PRO-2A SoSe 2020 Modulprojekt
Annalena Kohnert
Matrikelnr. 785523
ankohnert@uni-potsdam.de

Projekt Authorship Attribution – Report

Die Aufgabe

Ich habe mich dafür entschieden, Tweets einem Autor zuzuordnen und dabei als Datensatz Tweets von Hillary Clinton und Donald Trump Tweet zu den US-Präsidentschaftswahlen 2016 verwendet (https://www.kaggle.com/ benhamner/clinton-trump-tweets). Diese habe ich im Verhältnis 70:20:10 in Trainings-, Test- und Validierungsdatensätze aufgeteilt und in jedem Datensatz den gleichen Autorenanteil wie im gesamten Datensatz benutzt.

Bei solch offiziellen Twitteraccounts ist es natürlich recht einfach herauszufinden, wer welchen Tweet verfasst hat, aber grundlegend kann man so eine Art von Klassifizierer auch für die Bestimmung von Alter, Geschlecht oder andere Eigenschaften eines Autors nutzen (Vielleicht benutzen Männer und Frauen andere Emojis? Benutzen ältere Menschen eher die alte Rechtschreibung?).

Das Trainieren und Testen bestand hauptsächlich aus dem Extrahieren von Features aus den Tweets. Dabei habe ich sowohl linguistische Features aus der Twitternachricht als auch andere (Meta)- Daten aus dem Datensatz genutzt (z.B., ob es sich um einen Retweet handelt oder eine Antwort auf einen anderen Tweet), da ich mir überlegt habe, dass eventuell nicht nur der Stil sondern auch die Art der Verwendung des Mediums sich zwischen verschiedenen Personen unterscheiden kann.
Für die linguistischen Features habe mich erst einmal an den vorgeschlagenen Features in der Aufgabenstellung orientiert und damit automatisch angefangen, die Feature in Gruppen einzuteilen: syntaktische (Part-of-Speech-Tags), semantische (Sentiment), auf Wort- und Zeichenebenen basierend (Interpunktionszeichen, verschiedene Arten von Token) usw.
Dies hat mir anfangs geholfen den Überblick zu behalten, hat sich aber anschließend auch als hilfreich dabei herausgestellt, das Programm möglicherweise später mit mehr Features zu erweitern – alle Feature-extrahierenden Klassen erben von *FeatureBaseClass(.py)* und können in eine der Feature-Gruppen in der Klasse *TweetFeatures* (*tweet_features.py)* hinzugefügt werden, sodass sie beim Extrahieren aller Features eines Tweets mit aufgerufen werden.
Zugegenerweise habe ich mich wenig damit beschäftigt, die besten Features auszuwählen, sondern mich eher mit der Umsetzung der Extrahierung beschäftigt und habe daher auch nur eine *Accuracy* von ca. 57% erreicht.

Das Programm

Wenn der Benutzer das Programm über *main.py* aufruft, wird die angegebene Datei mit den Trainingsdaten an die Klasse **DataSetProcessing** übergeben, die die für das Programm relevanten Teile aus der Datei ausliest, ggf. weiter auswertet und speichert. Relevant sind dafür der Autor (*handle*), Text (*text*), ausgewählte Attribute als Meta-Features und die ID, die ich zur Identifikation behalten haben. Solange diese Attribute enthalten und vom gleichen Typ

sind, können auch andere Texte klassifiziert werden (oder es können, etwas aufwändiger, auch die Attribute im Programm angepasst oder Feature hinzugefügt oder entfernt werden).
Da die Daten im csv-Format sind und ich das Format für alle weiteren Ausgaben behalten habe, habe ich viel mit dem Modul *pandas* gearbeitet, das mir viele nützliche Funktionen geboten hat.
DataSetProcessing ruft für jeden Tweet im Datensatz die Klasse **TweetFeatures** auf, die alle möglichen Features aus einem Tweet extrahiert. Dabei habe ich die Features weiter in Gruppen unterteilt:

1. Features, die direkt aus den Attributen/Werten der csv-Datei abgeleitet werden (*meta_features*) – die Klasse **MetaFeatures**
2. Features, die aus der unvorverarbeiteten Twitternachricht berechnet werden (*raw_text_features*) – die Klassen **CharFeatures**, **PunctFeatures**, **SemanticFeatures**
3. Features, die aus vorverarbeitetem Text berechnet werden (*preprocessed_text _features*) – die Klassen **WordFeatures**, **SyntacticFeatures**, **SentenceFeatures**

Für die *preprocessed_text_feature* habe ich SpaCy für PoS-Tagging und Lemmatisierung benutzt, im Vorfeld aber den Tokenisierer und Satzsegmentierer mit dem von SoMaJo (https://github.com/tsproisl/SoMaJo) überschrieben, da dieser deutlich besser mit den Twitter-spezifischen Token umgehen kann (z.B. splittet er keine Hashtags in #-Symbol und Text, oder behandelt URLs und Hashtags am Ende eines Satzes einheitlicher). Außerdem bestimmt SoMaJo verschieden Klassen von Token, die ich ebenfalls für die Features verwendet habe. Dieses Preprocessing geschieht durch die Klasse **Preprocessing**, in der die angepasste SpaCy Pipeline definiert ist.
Für die *semantic_features* habe ich außerdem *textblob* genutzt, dass schnelle und auf den ersten Blick für die Tweets recht akkurate Sentimentanalyse liefert.

Die Feature-Klassen aller Feature-Gruppen erben von **FeatureBaseClass** und können so beliebig erweitert werden (innerhalb einer Klasse oder als neue Klasse), solange sie die Methode *feature_occurences* implementieren, die alle Features, die eine Klasse extrahieren kann, als Dictionary oder Counter zurückgibt. Falls dabei Feature-Namen (Strings) mehrmals vergeben werden, wird **TweetFeatures** darauf aufmerksam und loggt eine Warnung, dass ein Feature mit einem anderen Wert überschrieben wird.
Da das Preprocessing und damit das gesamte Extrahieren der Features etwas länger dauert, habe ich in **DataSetProcessing** noch *tdqm* importiert und den Fortschritt anzeigen lassen, damit der Benutzer informiert ist, dass noch etwas passiert (wenn auch langsam).

Sind alle Feature extrahiert, wird im Trainingsmodus der Mittelwert für jedes Features eines Autors gebildet. Diese Mittelwerte werden ebenso wie die Features jeden einzelnen Tweets in einer csv-Datei gespeichert.

Wird dann der Testmodus aufrufen, wird über die Klasse **Predict** wieder **DataSetProcessing** aufgerufen und für alle Tweets wieder wie im Trainingsmodus die Features extrahiert, direkt nach dem Extrahieren wird die Distanz jedes Features zu dem Mittelwert des jeweiligen Features aller Autoren berechnet. Der Autor, zu dem die geringste Distanz besteht, wird ausgewählt und alle Features, die Vorhersage und der wahre Autor ebenfalls in einer csv-Datei gespeichert. Außerdem werden die Präzision für jeden Autor und für das gesamte Testset angegeben.

Insgesamt habe ich bei anderen Klassen versucht, eher kleine Klassen mit eng verwandten Methoden zu schreiben, um den Überblick zu behalten. Die Klassen delegieren die Verantwortung weiter: **Predict** verweist an **DataSetFeatures**, diese ruft **TweetFeatures** für jeden Tweet, **TweetFeatures** die einzelnen Feature-Klassen und eventuell vorher **Preprocessing**.
Dazu kommen **Split** bzw. **split_data.py** (das spezifisch für den Datensatz ist) und der Unittest in **test_feature_extraction.py**, in dem vier Feature-Klassen mit vier Tweets getestet werden.

Reflektion

Prinzipiell muss ich gestehen, nicht die volle Bearbeitungszeit genutzt zu haben. Wenn es aber möglich ist, denke ich, dass es gut ist, immer wieder ein paar Stunden oder Tage Abstand vom Projekt nehmen zu können, um nicht „betriebsblind“ zu werden – die meisten Einfälle zum Projekt hatte ich meistens, wenn ich gerade nicht vor dem Bildschirm saß.
Obwohl ich die commits und das Verwalten über Git am Anfang eher nervig fang, hat es mir geholfen, das Projekt besser zu strukturieren. Genau zu überlegen, wie man eine commit message nennt, heißt ja auch, dass man genau wissen muss, was man eigentlich alles seit dem letzten Mal verändert hat. Das „git push“ und „git pull“ hat mir anfangs viel Schwierigkeiten bereitet, ebenso, wie ich mit dem Umstrukturieren von Ordnern oder dem Löschen/Verschieben von Dateien umgehen soll. Da ich Git bis zu diesem Kurs nie benutzt habe, war das war ein nervenaufreibender, aber in meinen Augen lohnenswerter Lernprozess.

Das logging und Fehler abfangen habe ich erst spät zum Projekt hinzugefügt habe, ebenso wie mich erst das Review darauf aufmerksam gemacht hat, dass ich die Encapsulation vernachlässigt habe. Zwar konnte ich beides noch nachträglich ohne größeren Aufwand hinzufügen, aber ich denke, dass ich einige Sachen von Anfang an anders (und besser) strukturiert hätte, wenn ich dies gleich mit eingeplant hätte – z.B. wie ich Methoden aufrufe, und ob diese auch im fertigen Projekt für alle Klassen oder nur die eigene zur Verfügung stehen sollen. Auch beim Preprocessing mit SpaCy und SoMaJo hätte ich im Nachhinein Sachen anders gemacht und mehr ausprobiert, wie ich die Verarbeitung beschleunigen kann, z.B. mit *spacy.nlp.pipe* – da ich aber bereits meine grundlegenden Strukturen hatte und wieder verändern hätte müsse, habe ich mich dagegen entschieden und stattdessen notgedrungen *tqdm* genutzt, um immerhin anzuzeigen, wie viel noch verarbeitet werden muss.
Allgemein nehme ich daher mit, mich früher und ausführlich mit der allgemeinen Struktur des Programms auseinanderzusetzten, bevor ich „irgendwie“ loslege, um möglichst viel abzuarbeiten.

Das Review von Angelina hat mir vor allem dabei geholfen, Kommentare und Encapsulation nicht vollkommen zu vergessen und auch gezeigt, was für die Reviewerin nicht offensichtlich war – nach so langem Arbeiten fällt einem so etwas meist ja nicht mehr aus – im Nachhinein aber gut verstehe und versucht habe, noch zu verbessern.