Projektbericht zum Modul Data Mining Wintersemester 20221/2022

Reproduktion des Papers *Context-Sensitive Visualization of Deep Learning Natural Language Processing Models*[1]

Max Henze

14. März 2022

1 Einleitung

Neuronale Netzwerke sind ein beliebtes Hilfmittel im Bereich von NLP. Besonderst Modelle, welche sich mit dem Einsatz von Transformern , wie BERT [2] oder GPT-2 [3], behelfen, gehören schon lange zum state-of-the-art. Doch diese Modelle mit ihrer Vielzahl an Layern, Neuronen und Verbindungen, gewähren einen nicht gerade einfachen Einblick in ihre Verarbeitungsschritte. Ansätze wie Leave-One-Out oder Leave-N-Out versuchen durch Neuklassifikation von modifizierten Texten einen allgemeinen Enblick in die Verarbeitungsstrukturen von Neuronalen Netzwerken zu geben. Diese Ansätze betrachten aber nicht einen möglichen Kontext, wie er zwischen Wörtern in einem Satz existieren kann. Dunn et al. haben daher in ihrem Artikel „Context-Sensitive Visualization of Deep Learning Natural Language Processing Models“ eine Methode entwickelt um Wörter, mit ihrer unterschiedlichen Wichtigkeitsgewichtung innerhalb der Klassifizierung und unter Inbetrachtnahme von Kontext zu visualisieren. Als Replikationsziele wurden für diese Arbeit der gesamte Visualisierungsprozess von Dunn et al. gewählt. Zusätzlich dazu wurde ein eigenes BERT Modell auf dem gegebenen IMDB [4] Datensatz trainiert um dieses für den späteren Klassifikationsprozess zu verwenden. Darüberhinaus wurden zwei zusätzliche Methodiken zur Visualisierung mit Kontext überprüft, welche in ihrere Effektivität jedoch hinter der von Dunn et al. stehen. Durch die Replikation wird eine verständliche Codebeigabe zum Originalartikel erzeugt, welche dem Leser ein noch besseres Verständnis liefern soll. So können mit

dem System alle Dokumente des Testdatensatzes klassifiziert und visualisiert werden um so eine größere Vielfalt von Beispielen und ein besseres Verständnis bereitzustellen.

2 Umfang der Replikation/Reproduktion

Als Ziel dieser Replikation wurde die einzige Hypothese gewählt, welche Dunn et al. in ihrem Artikel behandeln. Sie propagieren, dass sich die Wichtigkeit eines Wortes, innerhalb der Klassifikation durch ein Neuronales Netzwerk, nicht nur durch den Vergleich der sogenannten Prediction Strength (Sicherheit des NN, dass Label richtig Klassifiziert) des Originaltextes zur Pediction Strength des Textes ohne das betrachtete Wort (Leave-One-Out) ergibt, sondern dass der Einbezug von kontextuell zusammhängenden Wörtern (Leave-N-Out) ebenfalls wichtig ist. „Unser Ansatz schaut auf die Kombination von Wörtern und Sätzen um deren Einfluss auf die Ausgabe des Modells zu erkennen, was zu einer Visualisierung führt, welche kontextsensitiver zum Originaltext ist. “ [1]

Somit ist folgende Behauptung das Ziel dieser Replikation:

- Leave-N-Out Ansatz ist geeigneter bei der Erkennung kontextueller Wörter und Strukturen als Leave-One-Out.

3 Methoden

Die Replikation des Originalartikels ergibt sich wie folgt. Durch die fehlende Beigabe von Code mussten alle Ideen und Modelle von Dunn et al. eigenständig implementiert werden. Dazu wurde sich an Wortangaben der Autoren wie zum Beispiel: „Der gesamte Code ist geschrieben in Python 3.8 und nutzt die Tensorflow Version der Transformersbilbiothek. [...] Texttokenisierung und Abhängigkeitsbestimmung wurden mit der spaCy NLP Bibliothek durchgeführt.“ [1] gehalten.

Zur Klassifizierung von Dokumenten wurde ein Modell unter der Verwendung von BERT trainiert. Da keine weitere Angaben zu finden waren und eine große Auswahl an unterschiedlichen BERT Modellen zu finden ist wurde das *BERT uncased L-12 H-768 A-12* Modell gewählt, welches auf Tensorflow Hub[1] zu den am häufigsten verwendet BERT Modellen zählt. (über 214.000 Downloads)

Entwickelt wurde innerhalb eines Jupyter Notebooks mit Python. Die folgenden essentiellen Packages fanden dabei Anwendung:

[1] `https://tfhub.dev`

Package Name	Package Funktion
tensorflow_hub	Einbindung des BERT Modells
tensorflow	Modellerzeugung und Training
official.nlp	Trainingsoptimisierung
spacy	Abhängigkeitsbestimmung (Dependency Parsing)
pandas	Arbeiten mit Dataframes
matplotlib	Visualisierung der Texte

Abbildung 1: Verwendete Packages

Das BERT Modell auf einer Nvidia Geforce RTX 3070 mit 8 GB Arbeitsspeicher trainiert.

3.1 Modellbeschreibung

Innerhalb des Originalartikels sind keine Angaben bezüglich der Zielfunktion und Parameter zu finden. Angaben zum Modell beruhen auf der Benennung eines BERT Modells und einer Modellbeschreibung, welche auf das Anhängen eines Dropout-Layers und Dense-Layers verweist.

Die beschriebene Methodik ist wie folgt:

Ein Text wird durch das Modell klassifiziert und die damit korrespondierende Ausgabestärke, der Score, wird notiert. Nun werden mit Hilfe einer Abhängigkeitsbestimmung alle Beziehungen zwischen Wörtern aufgedeckt. Anschließend werden neue Texte erzeugt, in denen jeweils ein Wortpaar, welches eine Verbindung zueinander aufweist, entfernt wurde. Die nun erhaltene Sammlung an neuen Texten wird wieder durch das Modell klassifiziert und die neuen Ausgabestärken werden mit der des Ausgangstextes verglichen. Texte mit größeren oder gleichen Ausgabestärken als der des Originals tragen scheinbar nicht zur Klassifikation bei und werden entfernt. Dies geht mit unserer Intuition einher wie das folgende Beispiel erklärt.

Nehmen wir an der Satz *I love this film so much.* wurde durch das Modell mit einem Score von 0.9 bewertet. Das Neuronale Netzwerk ist sich somit sehr sicher, dass dieser Satz das Label 1, also positiv bekommen sollte. Nehmen wir nun an, dass wir den Text so modifizieren: *I love film so much.* und dass das Modell nun einen Score von 0.95 vergibt. Durch das weglassen von *this* ist die Sicherheit, dass es sich hier um ein positives Label handelt, gestiegen. Somit lässt sich annehmen, dass *this* keinen Beitrag zur Klassifikation des Labels leistet. Somit können wir es von unseren Betrachtungen entfernen.

Je größer nun eine Differenz ist, umso wichtiger war das Wortpaar für die Klassifizierung. Mit Hilfe einer Linearisierung der Differenzen und einer Colormap können Wörter somit bezüglich ihrer Wichtigkeit farblich kenntlich gemacht werden. Je wichtiger umso grüner, je unwichiger, desto blauer.

3.2 Datenbeschreibung

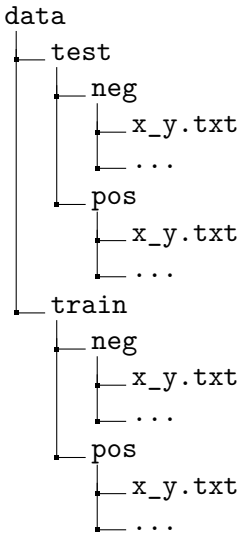


Abbildung 2: Ordnerstruktur des Datensatzes. x ist die Dokumentenid und y ist eine Sternewertung von Null bis Zehn.

Der im Originalartikel und dieser Replikation verwendete Datensatz ist das Large Movie Review Dataset [4] der Universität Stanford. Dieser umfasst 50.000 Dokumente, darunter 25.000 Trainingsdokumente und 25.000 Testdokumente. Er ist unter `https://ai.stanford.edu/~amaas/data/sentiment/` verfügbar.

Der Datensatz hat eine vorgegebene Ordnerstruktur, siehe Abbilung 2. So befinden sich die Trainingsdokumente und Testdokumente in eigenen Ordnern, wobei positive und negative Dokumente nochmals in eigene Ordner unterteilt sind. Die Dokumente unterscheiden sich stark in der Länge, so gibt es Dokumente mit knapp über 50 Zeichen aber auch solche mit über 13.000 Zeichen. Die Dokumente an sich sind nicht aufbereitet, enthalten englische Altagssprache und Sonderzeichen.

> Fair drama/love story movie that focuses on the lives of
> blue collar people finding new life thru new love.The acting
> here is good but the film fails in cinematography,screenplay,
> directing and editing.The story/script is only average at best.
> This film will be enjoyed by Fonda and De Niro fans and by
> people who love middle age love stories where in the coartship
> is on a more wiser and cautious level.
> It would also be interesting for people who are
> interested on the subject matter regarding illiteracy.....

Abbildung 3: Beispieltext eines positiven Trainingsdokuments

Bei der Verwendung der Daten zum Training des Modells, wurde der Trainingsdatensatz zusätzlich in einen Validierungsdatensatz aufgeteilt. Dieser umfasst 20 Prozent der Trainingsdaten und somit 5.000 Dokumente.

3.3 Hyperparameter

Folgende Hyperparameter wurden gesetzt:

Parameter	Wert
Batch Size	16
Epochs	1
Learningrate	3e-5
Dropoutrate	10%

Die Batch Size definiert die Menge an Trainingsdaten, welche von Netzwerk aufeinmal verarbeitet werden bevor es sich aktualisiert. Diese wurde auf 16 festgesetzt, da das Modell auf der zuvor schon erwähnten Nvidia Grafikkarte trainiert werden sollte. Größere Batch Sizes haben sich als Problem entpuppt, da diese nicht mehr in den Speicher passten. Bei der Epochenanzahl wurde nur eine festgelegt. Beginnend lag dieser Wert bei fünf. Doch verschiedene Durchläufe und eine Einbindung von Early Stopping ergaben, dass das Modell nach der ersten Epoche die beste Leistung aufwies. Mit zunehmenden Training stieg auf dem Trainingsdatensatz die Accuracy und im selben Moment sank der Loss. Doch auf dem Validierungsdatensatz stieg der Loss bei gleichbleibener Accuracy in jeder Epoche. Dies war ein eindeutiges Zeichen für Overfitting.

3.4 Implementierung

Der Code zur Implementierung ist abrufbar unter: `https://github.com/maxhenze/Klausurleistung.git` Die wichtigsten verwendeten Packages finden sich in Abbildung 1.

Durch die Verwendung eines Jupyter Notebooks ist der Code interaktiv gehalten. Parameter können angepasst werden und dadurch erzeugte Modelle können sofort neu trainiert werden, falls die Hardware dies zulässt. Falls nicht sind im Projekt zwei fertige Modelle vorhanden, welche eigenständig trainiert wurden.

Diese können im Notebook geladen und verwendet werden. Der entsprechende Flag muss gesetzt werden ob ein Modell trainiert oder geladen werden soll. Je nachdem werden ungebrauchte Codeteile übersprungen.

Der Datensatz wird automatisch heruntergeladen und entpackt, je nachdem ob Daten schon vorhanden sind.

Die Einteilung der Trainingsdaten in Training- und Validierungsmenge wird durch einen Seed bestimmt. Durch unterschiedliches setzen werden unterschiedliche Daten zum Training bzw. zur Validierung benutzt. Hauptsächlich ist dieser Wert aber dazu da, damit kein Dokument in beiden Datensätzen auftaucht.

Ein anderes BERT Modell kann ebenfalls geladen werden, dazu muss nur der passende Link von Tensorflow Hub für die Variable `tfhub_handle_encoder` ersetzt werden.

Bei dem zuvor schon erwähnte Optimierer handelt es sich um den AdamW [5] Optimierer, welcher die Parameter des Modells dynamisch während des Lernprozesses anpasst.

3.5 Aufbau der Experimente

Zur Durchführung der Experimente des Originalartikels wurden die Dokumente aus der Testdatemenge verwendet. Eine Zelle des Notebooks hat dabei die Aufgabe ein zufälliges Dokument zu wählen. Durch das Nacheinanderausführen der dahinter liegenden Zellen werden die Klassifikations und weiteren Rechenschritte zur Visualisierung automatisch abgearbeitet und man erhält am Ende ein fertiges Bild des eingefärbten Textes. Dabei wird auf der einen Seite der Leave-One-Out und auf der anderen der Leave-N-Out Ansatz durchlaufen, so dass am Ende zwei Texte herauskommen, welche miteinander verglichen werden können.

4 Ergebnisse

4.1 Ergebnis 1

```
1250/1250 [==============================] - 418s 325ms/step - loss: 0.3566 -
binary_accuracy: 0.8321 - val_loss: 0.2882 - val_binary_accuracy: 0.8770
```

Abbildung 4: Trainingsergebnisse des Klassifikationsmodells mit BERT und angehängtem Dropout und Dense Layer.

```
1563/1563 [==============================] - 172s 110ms/step - loss: 0.2732 -
binary_accuracy: 0.8820
```

Abbildung 5: Testsergebnisse des Klassifikationsmodells mit BERT und angehängtem Dropout und Dense Layer.

Die Replikation des Klassifikationsmodells ergab nach einer Trainingsepoche ein Modell mit einer Accuracy von 0.877 und einem Loss von 0.2882. Es wird also ein Großteil der Daten richtig klassifiziert. Diese Ergebnisse konnten innerhalb der Testdaten bestätigt werden, siehe Abbildung 4 und 5.
Training mit größerer Epochenanzahl ergab keine Verbesserung der Acurracy, aber einer Steigerung des Losses. Dies deutet auf Overfitting hin.

4.2 Ergebnis 2

Die Ergebnisse des Originalartikels konnten durch die Replikation bestätigt werden. Leichte Abweichungen ergeben sich in der genauen Einfärbung der Wörter. Dies lässt sich auf ein unterschiedliches Klassifikationsmodell, welches leicht abweichende Werte produziert oder ein abweichendes Farbschema. Ebenso gibt es Beispiele des Originalartikels, welche falsch vom Modell klassifiziert werden.

Die folgenden Beispiele sind die selben wie im Orginialartikel, welche jeweils mit dem Leave-One-Out und Leave-N-Out Ansatz visualisiert wurden.

Abbildung 6: Wie im Originalartikel wurde hier mit dem leave-n-out der Zusammenhang von *best* und *movies* besser gekennzeichnet. Im Gegensatz zum Originalartikel hingegen, wurde hier bei beiden das Wort *created* als beeinflussend gekennzeichnet.

Abbildung 7: In diesem Beispiel konnten die kontextuellen Zusammenhänge von *boring* und *film*, sowie *bad* und *acting* durch den leave-n-out Ansatz besser verdeutlicht werden.

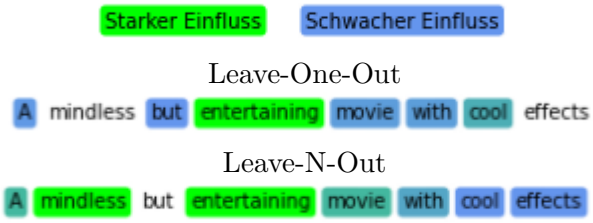


Abbildung 8: Auch in diesem Beispiel findet sich eine Übereinstimmung zum Originalartikel. *mindless* wird hier mit *entertaining* stärker in Beziehung gesetzt.

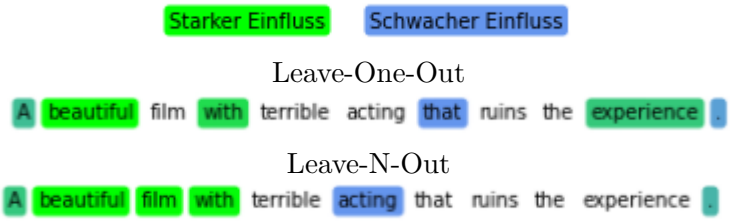


Abbildung 9: Dieses Beispiel bietet gar keine Übereinstimmung mit dem Originalartikel. *terrible* und *ruins* werden gar nicht in Betracht gezogen. Der Fehler scheint hier beim Modell zu liegen. Es klassifiziert bei diesem Beispiel das falsche Label. Vermutlich aufgrund der Einleitung mit *a beautiful film*.

Wie in den Abbildungen 6 bis 8 zu sehen ist, sind die Ergebnisse nahezu äquivalent zu denen des Originalartikels. Bei denen wo das Klassifikationsmodell richtig liegt lässt sich auch die spätere Visualisierung des Originalartikels, in leicht abweichenden Farbnuancen, replizieren. Das Experiment wo das Modell schon im Vorhinein scheitert, siehe Abbildung 9, ergibt auch eine stark abweichende Visualisierung.

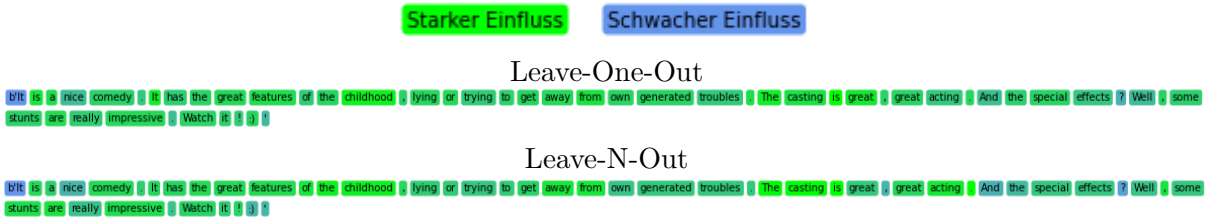


Abbildung 10: Verlgeich längerer Texte. Veränderte Einflüsse fallen hier nur in Nuancen auf.

Bei längeren Texten fallen die veränderten Zusammenhänge nicht sofort auf. Ebenso fällt bei längeren Texten auf, dass oft nur einzelne Wörter vom Modell als wichtig interpretiert werden. Wodurch sich Visualisierungen wie in Abbildung 11 ergeben.

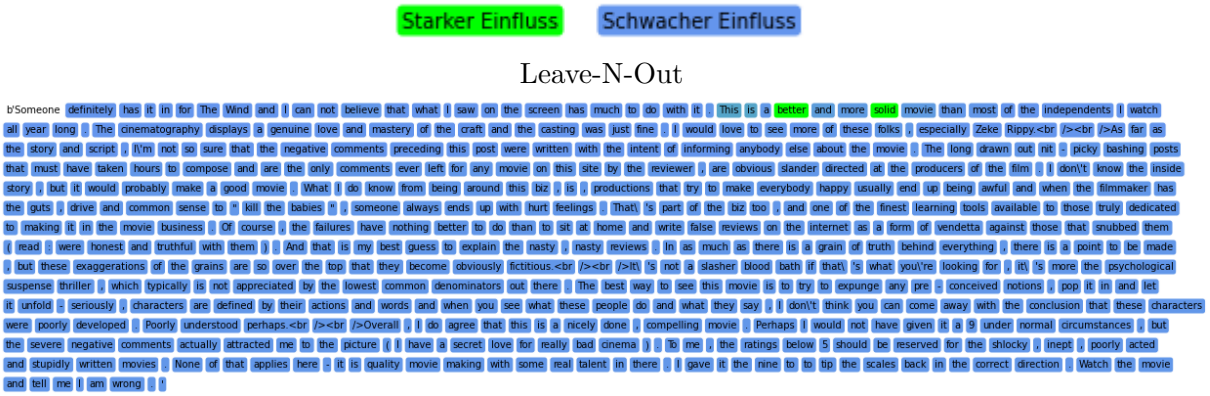


Abbildung 11: Visualisierung langer Texte. Einzelne Wörter fallen hier stark ins Gewicht, wohingegen alle anderen Wörter als nicht stark beeinflussend gekennzeichnet werden.

4.3 Zusätzliche Ergebnisse, die nicht im Originalartikel enthalten waren

Zusätzlich zu den im Experimenten des Originalartikels wurde zwei weitere Methodiken des Leave-N-Out Ansatzes implementiert.

Ersteres versucht das Weglassen eines Elternwortes mit allen von ihm abhängigen Kindern. Dabei wird deutlich, dass hierbei zu viele Wörter entfernt werden, wodurch Abhängigkeiten schlechter dargestellt werden. Diese Methodik ähnelt in der Visualisierung stark der Leave-One-Out Methodik.

Letzteres hingegen versucht zusätzlich zum Elternwort, alle Nachfahren dieses Wortes weggelassen. Somit werden auch implizite Verbindungen von Wörtern in Betracht gezogen, welche nicht durch eine direkte Abhängigkeit miteinander verbunden sind. Diese Methodik erzeugt ähnliche Darstellungen wie die Leave-N-Out Methodik des Originalartikels, wobei direkte Abhängigkeiten etwas schwächer visualisiert werden. Indirekte werden hingegen hervorgehoben, wodurch größere Teilabschnitte bzgl. der Farbgebung zusammengefasst werden. Dies ist sichtbar in Abbildung 12

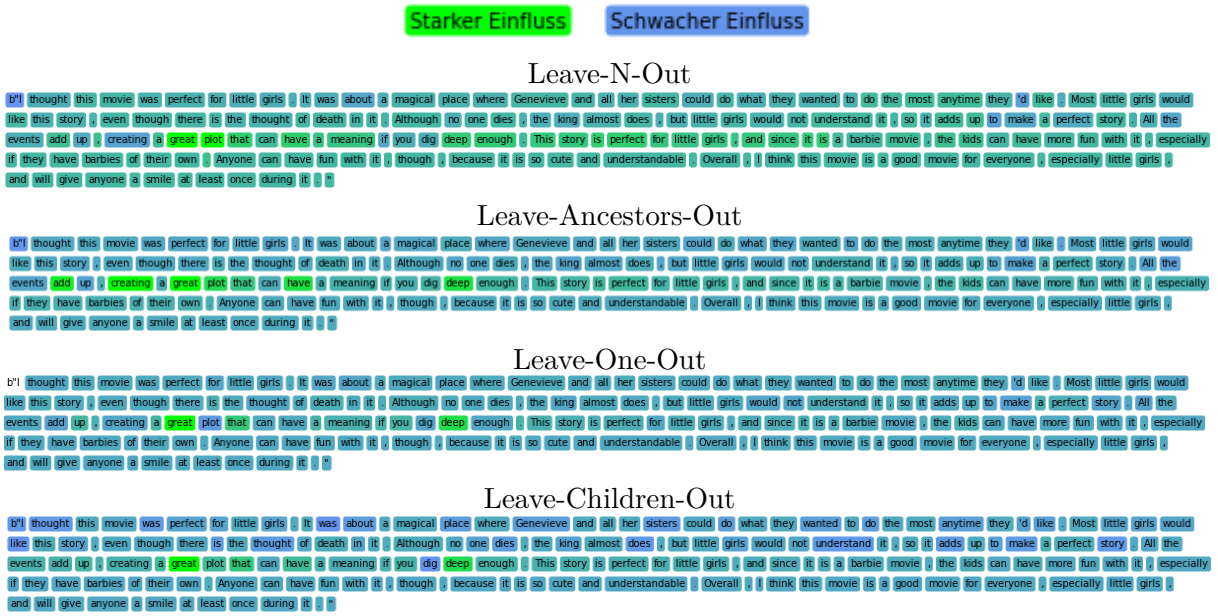


Abbildung 12: Visualisierung eines Textes mit zusätzlich erdachten Methodiken. In dieser Abbildung werden die Methodiken der Leave-N-Out und Leave-One-Out des Originalartikels mit den selbsterdachten Methodiken Leave-Ancestors-Out und Leave-Children-Out verglichen.

5 Diskussion

Der im Originalartikel propagierte Leave-N-Out Ansatz schafft es im Vergleich zum Leave-One-Out Ansatz, kontextuell abhängige Wörter bei der Visualisierung besser in Bezug zu setzen. Die Frage die sich stellt, ist jene, ob die dadurch in Bezug gesetzten Wörter einen besseren Einblick in die Klassifikation eines Neuronalen Netzwerkes ermöglicht ? Natürlich ist es bei Sätzen wie *One of the best movies ever created.* richtig, dass die Wörter *best* und *movie* durch ihren kontextuellen Zusammenhang eine gleichwertige Wichtigkeitsgewichtung erhalten sollten. Dennoch wird durch den Leave-One-Out Ansatz deutlich, dass das Neuronale Netzwerk das Wort *best* stärker für die Klassifizierung benutzt als das Wort movies. Dies gibt eine stärkere Einsicht in die Wichtigkeit eines einzelnen Wortes, wohingegen beim Leave-N-Out Ansatz unklar ist ob durch das Weglassen beider Wörter die Klassifizierung schlechter geworden ist oder nur durch das Weglassen des stärker klassifizierten Wortes.

Der Originalartikel war trotz fehlenden Codes replizierbar. Die Aussagen der Autoren gaben einen guten Einblick in die verwendeten Strukturen und Techniken, welche mit Hilfe von state-of-the-art Modellen gut umgesetzt werden konnten. Dennoch sind leichte Abweichungen, durch fehlende Angaben zum BERT Modell, vorhanden. Ebenso ist eine abweichende Farbskala ein ebenso möglicher Grund für leicht verschiedene Darstellungen.

Es wurde deutlich, dass der Originalansatz im Vergleich zu den selbst dargestellten Methodiken, eine bessere Visualisierung von kontextuell abhängigen Wörtern ermöglicht.

5.1 Was war einfach ?

Aufgrund des recht einfachen, propagierten Ansatzes, waren nahezu alle Umsetzungen der Leave-N-Out Methodik nicht schwer zu implementieren. Durch klare Package Benennung und einer einfachen Algorithmusbeschreibung, konnten alle Schritte von der Klassifikation, dem produzieren neuer Text, der Filterung der Neuklassifikation und der Visualisierung problemlos repliziert werden. So benötigte das Einlesen in die API des jeweiligen Packages eine gewisse Zeit, dennoch handelte es sich dabei nie um zu umfängliche Beschreibungen.

Das angegebene spacy Package ist ein sehr gutes Tool für die Erstellung von Abhängigkeitsgraphen auf denen sich recht einfach navigieren lässt, wodurch die Abhängigkeitsbestimmung dadurch auch nicht besonderst schwer viel. Und die Bearbeitung auf den neu generierten Texten viel ebenfalls, durch das sehr umfängliche pandas Package, sehr leicht aus.

5.2 Was war schwer ?

Die Implementierung des Klassifikationsmodells, sowie der allgemeine Umgang mit diesem, war nicht sehr einfach. Durch fehlende Angaben der Autoren zum Modell war beginnend eine gewisse Recherche notwendig, welches BERT Modell nun genau benutzt werden soll. Es wurde zwar grob eine Implementierung der verschiedenen Layer erwähnt, diese war aber nicht ausreichend für die eigenständige Erstellung. Angaben der Daten waren ebenfalls recht sporadisch, wodurch erst eine geeignete Implementationsstruktur erdacht werden musste. Auch die Tatsache, dass keine Angaben zu Hyperparametern gemacht wurden, trug erschwerend bei. Das finden der richtigen Batch Size und das Einstellen des Trainings auf der Grafikkarte erwies sich als äußert umständlich, da zu große Batch Sizes für einen `Out Of Memory` Error sorgten und das Training auf einer CPU (8 Kerne) dennoch zu langsam war.

5.3 Empfehlungen für die Replizierbarkeit

Natürlich wäre ein, von vornherein beigereichter, Code sehr praktisch gewesen. Wobei sich auch hier sagen lässt, dass das Verstehen von fremden Code eine Problematik in sich ist. Genauere Angaben zur Implementierung würden hier schon reichen. Eine genaue Angabe des verwendeten BERT Modells sowie eine Bereichung der verwendeten Hyperparameter und einer Hardwarebeschreibung würde viel Testen obsolet machen. Zusätzlich dazu, auch wenn nicht unbedingt von Nöten, wäre eine Angabe von verwendeten Funktionen der Packages, welche zur Realisierung des Algorithmus verwenden wurden, sehr bequem.

6 Kommunikation mit den Autoren

7 Anhang

7.1 Notebook Code

main

March 14, 2022

1 Used Packages

[ ]:
```
import os
import shutil
import random

import pandas as pd
import spacy
from sklearn.preprocessing import MinMaxScaler
import numpy as np
import tensorflow as tf
import tensorflow_text as text
import tensorflow_hub as hub
from official.nlp import optimization
from IPython.display import display, HTML

import matplotlib.pyplot as plt
import matplotlib
```

[ ]:
```
css = """
.output {
    display: flex;
    flex-direction: row;
}
"""

HTML('<style>{}</style>'.format(css))
```

[ ]:
```
<IPython.core.display.HTML object>
```

[ ]:
```
cmap_org = matplotlib.colors.LinearSegmentedColormap.from_list("",␣
↪["cornflowerblue","lime"], gamma=0.75)
cmap_new = plt.get_cmap("YlOrRd")
```

2 Model Creation

[ ]:
```
PATH = 'data'
# set this parameter if you want to train another model
TRAIN_NEW_MODEL = True
NEW_MODEL_NAME = 'imdb3'
```

2.1 Dataset

[ ]:
```
# downloading the imdb dataset (if not already done)
# removing the unnecessary unsup folder because this is a supervised ml task
if not os.path.isdir('data'):
    url = "https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz"

    dataset = tf.keras.utils.get_file("aclImdb_v1", url,
                                      untar=True, cache_dir=PATH,
                                      cache_subdir='')

    shutil.rmtree('unsup')
```

[ ]:
```
# setting the directory for the training and test data
train_dir = os.path.join(PATH, 'train')
test_dir = os.path.join(PATH, 'test')
```

2.1.1 Dataset Parameters

[ ]:
```
# setting model parameters
# autotune allows the automatic setting of the number of prefetched data ahead
# of time they are requested in the learning process
AUTOTUNE = tf.data.AUTOTUNE
batch_size = 16
epochs = 1
seed = 42
init_lr = 3e-5
```

2.1.2 Splitting Dataset

[ ]:
```
# training set 80 percent of all files with 20 left for validation
raw_train_ds = tf.keras.utils.text_dataset_from_directory(
    train_dir,
    batch_size=batch_size,
    validation_split=0.2,
    subset='training',
    seed=seed)

class_names = raw_train_ds.class_names
train_ds = raw_train_ds.cache().prefetch(buffer_size=AUTOTUNE)
```

```
raw_val_ds = tf.keras.utils.text_dataset_from_directory(
    train_dir,
    batch_size=batch_size,
    validation_split=0.2,
    subset='validation',
    seed=seed)

val_ds = raw_val_ds.cache().prefetch(buffer_size=AUTOTUNE)

raw_test_ds = tf.keras.utils.text_dataset_from_directory(
    test_dir,
    batch_size=batch_size)

test_ds = raw_test_ds.cache().prefetch(buffer_size=AUTOTUNE)
```

```
Found 25000 files belonging to 2 classes.
Using 20000 files for training.
Found 25000 files belonging to 2 classes.
Using 5000 files for validation.
Found 25000 files belonging to 2 classes.
```

2.2 Model Training

[ ]:
```
if TRAIN_NEW_MODEL:
    # setting the bert encoder and preprocessor
    tfhub_handle_encoder = 'https://tfhub.dev/tensorflow/
 ↪bert_en_uncased_L-12_H-768_A-12/4'
    tfhub_handle_preprocess = 'https://tfhub.dev/tensorflow/
 ↪bert_en_uncased_preprocess/3'

    # generating the bert encoder and preprocess layer for the model
    # (save model error can be fixed by deleting temp folder)
    bert_preprocess_model = hub.KerasLayer(tfhub_handle_preprocess)
    bert_model = hub.KerasLayer(tfhub_handle_encoder)

    # function for building the classifiert model
    # text input -> preprocessing -> encode -> droput -> dense

    def build_classifier_model():
        text_input = tf.keras.layers.Input(shape=(), dtype=tf.string,␣
 ↪name='text')
        preprocessing_layer = hub.KerasLayer(tfhub_handle_preprocess,␣
 ↪name='preprocessing')
        encoder_inputs = preprocessing_layer(text_input)
        encoder = hub.KerasLayer(tfhub_handle_encoder, trainable=True,␣
 ↪name='BERT_encoder')
        outputs = encoder(encoder_inputs)
```

```
        net = outputs['pooled_output']
        net = tf.keras.layers.Dropout(0.1)(net)
        net = tf.keras.layers.Dense(1, activation=None, name='classifier')(net)
        return tf.keras.Model(text_input, net)

    # initialize classifier model
    classifier_model = build_classifier_model()

    # set loss and metric functions
    loss = tf.keras.losses.BinaryCrossentropy(from_logits=True)
    metrics = tf.metrics.BinaryAccuracy()

    # create model hyperparameter optimizer
    steps_per_epoch = tf.data.experimental.cardinality(train_ds).numpy()
    num_train_steps = steps_per_epoch * epochs
    num_warmup_steps = int(0.1*num_train_steps)

    optimizer = optimization.create_optimizer(init_lr=init_lr,
                                              num_train_steps=num_train_steps,
                                              num_warmup_steps=num_warmup_steps,
                                              optimizer_type='adamw')

    # early stopping
    # early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss')

    # compile the model
    classifier_model.compile(optimizer=optimizer,
                             loss=loss,
                             metrics=metrics)

    # train the model
    history = classifier_model.fit(x=train_ds,
                                   validation_data=val_ds,
                                   epochs=epochs)
```

```
1250/1250 [==============================] - 390s 301ms/step - loss: 0.3586 -
binary_accuracy: 0.8336 - val_loss: 0.2872 - val_binary_accuracy: 0.8800
```

[ ]:
```
if TRAIN_NEW_MODEL:
    loss, accuracy = classifier_model.evaluate(test_ds)
```

```
1563/1563 [==============================] - 172s 110ms/step - loss: 0.2773 -
binary_accuracy: 0.8805
```

[ ]:
```
if TRAIN_NEW_MODEL:
    saved_model_path = f'./models/{NEW_MODEL_NAME}_bert'
    classifier_model.save(saved_model_path, include_optimizer=False)
```

```
WARNING:absl:Found untraced functions such as restored_function_body,
```

```
restored_function_body, restored_function_body, restored_function_body,
restored_function_body while saving (showing 5 of 915). These functions will not
be directly callable after loading.

INFO:tensorflow:Assets written to: ./models/imdb3_bert\assets

INFO:tensorflow:Assets written to: ./models/imdb3_bert\assets
```

3 Loading the Model and working with it

[ ]:
```
if not TRAIN_NEW_MODEL:
    saved_model_path = f'models/{os.listdir("models")[1]}'
    model = tf.keras.models.load_model(saved_model_path, compile=False)
else:
    model = classifier_model
```

3.0.1 Retrieve Complete Test Data

[ ]:
```
test_data_unbatched = list(test_ds.unbatch().as_numpy_iterator())
```

3.0.2 Select Random Pair

[ ]:
```
rand_ind = random.randint(0, len(test_data_unbatched))
rand_sen_label_pair = test_data_unbatched[rand_ind]
rand_sen_label_pair[0]
# rand_sen_label_pair = ('Just a boring and uninspired film filled with bad␣
 ↪acting.', 0)
```

[ ]:
```
b"What looks like a ho-hum Porky's rip-off turns out to be quite a touching film
about being young and in love.<br /><br />The story concerns three friends,
Gary, Ricky and David, who spend their after school hours looking for sex. When
a new girl arrives in town Gary falls head over heels in love with her.<br /><br
/>The film goes from being a sleazy sex film to an examination of teenage
insecurities. It is funny and sad at the same time. It never completely gives
into that love story formula that seems prominent in every movie made. You know
the guy meets girl, guy loses girl, guy gets girl back in the final frame
formula. That formula is tossed aside after guy meets girl. Maybe that is why I
liked the film so much.<br /><br />The soundtrack is especially good and the
ending is a definite tear jerker. It also might be one of the most realistic
endings I've ever seen in a love story.<br /><br />"
```

3.0.3 Function for Text Classification

[ ]:
```
def classify_text(model, text, parent_ind=None, child_ind=None):
    '''
    Function to predict a given text given a model.
    The predicted score is furthermore transformed into the corresponding
    label.
```

```
    '''
    score = tf.sigmoid(model(tf.constant([text])))[0][0].numpy()
    pred_label = np.where(score > 0.5, 1, 0).item()

    return (score, pred_label, parent_ind, child_ind)
```

3.0.4 Predict Random Pair Label and Score

[ ]:
```
org_text_pred = classify_text(model, rand_sen_label_pair[0])
print(f'Predicted Label: {org_text_pred[1]} \nScore: {org_text_pred[0]} \nReal␣
 ↪Label: {rand_sen_label_pair[1]}')
```

```
Predicted Label: 1
Score: 0.9611169099807739
Real Label: 1
```

3.0.5 Dependecy Parsing

[ ]:
```
depend_parser = spacy.load('en_core_web_sm')
```

[ ]:
```
parsed_text = depend_parser(str(rand_sen_label_pair[0]))
sentence_spans = list(parsed_text.sents)[0]
```

[ ]:
```
spacy.displacy.render(sentence_spans, jupyter=True, options={"compact": True})
```

```
<IPython.core.display.HTML object>
```

3.0.6 Functions for Text Generation

[ ]:
```
def leave_n_out(text):
    '''
    Function for generating texts from an original text, where every
    text is missing a different parent-child-word-combination of
    the original text.

    Go over all words, if a word has children, then for every parent-child-pair
    return a text with both removed.
    '''
    leave_n_out_texts = []

    for parent_to_remove in text:
        child_list = [child for child in parent_to_remove.children]

        if child_list:
            for child_to_remove in child_list:
                new_text = []
```

```
                for word in text:
                    if (word.i != parent_to_remove.i and
                        word.i != child_to_remove.i):
                            new_text.append(word.text)

                leave_n_out_texts.append(
                    (" ".join(new_text), parent_to_remove.i, child_to_remove.i))

    return leave_n_out_texts


def leave_one_out(text):
    '''
    Function for generating texts from an original text, where
    every text is missing one different word of the original text.
    '''
    leave_one_out_texts = []

    for word_to_remove in text:
        new_text = []

        for word in text:
            if word_to_remove.i != word.i:
                new_text.append(word.text)

        leave_one_out_texts.append(
            (" ".join(new_text), word_to_remove.i, None))

    return leave_one_out_texts

def leave_childs_out(text):
    '''
    Function for generating texts from an original text, where
    every text is missing a prent word and all of it's children.
    '''
    leave_childs_out_texts = []

    for word_to_remove in text:
        new_text = []
        child_ids = [child.i for child in word_to_remove.children]

        for word in text:
            if word_to_remove.i != word.i and word.i not in child_ids:
                new_text.append(word.text)

        leave_childs_out_texts.append(
            (" ".join(new_text), word_to_remove.i, child_ids))
```

```
    return leave_childs_out_texts

def leave_n_ancestors_out(text):
    '''
    Function for generating texts from an original text, where every
    text is missing a different parent-ancestor-word-combination of
    the original text.

    Go over all words, if a word has ancestors, then for every␣
 ↪parent-ancestor-pair
    return a text with both removed.
    '''
    leave_n_out_texts = []

    for parent_to_remove in text:
        ancest_list = [ancest for ancest in parent_to_remove.ancestors]

        if ancest_list:
            for ancest_to_remove in ancest_list:
                new_text = []

                for word in text:
                    if (word.i != parent_to_remove.i and
                        word.i != ancest_to_remove.i):
                            new_text.append(word.text)

                leave_n_out_texts.append(
                    (" ".join(new_text), parent_to_remove.i, ancest_to_remove.
 ↪i))

    return leave_n_out_texts
```

3.0.7 Functions for Further Processing

[ ]:
```
def drop_unimportant_words(df, label, org_score):
    '''
    Function to drop unimportant words.
    An unimportant word is one which increases
    the difference of the classificaion with out it vs. with it .

    For example: "The movie was very good."
    Score: 0.9
    Score without "the": 0.95

    "The" is unimportant, because the classification without it increases.
    '''
```

```
    if label == 1:
        df = df.drop(df.index[df['Score'] >= org_score] )
    else:
        df = df.drop(df.index[df['Score'] <= org_score] )

def calc_score_diff(df, label, org_score):
    '''
    Function to calculate the score differences of the original text to
    those with certain words removed.
    '''
    if label == 1:
        df['Score Difference'] = org_score - df['Score']
    else:
        df['Score Difference'] = df['Score'] - org_score

def create_df(texts):
    '''
    Function to create a dataframe out of a given list of texts.
    The constructed dataframe consists of four columns:
    Score, Predicted Label, Parent Index and Child Index.

    The values for all columns come from the classify_text function.
    '''

    df = pd.DataFrame(
        [classify_text(model, text[0], text[1], text[2]) for text in texts],
        columns=['Score',
                 'Predicted Label',
                 'Parent Index',
                 'Child Index'])

    return df

def linearize_score_diff(df, choose_best_diff=False):
    '''
    Function to linearize the score differences in a given dataframe.
    If choose_best_score flag is set, the best difference for every
    token is choosen.

    For example:
    If the word "good" is a child of the word "movie" and the word "story"
    the best score difference of both pairs is choosen. If "good" happens
    to be a parent with children itself, all those possible pairs are
    considered either for best difference.
    '''
    if choose_best_diff:
        df_copy = df.copy()
```

```
        df_copy[['Child Index', 'Parent Index']
                ] = df[['Parent Index', 'Child Index']]
        df = pd.concat([df, df_copy]).reset_index(drop=True)
        df = df.loc[df.groupby(['Parent Index'])[
            "Score Difference"].idxmax()]

    df['Score Difference'] = MinMaxScaler(
    ).fit_transform(df[['Score Difference']])

    df.drop(
        ["Score", "Predicted Label", "Child Index"],
        inplace=True, axis=1)

    df = df.sort_values(by=['Score Difference'])

    df.rename(columns={"Parent Index": "Token Index"}, inplace=True)

    return df
```

3.0.8 Functions for Visualization

[ ]:
```
def vis_text(df):
    '''
    Function which creates a plt plot without axis of a given text
    with given word importances.

    It draws all words in a row, creating a new one whenever the current
    row is to full. Words are colored given the corresponding importance
    in the dataframe.

    If a word is not given in the dataframe, for example if it was removed
    due to not contributing to the classification, it is colored white.
    '''
    start_x = 20
    start_y = 500
    end = 1200
    whitespace = 8

    figure = plt.figure(figsize=(20, 10))
    rend = figure.canvas.get_renderer()

    for token in parsed_text:
        if df.loc[df["Token Index"] == token.i, 'Score Difference'].values.size␣
↪> 0:
            col = cmap_org(df.loc[df["Token Index"] == token.i, 'Score␣
↪Difference'].values[0])
        else:
```

```
            col = "white"
        bbox = dict(boxstyle="round,pad=0.3", fc=col, ec="white")

        txt = plt.text(start_x, start_y, str(token), bbox=bbox, transform=None)

        bb = txt.get_window_extent(renderer=rend)

        start_x = bb.width + start_x + whitespace

        if start_x >= end:
            start_x = 20
            start_y -= 20

    plt.axis("off")
    plt.show()
```

3.1 Generating New Texts

[ ]:
```
# Original Experiments
new_texts_lno = leave_n_out(parsed_text)
new_texts_loo = leave_one_out(parsed_text)
```

3.2 Generating DataFrames

[ ]:
```
df_lno = create_df(new_texts_lno)
df_loo = create_df(new_texts_loo)

display(df_lno)
display(df_loo)
```

```
        Score  Predicted Label  Parent Index  Child Index
0    0.959754                1             1            0
1    0.947363                1             1            2
2    0.862692                1             1           12
3    0.958654                1             1           26
4    0.953679                1             1           27
..        …               …           …          …
170  0.955708                1           178          176
171  0.955708                1           178          177
172  0.955708                1           178          179
173  0.955708                1           179          181
174  0.955708                1           181          180

[175 rows x 4 columns]
```

```
        Score  Predicted Label  Parent Index Child Index
0    0.951701                1             0        None
1    0.946163                1             1        None
```

```
2    0.954155                1             2        None
3    0.951021                1             3        None
4    0.950034                1             4        None
..        …                  …             …         …
182  0.955708                1           182        None
183  0.955708                1           183        None
184  0.955708                1           184        None
185  0.955708                1           185        None
186  0.955708                1           186        None

[187 rows x 4 columns]
```

3.3 Dropping Unimportant Words

[ ]:
```
drop_unimportant_words(df_lno, org_text_pred[1], org_text_pred[0])
drop_unimportant_words(df_loo, org_text_pred[1], org_text_pred[0])
```

3.4 Calculating Score Differences

[ ]:
```
calc_score_diff(df_lno, org_text_pred[1], org_text_pred[0])
calc_score_diff(df_loo, org_text_pred[1], org_text_pred[0])
```

3.5 Linearizing Score Difference

[ ]:
```
df_lno = linearize_score_diff(df_lno, choose_best_diff=True)
df_loo = linearize_score_diff(df_loo)

display(df_lno)
display(df_loo)
```

```
     Token Index  Score Difference
262           93          0.000000
238           67          0.018510
245           72          0.022629
268           97          0.038391
237           64          0.048408
..           …               …
186            6          0.617579
187            7          0.691251
12            12          0.691251
20            19          1.000000
195           18          1.000000

[186 rows x 2 columns]
```

```
    Token Index  Score Difference
68           68          0.000000
72           72          0.002065
```

```
26          26          0.018727
27          27          0.026063
65          65          0.031610
..         …              …
85          85          0.290025
95          95          0.295534
89          89          0.375238
18          18          0.550888
12          12          1.000000

[187 rows x 2 columns]
```

3.5.1 Additional Experiments

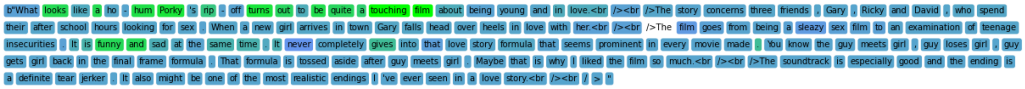
[ ]:
```
# Removing the Parent and all Children before Classification
new_texts_lco = leave_childs_out(parsed_text)
df_lco = create_df(new_texts_lco)
drop_unimportant_words(df_lco, org_text_pred[1], org_text_pred[0])
calc_score_diff(df_lco, org_text_pred[1], org_text_pred[0])
df_lco = df_lco.explode('Child Index')
df_lco = linearize_score_diff(df_lco, choose_best_diff=True)

# Removing the Parent and all Ancesters (implicite Connections) before␣
↪Classification
new_texts_lnao = leave_n_ancestors_out(parsed_text)
df_lnao = create_df(new_texts_lnao)
drop_unimportant_words(df_lnao, org_text_pred[1], org_text_pred[0])
calc_score_diff(df_lnao, org_text_pred[1], org_text_pred[0])
df_lnao = linearize_score_diff(df_lnao, choose_best_diff=True)
```

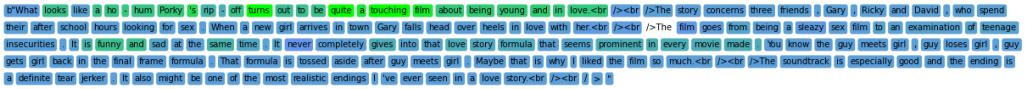
3.5.2 Visualize Texts

[ ]:
```
print('Leave n out:')
vis_text(df_lno)
print('Leave all ancestors out')
vis_text(df_lnao)
print('Leave children out')
vis_text(df_lco)
print('Leave one out:')
vis_text(df_loo)
```

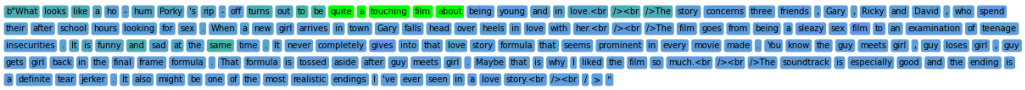
```
Leave n out:
```



Leave all ancestors out



Leave children out



Leave one out: