll) und ich immer ein Ziel vor Augen hatte. Da ich schon im sechsten Semester bin, hatte ich mit dem Programmieren an sich keine großen Probleme.

In einem anderen Kurs habe ich dieses Semester öfter mit `pandas` gearbeitet, weshalb ich hier mein Wissen noch vertiefen wollte. Bei einem binären Klassifikationsproblem hätte man bestimmt auch einfach mit Listen von Features arbeiten können, aber bei meiner Aufgabe war es eine große Hilfe und das Einarbeiten hat sich m.M.n. gelohnt.

Logging war enorm hilfreich beim Testen und ich frage mich, warum uns das nicht schon früher gezeigt wurde.

Das Reviewen für Hannah Peuckmann hat mir erwartungsgemäß Spaß gemacht. Ich denke, ich konnte ihr noch sinnvollen Input geben, obwohl sie die Aufgabe schon gut bearbeitet hatte. Ich habe dabei auch neue Anstöße für meinen Code bekommen und mir dann gewunschen, dass wir zusammengearbeitet hätten. Ihre Kommentare waren für mich v.a. bei einigen Details auch hilfreich.

Die Arbeit mit den im Kurs neu gelernten Inhalten war meistens leichter als erwartet. Ich fand die Links auf Moodle größtenteils hilfreich. Überrascht hat mich dagegen, dass es nicht trivial ist, eine sinnvolle Programmstruktur aufzubauen, obwohl die Funktionsweise eines Klassifikators ja eigentlich recht unkompliziert ist.

3.2.2 Schweres

Die größte Hürde für mich war Concurrency. Weil ich vorher noch nie mit dem Thema in Berührung gekommen war, hatte ich großen Respekt davor und habe noch mal alle Materialien aus dem Kurs gelesen, bevor ich irgendwas implementiert habe. Ich hätte mir aber extrem viel Zeit sparen können, wenn ich vorher herausgefunden hätte, dass `asyncio` (was ich zuerst verwendet habe) nicht mit `tweepy` kompatibel ist. Nach dem Umstieg auf `concurrent.futures` ging es dann zum Glück ganz leicht.

Im Zuge dessen habe ich auch den gesamten Teil der Feature-Extraktion umgeschrieben und verwende dafür jetzt Generators. Das würde ich mir nächstes Mal vorher überlegen statt einfach mit `pandas` und `df.apply()` loszulegen. Außerdem würde ich in Zukunft unbedingt die Arbeit an den Features in eine separate Klasse auslagern, was meinen Code viel übersichtlicher gemacht hätte. Leider war es irgendwann einfach zu spät, um das noch sinnvoll zu machen.

Die Accuracy des Klassifikators ist natürlich sehr schlecht. Beim Herumprobieren mit Features bin ich nie über 10% gekommen und habe mich letztendlich dazu entschieden, fast alle ursprünglichen Features beizubehalten, da wir ja sowieso NLP implementieren sollten. Ich denke, es liegt daran, dass 1. ich nicht allzu viel darüber weiß wie sich die einzelnen MBTI-Typen unterscheiden, 2. Persönlichkeit sich nur schwer in einer handvoll handgeschriebenen Features

einfangen lässt und 3. viele Menschen nicht nur in einer Sprache twittern. Außerdem ist der Trainingskorpus mit nur 411 Datenpunkten sehr klein – v.a., da viele Accounts inzwischen nicht mehr zugänglich sind – und einige Klassen waren <10x vertreten. Weil Accuracy aber nicht der Fokus des Projekts war, habe ich zu viel Zeit darauf verwendet, perfekte Features zu finden.

Die meiste Zeit habe ich damit verbracht, zunächst schnell hingeschriebenen Code (um erst mal zu schauen, ob es funktioniert) neu zu strukturieren. Bspw. wurde anfangs die Unterscheidung in Trainings-/Inferenzmodus im Konstruktor des Klassifikators gehandled, was sich aber als recht unübersichtlich erwiesen hat. Ich habe mir auf jeden Fall viele Gedanken darüber gemacht, was am einfachsten und elegantesten ist.
Ansonsten habe ich vor allem anfangs lange recherchiert, welche Module ich verwenden möchte und wie diese am effizientesten arbeiten. Mit Ausnahme von `pandas` habe ich mit allen Modulen zum ersten Mal gearbeitet. Bei technischen Schwierigkeiten habe ich meist auf StackOverflow Hilfe gefunden und bei Designentscheidungen mich manchmal mit Kommiliton_innen ausgetauscht.

3.3 Weggelassenes

Anfangs wollte ich im Inferenzmodus neben User-IDs auch „normale“ Usernamen (d.h. Strings) als Eingabe erlauben. Das wäre aber jetzt am Ende aufwändig zu implementieren gewesen, weil das Programm sich an mehreren Stellen darauf verlässt, dass `user_id` ein int64 ist. Stattdessen würde ich ein separates Skript schreiben, um Usernamen in IDs umzuwandeln und diese dann in den Klassifikator geben.

Eigentlich müsse man bei jedem Tweet die Sprache überprüfen und entweder weitere Sprachmodelle laden oder nur die deutschen Tweets behalten. Ich habe mich gegen Sprachidentifikation entschieden, weil sie die Laufzeit enorm verschlechtert hätte und meine Features weitestgehend sprachunabhängig sind (Emoticons sind Emoticons sind Emoticons). Außerdem waren im TwiSty-Korpus tlw. auch englischsprachige Tweets unter den angeblich *confirmed tweet ids*.

4 Literatur

Verhoeven, B., Daelemans, W., & Plank, B. (2016). TwiSty: a multilingual Twitter Stylometry corpus for gender and personality profiling. *Proceedings of the 10th International Conference on Language Resources and Evaluation*. Portorož, Slovenia.

Autor_in

Noël Simmel
simmel@uni-potsdam.de
Universität Potsdam, Matrikelnummer 791794
PRO2-A, Sommersemester 2020
Version 1.0, 31.08.2020