



# Word2Vec

## Einsatz im Online-Marketing

von

**Silas Dohm** 018102263

**Felix Schumpa** 017200382

**Joel Vongehr** 017200287

Hausarbeit  
im Modul  
„Maschinelles Lernen“

**Eingereicht am:** 9. August 2021

**1. Prüfer:** Prof. Dr. rer. nat. Jörg Frochte

# Inhaltsverzeichnis

<b>1</b>	<b>Abstract</b>	<b>1</b>
<b>2</b>	<b>Motivation</b>	<b>2</b>
<b>3</b>	<b>Grundlagen</b>	<b>3</b>
3.1	Linguistische Datenverarbeitung . . . . .	3
3.1.1	Vorverarbeitung der Sprachdaten . . . . .	3
3.1.2	1-aus-n-Kodierung . . . . .	4
3.1.3	Wortvektoren . . . . .	5
3.1.4	w2v from scratch . . . . .	6
<b>4</b>	<b>Umsetzung</b>	<b>12</b>
4.1	Analyse des Datensatzes . . . . .	12
4.1.1	Datenstruktur . . . . .	12
4.1.2	Anzahl der Reviews . . . . .	15
4.1.3	Reviewlänge . . . . .	15
4.1.4	Distribution der Bewertungen . . . . .	16
4.2	Datenaufbereitung . . . . .	17
4.2.1	Distribution . . . . .	17
4.2.2	Zeichensatz . . . . .	18
4.2.3	Sprachen . . . . .	19
4.3	Generatoren . . . . .	19
4.3.1	Erster Generator . . . . .	20
4.3.2	hdf5 . . . . .	23
4.4	Word2Vec mit Gensim . . . . .	30
4.5	Mean-Vektor-Klassifikationsmodell . . . . .	32
4.5.1	Implementierung . . . . .	32
4.5.2	Konfusionsmatrix . . . . .	36
4.6	Word2Vec-CNN-Modell . . . . .	37
4.6.1	Implementierung . . . . .	37
4.6.2	Konfusionsmatrix . . . . .	42
<b>5</b>	<b>Ergebnisse</b>	<b>43</b>
5.1	Subjektivität . . . . .	43
5.2	Auswertung . . . . .	44
5.2.1	Gewichte . . . . .	44
5.2.2	Mean vs. CNN . . . . .	44
5.2.3	Problemstellung . . . . .	45
5.3	Fazit . . . . .	46

## 1 Abstract

Diese Hausarbeit befasst sich mit der Word2Vec-Technik und ihrem Einsatz im Online-Marketing. Bei dem Datensatz handelt es sich um das *Yelp restaurant review dataset*, welches Rezensionen von über 160000 Unternehmen aus acht Metropolregionen beinhaltet. Bei jeder Rezension vergibt der Verfasser eine bestimmte Anzahl an Sternen die zwischen eins und fünf liegt. In dieser Arbeit werden zwei Verfahren ausgearbeitet, die es ermöglichen, anhand des Bewertungstextes die Anzahl der zugeteilten Sterne vorherzusagen.

Bei dem ersten Verfahren handelt es sich um ein *Mean-Vektor-Klassifikationsmodell*. Hierbei wird der durchschnittliche Wortvektor einer Rezension bestimmt. Mit Hilfe dieses Rezensionsvektors kann die gesamte Bewertung in den zuvor generierten Vektorraum eingeordnet werden. Mittels dieser Einordnung wird nachfolgend die Klassifikation durchgeführt.

Der zweite Ansatz basiert auf einem *Convolutional Neural Network*. Diese Art von neuronalem Netz kommt verstärkt im Feld des maschinellen Sehens zum Einsatz. Durch die Faltungsoperation können CNNs sequenzielle Daten besser verarbeiten, was nicht nur bei Bildern sondern auch bei Texten von Vorteil ist. So kann das Netz Wörter in einem Satz als wiederkehrende Sequenz erkennen und entsprechend gewichten. Aufgrund dieser Gewichtung werden die Rezensionen anschließend klassifiziert.

Da diese Hausarbeit auf den Vorlesungsinhalten des Moduls „Maschinelles Lernen“ von Professor Jörg Frochte basiert, wird nachfolgend lediglich die Verwendung der CNNs motiviert. Die theoretischen Hintergründe zu dieser Kategorie von neuronalen Netzen sind aus der Vorlesung bekannt.

Die beiden Ansätze zur Klassifizierung treffen jeweils eine ca. 80 % genaue Aussage über die Rezensionen, die in positiv, neutral oder negativ eingeordnet werden. Dabei werden die Vorgehensweisen mithilfe der Konfusionmatrizen genauer verglichen.

## 2 Motivation

Möchte man heutzutage die Dienstleistungen eines Unternehmens in Anspruch nehmen, so wird man sich zuerst die Meinungen anderer Kunden durchlesen. Im Hinblick auf diese Kundenbewertungen kann mit Hilfe der Word2Vec-Technik in Kombination mit neuronalen Netzen eine umfassende Aussage über die Art der Rezension getroffen werden. Auf diese Weise kommt Deep Learning im Online-Marketing zum Einsatz.

Wie bereits aus der Vorlesung bekannt ist, kann Deep Learning sehr vielseitigen Zwecken dienen. Interessant ist jedoch, wie es in Sprachanwendungen zum Einsatz kommt. Aus der Philosophie geht hervor: „Die Bedeutung eines Wortes ist sein Gebrauch in der Sprache.“ [WS19] Daraus lässt sich schließen, dass ein einzelnes Wort keinerlei Aufschluss über seine Bedeutung liefert.

Ebenfalls bekannt ist, dass Computer zwar überaus effizient mit Zahlen arbeiten können, mit Wörtern hingegen nicht. Daher muss der alphanumerische Eingabedatensatz zunächst umgewandelt werden, sodass er von der Anwendung verarbeitet werden kann. Bei dieser Neudarstellung der Texte darf die Bedeutung der Wörter nicht untergehen. Die Schwierigkeit liegt also darin, für den Computer Texte so aufzubereiten, dass er sie zum einen verarbeiten kann und zum anderen der Kontext der Wörter nicht verloren geht. Genau mit dieser Herausforderung befasst sich die linguistische Datenverarbeitung.

Um die Ergebnisse abschließend aussagekräftig bewerten zu können, werden, wie einführend bereits erwähnt, zwei Techniken für die Klassifikation vorgestellt. Dabei wird im ersten Ansatz bewusst auf das Verwenden eines neuronalen Netzes verzichtet, um zu zeigen, wie sich Deep-Learning-Techniken im Vergleich zu eher konventionellen Methoden verhalten. Die Wahl fällt hier auf ein *Convolutional Neural Network*.

Der Unterschied zu anderen Feedforward-Netzen steckt im Wort *convolutional*, was auf Deutsch *Faltung* bedeutet. Diese mathematische Faltungsoperation sorgt dafür, dass CNNs nicht nur bei der Bildverarbeitung, sondern generell sequenzielle Daten effizienter verarbeiten. So können „kleine eindimensionale CNNs bei einfachen Aufgaben wie Textklassifizierung [...] eine schnelle Alternative zu RNNs darstellen.“ [Cho18, S. 288] Aufgrund dieser Tatsache wird der Ansatz mit RNNs bereits in der Aufgabenstellung ausgeschlossen.

## 3 Grundlagen

Im Folgenden wird das Basiswissen zur Anwendung der **Word2Vec-Technik** erläutert. Zu Beginn bedarf es Methoden zur Umwandlung von Sprache in eine für Computer lesbare Form. Darauf folgt eine Implementierung der Technik von Grund auf.

### 3.1 Linguistische Datenverarbeitung

Die Technik der **linguistischen Datenverarbeitung** (engl. *Natural Language Processing*, kurz NLP) beschreibt die maschinelle Verarbeitung von menschlicher Sprache. Das NLP erfordert interdisziplinäre Kompetenzen aus sowohl Linguistik und Informatik als auch dem Forschungsfeld der künstlichen Intelligenz. Typische Anwendungen für NLP in der Industrie sind die Klassifikation von Dokumenten, maschinelle Übersetzungen, automatische Vervollständigung von Suchanfragen, Spracherkennung und Chatbots [KBB20, S. 28].

Die Klassifikation von Dokumenten ist eine NLP-Aufgabe mit mittlerer Komplexität. Dazu gehört beispielsweise eine Kritik „in eine bestimmte Kategorie einzuordnen“ [KBB20, S. 28]. Die Verarbeitung von Sprache bedarf einer Darstellung, in der sie von Computern ausgewertet werden kann. Um eine solche quantitative Form zu erreichen, gibt es zwei gängige Methoden. Zum einen die **1-aus-n-Kodierung** (auch *One-Hot-Kodierung*, engl. *One-Hot-Encoding*) und zum anderen **Wortvektoren** (auch *Worteinbettungen*, engl. *Word Embeddings*).

#### 3.1.1 Vorverarbeitung der Sprachdaten

Die Vorverarbeitung eines Datensatzes spielt bei der linguistischen Datenverarbeitung eine wichtige Rolle. Thematisch ist diese Vorverarbeitung an dieser Stelle, also vor der Umwandlung in eine für den Computer nutzbare Form, einzuordnen. Bei den Veranschaulichungen der nächsten Abschnitte des Grundlagenteils werden die Techniken jedoch nur teilweise bis gar nicht angewendet. Erst in dem verwendeten Word2Vec-Modell, welches in Abschnitt 4.4 beschrieben wird, kommen die nachfolgend vorgestellten Methoden am eigentlichen Datensatz zum Einsatz.

Für den Anfang gilt es, die Wörter eines Textes einzeln und unabhängig voneinander abzuspeichern. Dieser Prozess heißt **Tokenisierung**, da jedes Wort als ein **Token** gesehen wird. Im **Textkorpus** befinden sich demnach die tokenisierten Wörter des Rohtextes [KBB20, S. 241].

Im nächsten Schritt geht es darum, alle Großbuchstaben in Kleinbuchstaben umzuwandeln. Gerade im Englischen werden fast ausschließlich Wörter an Satzanfängen

großgeschrieben. Ein kontextueller Unterschied besteht nur in seltenen Ausnahmefällen. Im selben Zuge können auch Satzzeichen entfernt werden, da es für den Computer keinen Unterschied macht, ob ein Wort in einem Teilsatz oder am Satzende steht. Durch das Entfernen von Großbuchstaben und Satzzeichen kann die Größe des Datensatzes etwas verringert werden.

Um den Korpus weiter zu verkleinern, werden anschließend die sogenannten **Stoppwörter** (engl. *stop words*) entfernt. Ein Stoppwort zeichnet sich dadurch aus, dass es zum einen häufig vorkommt und zum anderen eine geringe Bedeutung aufweist. Im Deutschen trifft das beispielsweise auf Präpositionen wie *an* oder *in* zu. Im Englischen können es Wörter wie *the* oder *of* sein. Es gibt keine allgemeingültige Sammlung von Stoppwörtern, da diese immer auch auf den Kontext abgestimmt werden müssen [KBB20, S. 242].

Meist sind diese Techniken schon ausreichend, um den Datensatz bedeutend zu reduzieren und zu bereinigen, ohne jedoch den Kontext zu verfälschen. Es gibt darüber hinaus noch das **Stemming** und die sogenannten **N-Gramme**. Beides sind komplexere Methoden, die in dieser Ausarbeitung allerdings keine Anwendung finden, dennoch erwähnt werden sollten. Ersteres ist „die Zurückführung von Wörtern auf ihren Wortstamm.“ [KBB20, S. 242] Letzteres beschreibt Wörter, die überwiegend zusammen vorkommen. Als Beispiele eignen sich das *Bigramm New York* beziehungsweise das *Trigramm New York City*, welche sich zu einem Token zusammenfassen lassen.

### 3.1.2 1-aus-n-Kodierung

Bei der 1-aus-n-Kodierung werden die im jeweiligen Textbeispiel vorkommenden Wörter in den Spalten der Matrix abgebildet. In die Zeilen dieser Matrix werden die eindeutigen Wörter geschrieben. Eine Zelle der Matrix repräsentiert das Vorhandensein eines einzigartigen Wortes an einer bestimmten Stelle im Text mit einer *1*. Die restlichen Zellen der Zeile markieren mit einer *0* die Abwesenheit des eindeutigen Wortes. Es ergibt sich also eine dünnbesetzte, hochdimensionierte Binärmatrix, wobei jede Spalte maximal eine festprogrammierte *1* enthält [Cho18, S. 237].

Das entnommene Beispiel in Tabelle 1 zeigt die 1-aus-n-Kodierung des Satzes „The bat sat on the cat.“ Der Satz besteht aus 6 Wörtern, wovon 5 eindeutig sind. Daraus resultiert die folgende  $5 \times 6$ -Matrix. Das Wort „the“ kommt in diesem Satz zweimal vor, was durch die entsprechenden Einträge an erster und fünfter Stelle der ersten Zeile zu sehen ist. Die übrigen eindeutigen Wörter werden mit einer *1* an ihrer Position im Satz markiert. Eine solche Darstellungsform ist sehr intuitiv und für kleine Textbeispiele sehr überschaubar. Bei großen Datensätzen wie dem *Yelp restaurant review dataset* führt die dünne Besetzung dazu, dass zwar die Wortstellung gespeichert wird, der Kontext jedoch

verloren geht. Darüber hinaus wird diese Art der Codierung bei entsprechend großen Datensätzen zunehmend ineffizient, aufgrund der schnell steigenden Dimensionen.

words	The	bat	sat	on	the	cat.
the	1	0	0	0	1	0
bat	0	1	0	0	0	0
on	0	0	0	1	0	0
⋮						
$n$						

Tabelle 1: 1-aus-n-Kodierung [KBB20, S. 32]

### 3.1.3 Wortvektoren

Das Einbetten von Wörtern in Wortvektoren bietet gegenüber dem One-Hot-Encoding den entscheidenden Vorteil, dass die Wortbedeutung über die Daten erlernt wird und abgelesen werden kann. Bei der Erstellung von Wortvektoren werden zunächst die eindeutigen Wörter an eine beliebige Stelle eines *Vektorraums* überführt. Dabei verschiebt sich die Position der Wörter so lange, bis der gesamte Datensatz eingelesen ist. Durch die Betrachtung zusätzlicher *Kontextwörter* kann die Bedeutung des eigentlichen *Zielwortes* im Wortvektor repräsentiert werden. Bei beispielsweise drei Kontextwörtern werden die drei Wörter vor und nach dem Zielwort betrachtet. Am Ende bekommt eine Position im Vektorraum eine bestimmte Bedeutung zugeordnet.

Im so entstehenden  $n$ -dimensionalen Vektorraum finden sich die einzelnen Wörter des Datensatzes wieder, wobei jedes durch einen Vektor (bspw.  $\vec{king}$ ) beschrieben wird. Der Spaltenvektor der Dimension  $n \times 1$  bestimmt den Ort des Wortes in allen verfügbaren Dimensionen. Aufgrund fehlendem Vorstellungsvermögen ist es für den Menschen schwierig sich mehr als drei Dimensionen vorzustellen. Ein zweidimensionales Beispiel für einen Vektorraum findet sich in Abbildung 1. Allgemein gilt, dass „je enger zwei Wörter im Vektorraum beieinander stehen, umso ähnlicher ist auch ihre Bedeutung, was durch die Ähnlichkeit der Kontextwörter bestimmt wird, die ihnen in der natürlichen Sprache nahe sind.“ [KBB20, S. 35] So liegen Synonyme und Falschschreibungen eng beieinander. Wortgruppen mit ähnlichem Kontext bilden Cluster und Subcluster innerhalb eines großen Vektorraumes.

Für Wortvektoren gilt: „Die geometrischen Beziehungen zwischen Wortvektoren sollten die semantische Beziehungen zwischen den Wörtern widerspiegeln.“ und „Wortvektoren haben die Aufgabe, die menschliche Sprache auf einen geometrischen Raum abzubilden.“ Ein Vektor von einem Wort zu einem anderen repräsentiert die „semantische Beziehung zwischen diesen Wörtern.“ [Cho18, S. 238f] Ein bekanntes Beispiel dafür ist die Hauptstadt-Land-Beziehung, bei der Abstand und Orientierung von einem Land

zu seiner Hauptstadt im Vektorraum auf andere Länder übertragen werden können, sodass der Vektor immer auf die zugehörige Hauptstadt zeigt. Eine konkrete Anwendung mit dem ebenso bekannten „*king-queen*“-Beispiel ist im nächsten Abschnitt unter Gleichung 1 zu sehen.

### 3.1.4 w2v from scratch

Da Word2Vec ein großer Bestandteil dieser Ausarbeitung ist, jedoch durch einen Import gelöst wurde, wird in diesem Kapitel anhand eines simplen Beispiels das Prinzip dahinter erklärt. Dabei werden hier keine hochdimensionalen Wort-Vektoren verwendet, welche letztlich durch ihre Cluster ausgeprägte Bedeutungsräume bilden. Sondern es werden die Beziehungen der Wörter zu ihren Nachbarn dargestellt, indem anhand eines Wortes das Nachbarwort bestimmt werden soll.

Listing 1: Datensatz Word2Vec from scratch

```
0 corpus = ['king is a strong man',
1         'queen is a wise woman',
2         'boy is a young man',
3         'girl is a young woman',
4         'prince is a young king',
5         'princess is a young queen',
6         'man is strong',
7         'woman is pretty',
8         'prince is a boy will be king',
9         'princess is a girl will be queen']
```

Der hier genutzte Datensatz ist dabei recht überschaubar und auch in einer einfachen Struktur gehalten. Auf diese Weise lassen sich die Interaktionen und Einflüsse der Wort-Vektoren übersichtlicher darstellen.

Listing 2: Entfernen der Stoppwörter

```
10 def remove_stop_words(corpus):
11     stop_words = ['is', 'a', 'will', 'be']
12     results = []
13     for text in corpus:
14         tmp = text.split(' ')
15         for stop_word in stop_words:
16             if stop_word in tmp:
17                 tmp.remove(stop_word)
18         results.append(" ".join(tmp))
19
```



```
20     return results
21
22 corpus = remove_stop_words(corpus)
```

Zunächst werden die, in Kapitel 3.1.1 bereits erklärten stop words, entfernt. Dadurch, dass hier ein kurzes, akademisches Beispiel vorliegt, ist auch die Liste der stop words eher kurz. In der Regel sind Wörter, wie die auch hier genutzten, ebenfalls in diesen Listen zu finden, aber je nach Anwendungsfall kann es auch Sinn ergeben diese mit ungewöhnlicheren Wörtern zu füllen.

Listing 3: Nachbarwörter speichern

```
23 from ordered_set import OrderedSet
24
25 words = OrderedSet()
26 for text in corpus:
27     for word in text.split(' '):
28         words.add(word)
29
30 word2int = {}
31
32 for i, word in enumerate(words):
33     word2int[word] = i
34
35 sentences = []
36 for sentence in corpus:
37     sentences.append(sentence.split())
38
39 WINDOW_SIZE = 2
40
41 data = []
42 for sentence in sentences:
43     for idx, word in enumerate(sentence):
44         for neighbor in sentence[max(idx - WINDOW_SIZE, 0)
45                                 : min(idx + WINDOW_SIZE, len(sentence)) + 1] :
46             if neighbor != word:
47                 data.append([word, neighbor])
48
49 import pandas as pd
50 df = pd.DataFrame(data, columns = ['input', 'neighbour'])
```

Als nächstes wird ein `OrderedSet` erstellt, welches alle Wörter nur ein einziges Mal enthält und entsprechend durchnummeriert. Dies wird im späteren Verlauf noch benötigt, wobei die Reihenfolge wichtig für die korrekte Visualisierung ist. Anschließend wird ein Datenset erstellt, welches alle Wörter mit ihren jeweiligen Nachbarn enthält. Dabei entspricht die `WINDOW_SIZE` in Zeile 17 wie viele Nachbarwörter betrachtet werden sollen. Schließlich haben in langen Sätzen Wörter, welche zum Ende eines Nebensatzes stehen, häufig eine eher schwache oder gar keine Bindung zu den ersten Wörtern des Hauptsatzes. Die ideale Größe der `WINDOW_SIZE` hängt dabei individuell vom Text ab. Jedoch ist in der Regel eine eher kleine `WINDOW_SIZE` zu empfehlen,  $\pm 2$  bis 3 Wörter sind in vielen Fällen genug. Abschließend wird das Datenset in ein pandas Dataframe überführt, wodurch das Arbeiten mit diesem erleichtert wird. Weiterhin sind noch 2 Aspekte des Datensets auffällig. Zum einen werden die Wörter mit ihren Nachbarn zwangsweise bidirektional abgespeichert. Die Erklärung dafür ist simpel: Die Beziehung zwischen den Wörtern ist auch bidirektional und so wird sichergestellt, dass im späteren Verlauf des Codes diese Bidirektionalität auch eingehalten wird, ohne dass darauf explizit geachtet werden muss. Zum anderen werden mehrfach auftretende Kombinationen auch mehrfach abgespeichert und nicht im Nachhinein wieder auf ein unique-dataset bereinigt. Das kommt daher, dass diese Häufigkeit essenzielle Daten enthält. Man geht letztlich davon aus, dass Wörter, die häufig zusammen vorkommen, eine stärkere Bindung haben und auch außerhalb des benutzten Datensets verstärkt gemeinsam auftreten.

Listing 4: Beispielmodell

```
51 import tensorflow as tf
52 import numpy as np
53 from tensorflow.keras.models import Sequential
54 from tensorflow.keras.layers import Dense
55 from tensorflow import keras
56
57 ONE_HOT_DIM = len(words)
58 EMBEDDING_DIM = 2
59
60 # function to convert numbers to one hot vectors
61 def to_one_hot_encoding(data_point_index):
62     one_hot_encoding = np.zeros(ONE_HOT_DIM)
63     one_hot_encoding[data_point_index] = 1
64     return one_hot_encoding
65
66 X = [] # input word
67 Y = [] # target word
68
```

```

69 for x, y in zip(df['input'], df['neighbour']):
70     X.append(to_one_hot_encoding(word2int[ x ]))
71     Y.append(to_one_hot_encoding(word2int[ y ]))
72
73 # convert them to numpy arrays
74 X_train = np.asarray(X)
75 Y_train = np.asarray(Y)
76
77 model = Sequential()
78 model.add(Dense(EMBEDDING_DIM, activation='relu', input_dim=
    X_train.shape[1]))
79 model.add(Dense(X_train.shape[1], activation = "softmax"))
80
81 model.compile(optimizer='adam', loss='
    categorical_crossentropy', metrics=["accuracy"])
82
83 model.fit(X_train, Y_train, epochs=1000, verbose=True)

```

Bevor mit den Daten trainiert werden kann müssen die Wörter in eine dafür nutzbare Form übertragen werden. Dies geschieht in diesem Beispiel durch ein One-Hot-Encoding. Dabei wird das zuvor erstellte Dataset der durchnummerierten, einzigartigen Wörter zu Hilfe genommen. Gemäß der ursprünglichen Zielsetzung entsprechen die Ausgangswörter den Eingangsdaten und die Nachbarwörter den Zieldaten. Zum Schluss werden diese Daten für das Training in numpy arrays übertragen und einem einfachen Modell fürs Training übergeben.

Listing 5: Plotten

```

84 vectors = np.array(model.weights[0])
85 w2v_df = pd.DataFrame(vectors, columns = ['x1', 'x2'])
86 w2v_df.insert(0, "word", words)
87
88 import matplotlib.pyplot as plt
89
90 fig, ax = plt.subplots()
91
92 for word, x1, x2 in zip(w2v_df['word'], w2v_df['x1'],
    w2v_df['x2']):
93     ax.annotate(word, (x1, x2))
94
95 PADDING = 1.0

```

```

96 x_axis_min = np.amin(vectors, axis=0)[0] - PADDING
97 y_axis_min = np.amin(vectors, axis=0)[1] - PADDING
98 x_axis_max = np.amax(vectors, axis=0)[0] + PADDING
99 y_axis_max = np.amax(vectors, axis=0)[1] + PADDING
100
101 plt.xlim(x_axis_min, x_axis_max)
102 plt.ylim(y_axis_min, y_axis_max)
103 plt.rcParams["figure.figsize"] = (10,10)
104
105 plt.show()

```

Die eigentliche Arbeit ist damit bereits getan. Da dieses Beispiel jedoch zur Veranschaulichung gedacht ist fehlt noch die Visualisierung. Dafür werden die Gewichte der jeweiligen Vektoren zunächst zusammen mit den entsprechenden Wörtern in ein Dataframe gespeichert. Anschließend wird der Plot vorbereitet und die Wörter werden anhand ihrer zweidimensionalen Vektoren auf einem Koordinatensystem verteilt. Das hier verwendete Beispiel ist insgesamt sehr akademisch gestaltet. Dadurch, dass zwei Dimensionen für Wortvektoren ziemlich gering sind, das Modell nicht ausführlich optimiert und auch die Ausgangsdaten relativ spärlich sind, können die Ergebnisse stark voneinander variieren. Eine sinnvolle Verteilung ist jedoch in Abbildung 1 zu sehen. Hier ist zu erkennen, dass die Begriffe, welche eher weiblich sind, im oberen Bereich und dementsprechend die männlichen im unteren Bereich sind. Weiterhin sind die allgemeinen Begriffe **man** und **woman** eher rechts zu finden, während spezifische Wörter wie **king** oder **queen** sich eher links befinden. In dieser Darstellung funktioniert auch das berühmte Beispiel von:

$$\vec{king} - \vec{man} + \vec{woman} \approx \vec{queen} \quad (1)$$

mit den grob abgelesenen Werten:

$$\begin{pmatrix} -0,5 \\ -0,5 \end{pmatrix} - \begin{pmatrix} 1,5 \\ -0,5 \end{pmatrix} + \begin{pmatrix} 1,75 \\ 1 \end{pmatrix} = \begin{pmatrix} -0.25 \\ 1 \end{pmatrix} \quad (2)$$

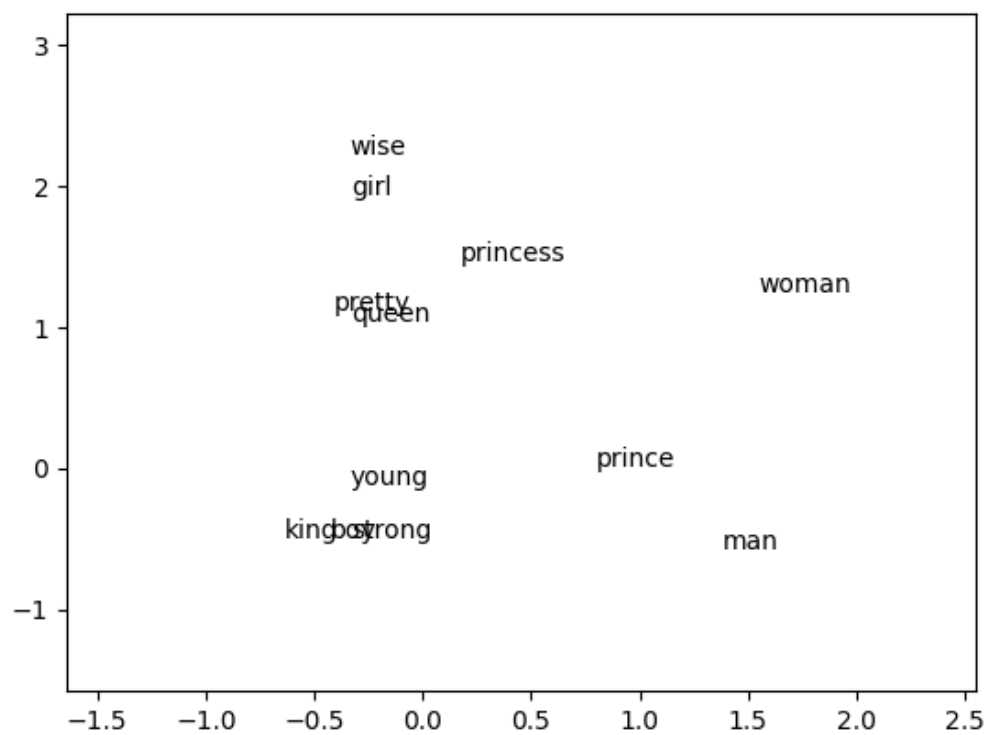


Abbildung 1: Wort-Vektoren Verteilung

## 4 Umsetzung

### 4.1 Analyse des Datensatzes

Ein wichtiger Schritt in jeder Aufgabe im Bereich des maschinellen Lernens ist es, die Datenbasis zu analysieren und zu verstehen.

Der Yelp-Datensatz besteht aus einer Ansammlung von json Dateien. Darunter befindet sich jeweils eine Datei für businesses, reviews, user, checkins, sowie tips - quasi short-reviews - und photos, welche als Cross-Referenz für andere Datensätze fungiert. für dieses Projekt ist jedoch nur die review Datei interessant.

Die Yelp-Reviews befinden sich in einer 6 GB großen json Datei. Diese Datei könnte mit einem leistungsstarken Computer vollständig in den Arbeitsspeicher geladen werden. Jedoch wird im Rahmen dieser Arbeit die Datei zeilenweise verarbeitet.

Das zeilenweise Verarbeiten der Datei ermöglicht es, die Systemanforderungen gering zu halten und bringt den zusätzlichen Vorteil, dass der in dieser Arbeit erarbeitete Ansatz problemlos auf größere Datensätze angewendet werden kann.

#### 4.1.1 Datenstruktur

Das Programm in Listing 6 gibt die zwei ersten Zeilen der json Datei aus.

Listing 6: Programm zur ermittlung der Datenstruktur

```
0 data_path = "yelp_academic_dataset_review.json"
1
2 for index, line in enumerate(open(data_path, encoding="utf8")
   ):
3     if(index>1):
4         break
5     print(line)
```

In Zeile 0 wird der Pfad des Datensatzes definiert.

Danach wird in Zeile 2 mit einer for-Schleife über jede Zeile des Datensatzes iteriert, wobei die Funktion **open** die Datei öffnet und ein iterierbares Objekt zurück liefert.

Durch das Benutzen der Funktion **enumerate** erhält man den aktuellen Index, sodass in der Variable **index** die aktuelle Zeilennummer gespeichert ist und in der Variable **line** die aktuelle Zeile selbst.

Die if-Abfrage in Zeile 3 sorgt dafür, dass die for-Schleife vorzeitig abgebrochen wird. In Zeile 5 wird der Text der aktuellen Zeile ausgegeben.

Das Ausführen des Quellcodes in Listing 6 führt zu folgender Ausgabe:

```
{"review_id":"xQY8N_XvtGbearJ5X4QryQ",
"user_id":"0wjRMXRC0KyPrIlcjaXeFQ",
"business_id":"-MhfebMOQIsKt87iDN-FNw",
"stars":2.0,
"useful":5,
"funny":0,
"cool":0,
"text":"As someone who has worked with many museums, I was
eager to visit this gallery on my most recent trip to
Las Vegas. When I saw they would be showing infamous
eggs of the House of Faberge from the Virginia Museum of
Fine Arts (VMFA), I knew I had to go!\n\nTucked away
near the gelateria and the garden, the Gallery is pretty
much hidden from view. It's what real estate agents
would call \"cozy\" or \"charming\" - basically any
euphemism for small.\n\nThat being said, you can still
see wonderful art at a gallery of any size, so why the
two *s you ask? Let me tell you:\n\n* pricing for this,
while relatively inexpensive for a Las Vegas attraction,
is completely over the top. For the space and the
amount of art you can fit in there, it is a bit much.\n*
it's not kid friendly at all. Seriously, don't bring
them.\n* the security is not trained properly for the
show. When the curating and design teams collaborate for
exhibitions, there is a definite flow. That means
visitors should view the art in a certain sequence,
whether it be by historical period or cultural
significance (this is how audio guides are usually
developed). When I arrived in the gallery I could not
tell where to start, and security was certainly not
helpful. I was told to \"just look around\" and \"do
whatever.\" \n\nAt such a *fine* institution, I find the
lack of knowledge and respect for the art appalling.",
"date":"2015-04-15 05:21:16"}

{"review_id":"UmFMZ8PyXZTY2QcwzsfQYA",
"user_id":"nIJD_7ZXHq-FX8byPM0kMQ",
"business_id":"lbrU8StCq3yDfr-QMnGrmQ",
```

```
"stars":1.0,  
"useful":1,  
"funny":1,"cool":0,  
"text":"I am actually horrified this place is still in  
    business. My 3 year old son needed a haircut this past  
    summer and the lure of the $7 kids cut signs got me in  
    the door. We had to wait a few minutes as both stylists  
    were working on people. The decor in this place is total  
    garbage. It is so tacky. The sofa they had at the time  
    was a pleather sofa with giant holes in it. And my son  
    noticed ants crawling all over the floor and the  
    furniture. It was disgusting and I should have walked  
    out then. Actually, I should have turned around and  
    walked out upon entering but I didn't. So the older  
    black male stylist finishes the haircut he was doing and  
    it's our turn. I tell him I want a #2 clipper around  
    the back and sides and then hand cut the top into a  
    standard boys cut. Really freaking simple, right? WRONG!  
    Rather than use the clippers and go up to actually cut  
    the hair, he went down. Using it moving downward doesn't  
    cut hair, it just rubs against it. How does this man  
    who has an alleged cosmetology license not know how to  
    use a set of freaking clippers??? I realized almost  
    immediately that he had no idea what he was doing. No  
    idea at all. After about 10 minutes of watching this guy  
    stumble through it, I said \"you know what? That's fine  
    .\", paid and left. All I wanted to do was get out of  
    that scummy joint and take my son to a real haircut  
    place.\n\nBottom line: DO NOT GO HERE. RUN THE OTHER WAY  
    !!!!!",  
"date":"2013-12-07 03:16:52"}
```

Hier ist zu erkennen, dass jede Zeile der review.json eine Rezension mit den Merkmalen `review_id`, `user_id`, `business_id`, `stars`, `useful`, `funny`, `cool`, `text` und `date` enthält. Die meisten dieser Merkmale sind selbsterklärend. Erwähnenswert sind dabei jedoch die Merkmale `useful`, `funny` und `cool`, welche Reaktionen anderer Nutzer auf diese Review darstellen. Für dieses Projekt sind hingegen nur die beiden Punkte `stars` und `text` von Bedeutung. In diesem Datensatz sind Sonderzeichen mit einem Backslash escaped.



### 4.1.2 Anzahl der Reviews

Um die Anzahl der Reviews zu ermitteln, wird die Datei einmal in Listing 7 komplett durchlaufen und die Anzahl der Zeilen aufaddiert.

Aufgrund des Aufbaus der Datei entspricht die Summe aller Zeilen der Anzahl der Reviews im Datensatz.

Listing 7: Programm zur Ermittlung der Anzahl der Reviews

```
0 nr_of_reviews= 0
1 for line in open(data_path,encoding="utf8"):
2     nr_of_reviews +=1
3 print (nr_of_reviews))
```

Das Programm liefert die Anzahl 8021122 zurück.

### 4.1.3 Reviewlänge

Listing 8: Programm zur Ermittlung der Reviewlänge

```
0 import json
1 import numpy as np
2 X = []
3 for line in open(data_path,encoding="utf8"):
4     l = json.loads(line)
5     X.append(len(l["text"].strip().split(" ")))
6 print("Der Median der Textlaenge ist: %d" %(np.median(X)))
7 print("Der Durschnitt der Textlaenge ist: %d"%(np.mean(X)))
```

Das Importieren der benötigten Libraries erfolgt in Zeile 0 und 1.

In Zeile 2 wird X als leere Liste definiert.

Anschließend wird in der 3. Zeile wie gehabt über jede Zeile des Datensatzes iteriert.

In der Schleife in Zeile 4 parsen wir die aktuelle Zeile des Datensatzes und konvertieren diese in ein Python Dictionary. Danach wird in Zeile 5 der Reviewtext der aktuellen Zeile in Wörter zerteilt und anschließend wird die Anzahl der Wörter der Liste X hinzugefügt.

Nach dem Durchlauf der Schleife erhalten wir eine Liste, die alle Reviewlängen beinhaltet.

In der 6. Zeile wird die Numpy-Funktion `median` genutzt, um den Median der Reviewlänge auszugeben. Anschließend wird in Zeile 7 die Durchschnittslänge mithilfe der Numpy-Funktion `mean` ausgegeben.

Die Medianlänge beträgt für diesen Datensatz 78 Wörter, die Durchschnittslänge beträgt 110 Wörter.

#### 4.1.4 Distribution der Bewertungen

Ein interessantes Merkmal eines Datensatzes ist es, wie ausgeglichen die Zieldaten sind.

Für diese Arbeit sind die Zieldaten die Bewertungen.

Listing 9: Programm zur Ermittlung der Distribution

```
0 import json
1 import numpy as np
2
3 ratings = []
4 for line in open(data_path, encoding="utf-8"):
5     json_line = json.loads(line)
6     ratings.append(float(json_line["stars"]))
7 ratings = np.array(ratings)
```

Im Grunde wird hier die gleiche Schleife durchlaufen wie bei der Textlänge.

Anstelle der Textlänge parsen wir diesmal die Bewertung und speichern diese in der Liste `ratings`.

In Zeile 7 wird die Liste zu einem Numpy-Array konvertiert.

```
8 from matplotlib import pyplot as plt
9 value , count = np.unique(ratings, return_counts=True)
10
11 fig = plt.figure()
12 ax = fig.add_axes([0,0,1,1])
13 ax.set_title('Rating distribution')
14 ax.set_xlabel('Rating')
15 ax.set_ylabel('Occurrences (%)')
16 ax.bar(value, count/ratings.size *100)
```

In Zeile 8 wird `pyplot` von der Library `matplotlib` unter dem Alias `plt` importiert.

Danach werden in Zeile 9 mit der Funktion `unique` von Numpy die verschiedenen Bewertungen in der Variable `value` gespeichert. Der Parameter `return_counts=True` sorgt dafür, dass zusätzlich die Häufigkeit der Bewertung zurückgegeben wird. Die Häufigkeit wird in der Variable `count` gespeichert.

In Zeile 11 bis 16 wird der Plot erstellt.

Das Ausführen des Quelltextes führt zu folgendem Plot, welcher in Bild 2 zu sehen ist. Hier ist zu erkennen, dass die Verteilung nicht gleichmäßig ist. Dies führt zu verschiedenen Ansätzen, welche im folgenden Kapitel 4.2.1 erläutert werden.

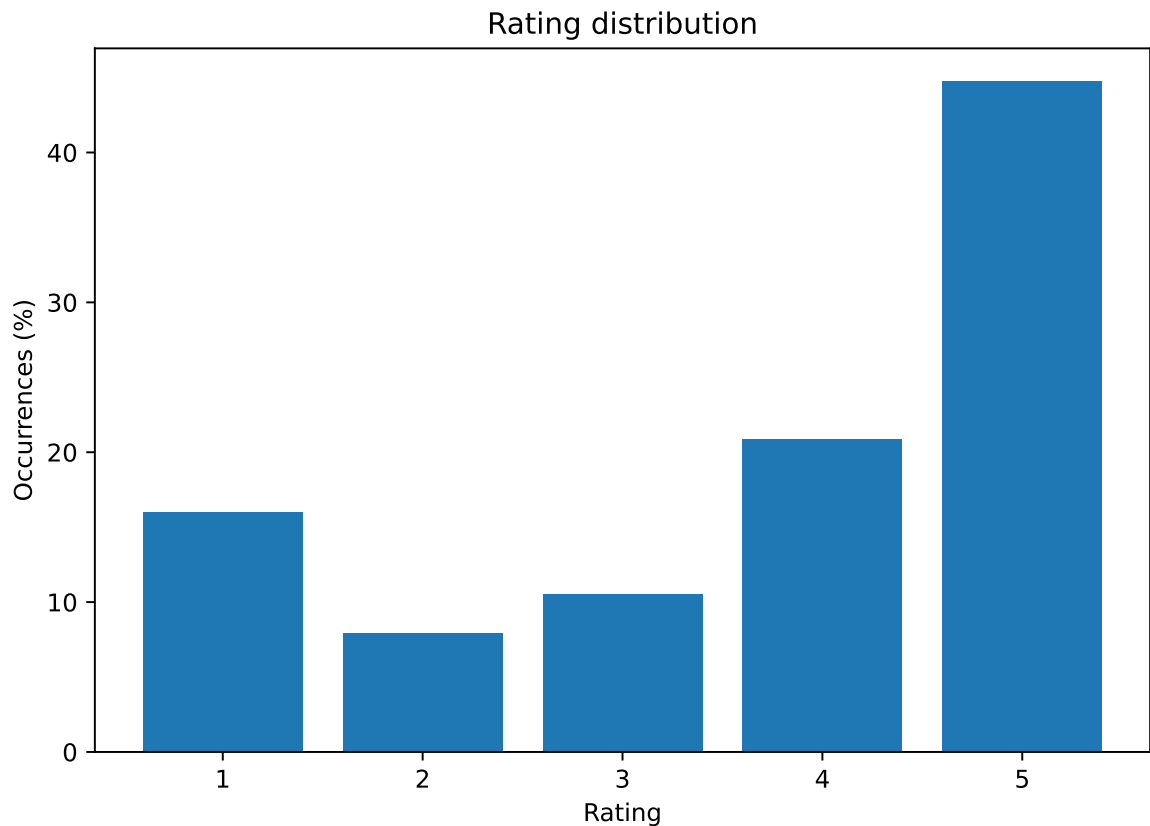


Abbildung 2: Distribution der Bewertungen

## 4.2 Datenaufbereitung

Durch das Wissen, wie Word2Vec funktioniert, lassen sich verschiedene Probleme bei diesem Datensatz feststellen. Dabei sind einige davon generisch und annähernd in jedem Word2Vec- oder machine-learning-Projekt vertreten. Andere jedoch sind verhältnismäßig speziell für einen Datensatz wie diesen.

### 4.2.1 Distribution

Eine generelle Voraussetzung für Algorithmen des maschinellen Lernens ist, dass mit einem ausgeglichenen Datensatz gearbeitet wird. Unausgeglichene Datensätze wie der *Yelp* Datensatz verleiten dazu, dass neuronale Netze eben genau diesen Bias lernen. Wenn der Datensatz jedoch die Realität relativ gut darstellt, ist es gerade für Anfänger schwierig zu entscheiden wie damit umgegangen werden soll. Schließlich ist es prinzipiell eine hilfreiche Information wenn es bekannt ist, dass z. B. Bewertungen eine Verteilung wie in Abbildung 2 besitzen. Der entwickelte Algorithmus soll die Kategorisierung

aber nicht an fundiertem Raten ausmachen, sondern anhand von selbst erarbeiteten Kriterien und damit unabhängig von der ursprünglichen Distribution.

Generell gibt es zwei Ansätze, um diesen Effekt zu bekämpfen. Zum Einem gibt es die Möglichkeit, den *Near-Miss-Algorithm* zu benutzen. Mit Hilfe von diesem werden im Grunde Einträge von zu oft vertretenen Klassen aus dem Datensatz entfernt, bis dieser ausgeglichen ist. Dadurch wird zwar verhindert, dass eine Klasse bei der Kategorisierung bevorzugt wird, aber man verringert auch den zu Grunde liegenden Datensatz. Je nach ursprünglicher Verteilung und gesamter Menge kann dies zu einem zu kleinen Datensatz und somit zu weiteren Problemen führen. Zum Anderen gibt es die Möglichkeit, die Klassen beim Trainieren zu gewichten. Hier werden selten auftretende Klassen stärker gewertet, sodass das Programm diesen eine höhere Aufmerksamkeit schenkt. Im Rahmen dieser Arbeit wurde der zweite Ansatz gewählt.

#### 4.2.2 Zeichensatz

Zuerst sollte der Datensatz in lowercase konvertiert werden, da bei den Wortvektoren sonst ein Unterschied zwischen z.B. *good* und *Good* bestünde. Dabei könnte zufallsbedingt, durch die Anwendung in leicht verschiedenen Kontexten und die generelle unterschiedliche Häufigkeit, die Interpretation dieser Wörter auseinandergehen.

In eine ähnliche Richtung gehen auch die Satzzeichen. Hier werden zwar Informationen gespeichert, welche durchaus die Bedeutung eines Satzes verändern können, aber dieser Unterschied ist in der Regel zu komplex, um ihn in den meisten Verfahren sinnvoll zu berücksichtigen. Eine Sonderbehandlung wäre dabei ohnehin von Nöten. Weiterhin kann man davon ausgehen, dass sich in der Regel die Bedeutung nicht so gravierend ändert, als dass man diese Review in eine andere Kategorie einordnen würde. Dadurch ist die effizienteste Methode, die Satzzeichen einfach zu entfernen.

Es gibt aber noch weitere Sonderzeichen, die Probleme bereiten. Darunter z.B. das Apostroph. Da die Reviews in der Regel englisch sind finden sich häufiger Wörter wie *didn't* oder *we've*. Der Datensatz besteht jedoch aus Reviews aus dem Internet. Hier werden neben Satzzeichen auch Apostrophe oft weggelassen, da Sätze und Wörter wie *didnt* auch ohne diese Zusätze verständlich sind. Aber auch hier werden diese Versionen als unterschiedliche Wörter interpretiert. Die simpelste Methode ist auch hier einfach das Apostroph zu entfernen. In diesem Fall werden aber zum Teil Wörter zusammengefasst, welche durchaus verschieden sind, wie z.B. *its* und *it's*. Dasselbe gilt dann auch für andere Sprachen bei denen solche Vorkommnisse möglicherweise üblicher sind. Jedoch wird auch bei Wörtern wie *it's* oft das Apostroph weggelassen, da häufig aus dem Kontext ersichtlich ist, welche Variante gemeint ist. Dadurch wird für den Algorithmus der Unterschied zwischen diesen Wörtern nicht klar und sie werden letztlich im Vek-

torraum recht nah aneinander liegen. Dies lässt sich auch aus der folgenden Ausgabe anhand des genutzten Modells ablesen.

```
In: model0.similarity("it's", "its")
```

```
Out: 0.8579778
```

### 4.2.3 Sprachen

Wie zuvor erwähnt besteht der Datensatz nicht nur aus englischen Reviews, sondern aus verschiedenen Sprachen. Oberflächlich gesehen stellt dies in der Theorie erstmal kein Problem dar, abgesehen von den unterschiedlichen Mengen an data-samples in den jeweiligen Sprachen. Wie durch den vorherigen Abschnitt jedoch zu erahnen ist, benötigen verschiedene Sprachen verschiedene Nuancen an Vorverarbeitung. Von Sonderzeichen, die je nach Sprache ggf. anders behandelt werden müssen, bis hin zum Fehlen ebendieser. In dem Datensatz sind z. B. auch japanische Reviews enthalten, eine Sprache ohne Leerzeichen, was das Isolieren einzelner Wörter erheblich erschwert. Um bei dem Beispiel Japanisch zu bleiben: Hier kommt noch hinzu, dass dies eine sehr kontextuelle Sprache ist. Das bedeutet, dass Wörter, die für das Verständnis des Satzes wichtig wären, aus dem Kontext als gegeben und gewusst angesehen werden. Dies ist etwas, was ein Code in der Regel nicht leisten kann.

Es stellt sich also die Frage, ob der Datensatz von anderen Sprachen bereinigt werden sollte. Ein möglicher Ansatz dafür wäre z. B. nur Reviews zu behalten, welche ausschließlich Zeichen in einer bestimmten (kleinen) utf-8 Reichweite enthalten. Während das Entfernen bei einigen asiatischen Sprachen noch verhältnismäßig einfach erscheint, wird dies bei z. B. europäischen Sprachen wie Französisch oder Spanisch ohne großen Mehraufwand nahezu unmöglich.

Aufgrund dessen wird darauf verzichtet solche Sprachen zu entfernen. Aber auch asiatische Sprachen werden wir nicht extra entfernen. Es können zwar überwiegend keine nützlichen Daten aus diesen Reviews gewonnen werden, jedoch wird der nützliche Part des Datensatzes auch nicht dadurch beeinflusst, da die asiatischen Sprachen mit ziemlicher Sicherheit einen eigenen, eher abgeschiedenen Wortcluster bilden werden. Weiterhin treten diese Reviews selten genug auf, dass man davon ausgehen kann, dass das Entfernen dieser Datenpunkte keinen merklichen Einfluss auf die Trainingszeiten haben wird. Daher ist es sinnvoller die Zeit, welche in die saubere Entfernung dieser Reviews investiert werden würde, hier einzusparen.

## 4.3 Generatoren

Spätestens, wenn mit etwas größeren Datenmengen trainiert wird, stellt sich bei einer unserer Methoden ein neues Problem dar. Während die mean-Methode ohne Probleme

durchläuft, bricht die CNN-Methode aufgrund mangelnden Arbeitsspeichers ab. Dies lässt sich auch anhand folgender Formel nachrechnen:

$$\text{Anzahl Einträge} \times \text{Vektorenanzahl} \times \text{Vektorendimension} \times \text{Datentyp-Größe} \quad (3)$$

Der Datentyp entspricht in unserem Fall float. Ein float wird in Python durch 64 Bit, also 8 Byte dargestellt. Geht man davon aus, dass alle 8 Millionen Einträge fürs fitten und somit gleichzeitig im Arbeitsspeicher benötigt werden, so ergibt sich für die mean-Vektor Methode folgende Rechnung:

$$8,000,000 \times 1 \times 100 \times 8 \approx 6 \text{ GB} \quad (4)$$

Dies könnte mit 8GB RAM etwas knapp werden, jedoch stellt es bei 16GB kein Problem dar. Bei der CNN-Methode sieht das jedoch wie folgt aus:

$$8,000,000 \times 72 \times 100 \times 8 \approx 430 \text{ GB} \quad (5)$$

Das Problem ist recht deutlich. Viele Personen hätten auf ihrem privaten PC vermutlich nicht einmal genug Festplattenspeicher für diese Datenmenge, geschweige denn Arbeitsspeicher.

Die Daten müssen bei der CNN-Methode also nur stückweise in den Arbeitsspeicher geladen werden, dann wird mit diesen trainiert und anschließend werden sie wieder aus dem Arbeitsspeicher gelöscht, um Platz für die nächsten Daten zu machen. Dabei gibt es wieder verschiedene Ansätze, welche im Folgenden beschrieben werden.

### 4.3.1 Erster Generator

Ein möglicher Ansatz ist, die rohen Daten nur schrittweise einzulesen und diese dann erst dem word2vec model zu übergeben, um daraus einen Vektor zu machen. Dafür wird eine Funktion geschrieben, welche dann der fit Methode des eigentlichen machine learning models als Datensatz-parameter übergeben wird. Dabei wird in der Funktion die batchsize berücksichtigt, sodass stets die gewünschte Menge an Daten gleichzeitig in das Modell geladen werden.

Um dies jedoch zu erfüllen, müssen die Daten nach ihrem Nutzen in verschiedene Dateien aufgeteilt werden. Dementsprechend in eine Trainings-, Validierungs- und Test-Datei.

Listing 10: Datenaufteilung

```
0 val = open('val.json', 'w', encoding='utf8')
1 test = open('test.json', 'w', encoding='utf8')
2 train = open('train.json', 'w', encoding='utf8')
```

```
3
4 i = 0
5 for line in open(corpus_path, encoding="utf8"):
6     if(i<3): train.write(line)
7     elif(i==3): val.write(line)
8     else:
9         i = -1
10        test.write(line)
11    i += 1
12
13 val.close()
14 test.close()
15 train.close()
```

Das Aufteilen wird durch ein einfaches Zählen und Sortieren gelöst. Dabei zählt eine Variable, hier `i`, hoch und je nachdem welchen Wert diese Variable hat werden die Daten in die entsprechende Datei sortiert. Das Ziel ist eine 60-20-20 Aufteilung der Daten. Dafür werden bei den Werten 0, 1 und 2 die Daten in die Trainingsdatei geschrieben und dementsprechend bei den Werten 3 und 4 in die Validierungs- bzw. Testdatei. Wenn in die Testdatei geschrieben wird, wird auch gleichzeitig die Zählervariable zurückgesetzt, sodass mit den gleichen Werten von vorne begonnen werden kann. Hier wurde sich bewusst für ein Zurücksetzen der Variable entschieden und gegen ein kontinuierliches Hochzählen und der Sortierung durch Modulo. Auf diese Weise werden bedenkenlos Overflow- und Signifikanzprobleme verhindert, ohne einen deutlichen Verlust der Lesbarkeit des Codes hinnehmen zu müssen. Da der genutzte Datensatz nur etwas über 8 Millionen Einträge besitzt und mit Ganzzahlen gearbeitet wird, wären beide Probleme zu vernachlässigen gewesen. Um einen generell robusteren Code zu nutzen, wurde dieses Vorgehen trotzdem gewählt. Diese Art der Aufteilung ist jedoch nur bedenkenlos einzusetzen in diesem Projekt, da die Daten ohnehin unabhängig sind und ebenfalls unsortiert vorliegen. Aus diesen Gründen sind die drei Datensätze auch ohne extra shuffle zufällig genug.

Um diese Dateien nun während des Trainings schrittweise auszulesen, muss ein sogenannter Generator erstellt werden. Dieser liest die Daten zum Teil aus, überträgt sie in das Word2Vec Modell und stellt sie anschließend dem Modell zur Verfügung.

Listing 11: Overhead einfacher Generator

```
0 import numpy as np
1 import json
2 def generate_arrays_from_file(path, batchsize):
3     inputs = []
```

```
4     targets = []
5     batchcount = 0
6     while True:
7         with open(path, encoding="utf8") as f:
8             for line in f:
9                 json_line = json.loads(line)
```

Die Zeile 7 mit **while True:** wird dabei vom Keras Modell erwartet, denn das Trainieren soll nicht mitten in einer Epoche enden, weil das Ende der Trainingsdatendatei erreicht wurde.

Listing 12: Sortierung einfacher Generator

```
10         try:
11             inputs.append(getSentenceVectorCNN(
12                 json_line["text"]))
13             y = float(json_line["stars"])
14             if(y <3):
15                 targets.append(0)
16             elif(y==3):
17                 targets.append(1)
18             else:
19                 targets.append(2)
20             batchcount += 1
21         except:
22             continue
```

Anschließend wird aus dem eigentlichen Text der Review ein Wortvektor für das CNN gemacht. Das Vorgehen dafür wird in Kapitel 4.6.1 im Listing 29 beschrieben. Das Ergebnis, ob diese Review als positiv, negativ oder neutral gelten soll, wird in diesem Beispiel als Index Kategorie gespeichert. Sollte eine Exception geschmissen werden, weil z. B. kein einziges Wort, welches in der Review vorkommt, im Word2Vec Modell vertreten ist, wird diese Review einfach übersprungen.

Listing 13: Yield der Daten

```
22         if batchcount > batchsize:
23             X = np.array(inputs)
24             y = np.array(targets)
25             yield (X, y)
26             inputs = []
27             targets = []
28             batchcount = 0
```



Abschließend werden die beiden Listen returned und die entsprechenden Variablen zurückgesetzt. Yield entspricht dabei in Python einem return, welches den dazugehörigen Code erst ausführt, wenn die Daten gebraucht werden. Die Ansprüche an den Arbeitsspeicher werden auf diese Weise deutlich gesenkt. Es müssen für das Training also nur so viele Daten gleichzeitig im Arbeitsspeicher gehalten werden wie es die batchsize vorgibt. Wenn diese z. B. 512 entspricht fallen dabei, an folgender Formel zu erkennen, lediglich 28 MB an Daten an.

$$512 \times 72 \times 100 \times 8 \approx 28 \text{ MB} \quad (6)$$

Ein deutlicher Nachteil bei diesem Vorgehen ist jedoch, dass dies sehr zeitaufwändig ist. Nicht nur müssen die Daten während des Trainings von der Festplatte anstelle des Arbeitsspeichers gelesen werden. Sondern diese Daten müssen zunächst auch noch in das Word2Vec Modell überführt werden. Beides sind verhältnismäßig langsame Prozesse, wodurch die Trainingszeit enorm steigt. Aber auch die Validierung benötigt nun in etwa genauso viel Zeit wie das Training an sich. Dafür ist diese Methode recht simpel und benötigt keinen zusätzlichen Festplattenspeicher. Jedoch muss hierfür der Datensatz für Trainings-, Validierungs- und Testmenge in verschiedene Dateien aufgeteilt werden. Sollte also der Festplattenspeicher und nicht so sehr die Zeit ein begrenzender Faktor sein, ist dies ein durchaus valider Ansatz.

#### 4.3.2 hdf5

Ein weiterer Ansatz ist die bereits in das Word2Vec Modell überführten Wörter, also die Vektoren an sich, abzuspeichern. Dafür eignen sich z. B. hdf5 Dateien, welche einen weiteren Vorteil besitzen, auf den später eingegangen wird. Das Prinzip ist bei dieser Methode nahezu identisch zu der zuvor besprochenen, auch hier wird der Fit-Methode ein Generator gegeben. Dieser liest jedoch nur die entsprechende Datei und muss die Daten nicht noch vorher umwandeln. Zunächst muss aber der Datensatz in das Word2Vec Modell überführt und entsprechend abgespeichert werden. Dies beginnt auch hier wieder mit overhead:

Listing 14: Overhead hdf5 Generator

```
0 import h5py
1
2 path = "path/to/my/workspace/"
3 sample_path = path + "sample.json"
4
5 with h5py.File(path + "w2vCNN.hdf5", "w") as hf:
6     i = 0
7     chunkSize = 10**4
```

```

8     trainChunk = int(chunkSize * 0.6)
9     valTestChunk = int(chunkSize * 0.2)
10    xTrain = []
11    yTrain = []
12    xVal = []
13    yVal = []
14    xTest = []
15    yTest = []

```

Zu Beginn wird eine Zähler-Variable auf 0 initialisiert, welche später für die Aufteilung in Trainings-, Validierungs- und Testmenge genutzt wird. Für diese Mengen und ihre jeweiligen Input- und Outputdaten werden auch ein paar Listen vorbereitet. In hdf5 Dateien kann man die Daten in sogenannte Chunks aufteilen, welche das schrittweise speichern möglich macht, aber auch das Laden verbessern soll. Die Größe dieser Chunks wird zunächst auf 10.000 festgelegt. Da ein 60-20-20 split des Datensatzes gewünscht ist werden ebenfalls die Größe dieser einzelnen Chunks gespeichert.

Listing 15: Sortierung hdf5 Generator

```

16    for index, line in enumerate(open(corpus_path, encoding=
17        "utf8")):
18        json_line = json.loads(line)
19        #X Data
20        xData = []
21        try:
22            xData = getSentenceVectorCNN(json_line["text"])
23        except:
24            continue
25        y = float(json_line["stars"])
26        if y < 3:
27            yData = 0
28        elif y == 3:
29            yData = 1
30        else:
31            yData = 2

```

Anschließend wird fast identisch zu dem Vorgehen im einfachen Generator aus Listing 12 der rohe Datensatz ausgelesen und entsprechend in nutzbare x und y Daten umgewandelt.

Listing 16: Zuordnung hdf5 Generator

```

31    if i < 3:

```

```
32         xTrain.append(xData)
33         yTrain.append(yData)
34     elif i == 3:
35         xVal.append(xData)
36         yVal.append(yData)
37     else:
38         xTest.append(xData)
39         yTest.append(yData)
40         i = -1
41     i += 1
```

Als nächstes werden die Daten den entsprechenden Listen angehängen. Das Vorgehen hier orientiert sich dabei an der zuvor besprochenen Aufteilung aus dem Code-Schnipsel 10.

Listing 17: Erstellen der hdf5 Datasets

```
42     if index == chunkSize - 1:
43         XTrain = hf.create_dataset("XTrain", data=
44             xTrain, maxshape=(None, 72, 100), chunks=(
45                 trainChunk, 72, 100))
46         YTrain = hf.create_dataset("YTrain", data=
47             yTrain, maxshape=(None, 3), chunks=(
48                 trainChunk, 3))
49
50         XVal = hf.create_dataset("XVal", data=xVal,
51             maxshape=(None, 72, 100), chunks=(
52                 valTestChunk, 72, 100))
53         YVal = hf.create_dataset("YVal", data=yVal,
54             maxshape=(None, 3), chunks=(valTestChunk, 3)
55         )
56
57         XTest = hf.create_dataset("XTest", data=xTest,
58             maxshape=(None, 72, 100), chunks=(
59                 valTestChunk, 72, 100))
60         YTest = hf.create_dataset("YTest", data=yTest,
61             maxshape=(None, 3), chunks=(valTestChunk, 3)
62         )
63
64         #reset Buffer-Data
65         xTrain = []
```

```
54         yTrain = []
55         xVal = []
56         yVal = []
57         xTest = []
58         yTest = []
```

Wenn das erste Mal genug Daten in den buffer-listen liegen können, diese noch nirgends appended werden, da in der entsprechenden Datei noch nichts vorliegt wo diese appended werden können. Hier zeigt sich ein weiterer Vorteil der hdf5 Dateien: Es können verschiedene Datasets, mit entsprechenden Labeln, in der gleichen Datei gespeichert und diese später auch entsprechend wieder ausgelesen werden.

Diese Datasets müssen aber zunächst erstellt werden. Dafür wird zum einen das Label des Datasets angegeben; die Dimensionen der maxshape, welche als Anzahl der Datensätze den Eintrag None bekommt, da alle vorhandenen Daten gespeichert werden sollen; die Dimensionen der chunks, wofür die eingangs berechneten Chunkgrößen nun genutzt werden; und zuletzt die tatsächlichen Daten, welche stets die gleichen Dimensionen haben müssen wie die Chunks. Abschließend werden die buffer-listen resetted, um keine Daten mehrfach zu speichern.

Listing 18: Nachfüllen der hdf5 Datasets

```
59         if (index+1) % chunkSize == 0 and index > chunkSize
60             :
61                 XTrain.resize(XTrain.shape[0]+trainChunk, axis
62                             =0)
63                 XTrain[-trainChunk:] = xTrain
64
65                 YTrain.resize(YTrain.shape[0]+trainChunk, axis
66                             =0)
67                 YTrain[-trainChunk:] = yTrain
68
69                 XVal.resize(XVal.shape[0]+valTestChunk, axis=0)
70                 XVal[-valTestChunk:] = xVal
71
72                 YVal.resize(YVal.shape[0]+valTestChunk, axis=0)
73                 YVal[-valTestChunk:] = yVal
74
75                 XTest.resize(XTest.shape[0]+valTestChunk, axis
76                             =0)
77                 XTest[-valTestChunk:] = xTest
```

```
75         YTest.resize(YTest.shape[0]+valTestChunk, axis
76                       =0)
77
78         #reset Buffer-Data
79         xTrain = []
80         yTrain = []
81         xVal = []
82         yVal = []
83         xTest = []
84         yTest = []
```

Für das appenden der Daten im folgenden Verlauf müssen die Datasets stets um die entsprechende Größe resized werden. Anschließend werden die neuen Felder entsprechend aufgefüllt und zuletzt wieder die buffer-listen resetted.

Listing 19: Behandeln der Traildaten

```
85     trainChunk = len(xTrain)
86     if trainChunk != 0:
87
88         XTrain.resize(XTrain.shape[0]+trainChunk, axis
89                       =0)
90         XTrain[-trainChunk:] = xTrain
91
92         YTrain.resize(YTrain.shape[0]+trainChunk, axis
93                       =0)
94         YTrain[-trainChunk:] = yTrain
95
96     valTestChunk = len(xVal)
97     if valTestChunk != 0:
98
99         XVal.resize(XVal.shape[0]+valTestChunk, axis=0)
100        XVal[-valTestChunk:] = xVal
101
102        YVal.resize(YVal.shape[0]+valTestChunk, axis=0)
103        YVal[-valTestChunk:] = yVal
104
105    valTestChunk = len(xTest)
106    if valTestChunk != 0:
```

```

105
106         XTest.resize(XTest.shape[0]+valTestChunk, axis
107                       =0)
108         XTest[-valTestChunk:] = xTest
109
110         YTest.resize(YTest.shape[0]+valTestChunk, axis
111                       =0)
112         YTest[-valTestChunk:] = yTest

```

Die letzten Daten, welche nicht mehr eine vollständige Chunkgröße füllen müssen nach der Schleife gesondert behandelt werden. Die resize Parameter müssen dabei zu der shape der Restdaten passen und somit entsprechend angepasst werden.

Das Auslesen der Daten erfolgt letztlich ähnlich wie bei dem anderen vorgestellten Generator:

Listing 20: hdf5 Generator

```

0 def hdf5Generator(filePath, batch_size, dataSet, loop=True)
  :
1     with h5py.File(filePath, 'r') as hf:
2         L = len(hf["X" + dataSet])
3         while True:
4             batch_start = 0
5             batch_end = batch_size
6
7             while batch_end < L:
8                 X = hf["X" + dataSet][batch_start:batch_end]
9                 Y = hf["Y" + dataSet][batch_start:batch_end]
10
11                 yield (X,Y)
12
13                 batch_start += batch_size
14                 batch_end += batch_size
15             if not loop: break

```

Die Hauptunterschiede sind dabei das hier entsprechend anfallende Zugreifen des passenden Datasets, das Wegfallen der word2vec Überführung und der zusätzliche Check ob `batch_size` über das Ende der Datasets gehen würde.

Der Vorteil bei diesem Vorgehen ist eine bessere Trainingszeit und vor allem eine bessere Validierungszeit gegenüber dem zuvor besprochenen Generator. Jedoch braucht diese Variante ein vielfaches mehr Festplattenspeicher und zwar so viel, wie in der

zuvor berechneten Gleichung 5:

$$8,000,000 \times 72 \times 100 \times 8 \approx 430 \text{ GB} \quad (5 \text{ revisited})$$

Das Erstellen dieser Datenmengen benötigt aber auch entsprechende Zeit. Dieses Vorgehen lohnt sich also besonders, wenn das Word2Vec Modell nicht mehr oder nur noch selten verändert wird und somit häufig auf dem gleichen Modell trainiert werden kann.

## 4.4 Word2Vec mit Gensim

Für das Trainieren des Word2Vec-Modells wird in dieser Arbeit die Open-Source-Library Gensim verwendet.

Gensim bietet viele hilfreiche Funktionen und die Möglichkeit, Generatoren zu verwenden. Um bei Gensim Generatoren benutzen zu können wird die Klasse `MyCorpus` geschrieben.

Listing 21: Generator für Gensim

```
0 from gensim import utils
1 import json
2 class MyCorpus:
3     def __init__(self, path):
4         self._path = path
5
6     def __iter__(self):
7         for line in open(self._path, encoding="utf8"):
8             json_line = json.loads(line)
9             yield utils.simple_preprocess(json_line["text"]
                                           ])
```

Im Konstruktor der Klasse `MyCorpus` (Zeile 3) wird der Pfad zu dem Yelp-Datensatz übergeben.

In Zeile 5 wird die Funktion `__iter__` implementiert, diese wird von Gensim erwartet und führt dazu, dass über Objekte der Klasse `MyCorpus` iteriert werden kann.

Die Funktion ist ein einfacher Generator mit der Besonderheit, dass in Zeile 9 die Funktion `utils.simple_preprocess` aufgerufen wird.

Die Funktion `utils.simple_preprocess` wird benutzt, um aus der aktuellen Review in `json_line["text"]` eine Liste von Tokens zu generieren.

Hierbei werden Wörter mit weniger als 2 Buchstaben und Wörter mit mehr als 15 Buchstaben ignoriert. Diese Grenzen können über die Parameter `min_len` und `max_len` gesetzt werden.

Listing 22: Funktion zum Laden und Erstellen des Modells

```
10 import gensim.models
11 def getWordVecModel(model_path, data_path=None):
12     try:
13         model = gensim.models.Word2Vec.load(model_path)
14         return model
```



```
15         except:
16             try:
17                 sentences = MyCorpus(data_path)
18                 model = gensim.models.Word2Vec(sentences=
19                     sentences)
20                 model.save(model_path)
21                 return model
22             except:
23                 raise ValueError('invalid model_path and
24                     data_path')
25                 return None
```

Anschließend wird in Zeile 10 die Funktion `getWordVecModel` definiert.

Die Funktion erwartet den Pfad zum Word2Vec-Modell als ersten Parameter. Der zweite Parameter ist der Datensatzpfad und wird nur benötigt, falls noch kein Word2Vec-Modell gespeichert wurde.

Zuerst wird in Zeile 12 geprüft, ob bereits ein Word2Vec-Modell existiert. Ist dies der Fall, wird dieses direkt zurückgegeben, gibt es kein Modell, so wird dieses im inneren try-Block mithilfe der `myCorpus` Klasse generiert, gespeichert und zurückgegeben.

## 4.5 Mean-Vektor-Klassifikationsmodell

Das Word2Vec-Modell bildet einen Vektorraum, in dem ähnliche Wörter nahe beieinander liegen.

Es ist somit naheliegend, dass Wörter, die mit einer negativen Rezension assoziiert werden, ein anderes Gebiet des Vektorraumes belegen als jene, die mit positiven Rezensionen assoziiert werden.

Der Gedanke dieses Modells ist es, den Mittelwert aller Wortvektoren in einer Rezension zu ermitteln und somit eine Art *Satzvektor* zu erhalten. Diese Satzvektoren werden als Eingangswerte genutzt, um mithilfe eines neuronalen Netzes eine Klassifikation der Rezensionen durchzuführen.

### 4.5.1 Implementierung

Durch die Bildung der Satzvektoren verringert sich der benötigte Speicherbedarf. Somit ist es möglich, auf die Generatoren zu verzichten und die Trainingsdaten im Arbeitsspeicher zu halten.

Dies hilft, die benötigte Trainingszeit zu reduzieren. Sollte die Arbeitsspeicherkapazität nicht ausreichen, befindet sich eine Implementierung, welche Generatoren benutzt, im Git-Repository.

Um das Word2Vec-Modell zu erhalten, wird die im Listing 22 geschriebene Funktion `getWordVecModel` importiert.

Listing 23: Satzvektorfunktion

```
0 from gensim import utils
1 from w2v_yelp_model import getWordVecModel
2 import json
3
4 model_path = "full_yelp_w2v_model"
5 data_path = "yelp_academic_dataset_review.json"
6 modelW2V = getWordVecModel(model_path)
7
8 def getSentenceVector(sentence):
9     split = utils.simple_preprocess(sentence)
10    wordVecs = []
11    for word in split:
12        try:
13            wordVecs.append(modelW2V.wv[word])
14        except:
```

```
15         pass
16     if wordVecs == []:
17         raise Exception('words not found in w2v model')
18     return np.mean(wordVecs,axis=0)
```

Die Funktion `getSentenceVector` in Zeile 8 nimmt eine Rezension entgegen und teilt diese in Zeile 9 in ihre Wörter auf. Danach wird jedes Wort geprüft, ob es im Word2Vec-Modell hinterlegt ist. Ist dies der Fall, wird der Wortvektor der Liste `wordVecs` hinzugefügt. Ist kein einziges Wort der Rezension im Word2Vec-Modell enthalten, wird eine `Exception` geworfen.

In Zeile 18 wird der Durchschnitt aller Wortvektoren zurückgeben.

Um die Daten für das Klassifikationsmodell zu generieren, muss der Datensatz in die Satzvektoren transformiert werden.

Listing 24: Daten

```
19 import numpy as np
20 from sklearn.model_selection import train_test_split
21 try:
22     X = np.load("X.npy")
23     Y = np.load("Y.npy")
24 except:
25     X = []
26     Y = []
27     for index, line in enumerate(open(data_path,encoding="
    utf8")):
28         json_line = json.loads(line)
29         #X Data
30         try:
31             X.append(getSentenceVector(json_line["text"]))
32         except:
33             continue
34         y = float(json_line["stars"])
35         if(y <3):
36             Y.append(0)
37         elif(y==3):
38             Y.append(1)
39         else:
40             Y.append(2)
41     X = np.array(X)
```

```

42     Y = np.array(Y)
43     np.save("X.npy",X)
44     np.save("Y.npy",Y)
45 X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
    test_size=0.2, random_state=42)

```

Falls die Daten bereits generiert wurden, werden diese geladen. Sind keine Daten vorhanden, wird der gesamte Datensatz durchlaufen und für jede einzelne Rezension der Satzvektor ermittelt.

Befindet sich kein Wort der Rezension im Word2Vec-Modell so wird diese Rezension in Zeile 33 übersprungen. Die Bewertungen werden wie folgt codiert: negativ (0), neutral (1) und positiv (2). Anschließend werden die generierten Daten gespeichert, um bei mehrfacher Ausführung Zeit zu sparen. Danach werden die Daten in Zeile 45 in Trainings- und Testdaten aufgeteilt, hierfür wird die Funktion `train_test_split` von `sklearn` verwendet.

Die Rezensionen liegen jetzt in Form von Satzvektoren bzw. genau genommen Rezensionsvektoren vor. Anhand dieser Daten kann nun ein neuronales Netz trainiert werden.

Listing 25: Neuronales Netz

```

46 from tensorflow.keras.models import Sequential
47 from tensorflow.keras.layers import Dense
48 from tensorflow import keras
49
50 modelNN = Sequential()
51 modelNN.add(Dense(50,activation='relu',input_dim=X[0].size)
    )
52 modelNN.add(Dense(15,activation='relu'))
53 modelNN.add(Dense(3,activation='softmax'))
54 modelNN.compile(optimizer='adam',loss='
    sparse_categorical_crossentropy',metrics=["
    sparse_categorical_accuracy"])

```

Hier wurde ein dichtes, mehrschichtiges neuronales Netzwerk verwendet, wobei die Eingangsdimension der Dimension der Satzvektoren (in diesem Fall 100) entspricht. Bis auf die Ausgangsschicht wurde für alle Schichten des neuronalen Netzwerks die Aktivierungsfunktion `relu` verwendet. Die Ausgangsschicht selbst verwendet die Aktivierungsfunktion `softmax`.

In dieser Implementation wurden die hinteren Schichten, wie bei Klassifikationen üblich, bewusst kleiner gewählt, um eine Verdichtung der Informationen zu erzwingen.

Nachdem die Struktur des neuronalen Netzwerkes nun feststeht, muss das Netz noch kompiliert werden. Hierzu werden in Zeile 54 einige Parameter gesetzt. Für den Optimierungsalgorithmus wird `adam` gewählt. Für die Fehlerfunktion, hier `loss` genannt, wird `sparse_categorical_crossentropy` benutzt.

Die Fehlerfunktion `sparse_categorical_crossentropy` ermöglicht es im Gegensatz zur `binary_crossentropy` mehr als zwei Klassen zu klassifizieren. Eine weitere mögliche Fehlerfunktion ist die `categorical_crossentropy`. Hierzu müssten bloß die Zielwerte *One-Hot-Encoded* werden. Zuletzt wird für den Parameter `Metrics` der Wert `sparse_categorical_accuracy` übergeben.

Nach dem Kompilieren des neuronalen Netzes ist jetzt das Trainieren, also das Optimieren der Gewichte, der nächste Schritt.

Listing 26: Neuronales Netz - Trainieren

```
55 earlystop = keras.callbacks.EarlyStopping(monitor='  
    val_sparse_categorical_accuracy',patience=10,verbose=  
    False,restore_best_weights=True)  
56 cbList = [earlystop]  
57 count = np.unique(Y_train,return_counts=True)[1]  
58 cWeight = 1/(count/Y_train.size)  
59 hist = modelNN.fit(X_train,Y_train,epochs=1000,  
    validation_split=0.2,batch_size=2048,class_weight={0:  
    cWeight[0],1:cWeight[1],2:cWeight[2]},callbacks=cbList
```

Um ein Overfitting zu vermeiden und die Trainingszeit zu minimieren, wird in Zeile 55 ein *Early-Stop-Callback* definiert. Der vorzeitige Stopp des Trainierens tritt ein, wenn 10 Epochen lang keine Verbesserung auf der Validierungsmenge aufzuzeigen ist. Hierzu wird die Metrik `val_sparse_categorical_accuracy` überwacht. Zusätzlich werden die Gewichte, die zu dem besten Ergebnis geführt haben, nach dem vorzeitigen Stopp wieder hergestellt.

In Zeile 59 findet das tatsächliche Training des neuronalen Netzes statt. Hierfür werden der Methode `fit` die Trainingsdaten übergeben. Der Parameter `validation_split = 0.2` sorgt dafür, dass 20% der Trainingsdaten als Validierungsmenge genommen werden. Hierbei ist aber anzumerken, dass es sich bei den Trainingsdaten ohnehin schon nur um 80% des Datensatzes handelt, weswegen die tatsächliche Validierungsmenge 16% des Datensatzes beinhaltet.

Die `batch_size` wird aufgrund des großen Datensatzes auf 2048 gesetzt, um das Optimieren der Gewichte zu beschleunigen. Eine kleinere `batch_size` hat beim Experimentieren in diesem Fall nicht zu einer schnelleren Konvergenz geführt.

Wie in Kapitel 4.2.1 diskutiert gibt es verschiedene Umgänge mit unausgeglichene Datensätzen. Die für diese Arbeit gewählte Methode der Gewichtung ist hier zu erkennen an den, an die Methode `fit` übergebenen, Klassengewichte.

Listing 27: Neuronales Netz - Evaluieren

```
60 modelNN.evaluate(X_test, Y_test)
```

Um das Modell zu evaluieren, wird in Zeile 60 die Methode `evaluate` mit den Testdaten aufgerufen. Das hier trainierte neuronale Netzwerk klassifiziert 80.02 % der Testdaten richtig.

#### 4.5.2 Konfusionsmatrix

Die bereits berechnete Genauigkeit ist ein gutes erstes Leistungsmerkmal des Klassifikators. Um den Klassifikator noch besser einschätzen zu können, lohnt es sich, eine Konfusionsmatrix aufzustellen. Hierfür wird die Funktion `confusion_matrix` von `sklearn` benutzt.

Listing 28: Konfusionsmatrix

```
61 from sklearn.metrics import confusion_matrix
62 y_pred = np.argmax(modelNN.predict(X_test), axis=-1)
63 confusion_matrix(Y_test, y_pred, normalize='true')
```

-	Negativ	Neutral	Positiv
Negativ	0.8046	0.1727	0.0228
Neutral	0.1682	0.6844	0.1474
Positiv	0.0242	0.1586	0.8172

Tabelle 2: Konfusionsmatrix mit Klassengewichtung

Wie in Tabelle 2 zu sehen ist, werden negative Bewertungen zu 80.46 % richtig als negative Bewertung klassifiziert, 17.27 % werden als neutral und 2.28 % werden als positiv klassifiziert.

Neutrale Bewertungen werden zu 68.44 % richtig klassifiziert, jedoch werden 16.82 % der neutralen Bewertungen falsch als negativ und zu 14.74 % falsch als positiv klassifiziert.

Die positiven Bewertungen werden zu 81.72 % richtig als positive Bewertungen klassifiziert, zu 15.86 % als neutral und zu 2.42 % als negativ klassifiziert.

Das gleiche Modell ohne die Gewichtung der Klassen erreicht eine Genauigkeit von 85.7 %. Betrachtet man jedoch die Konfusionsmatrix in Tabelle 3 so sieht man, dass dort bloß 27 % der neutralen Rezensionen richtig klassifiziert wurden.

-	Negativ	Neutral	Positiv
Negativ	0.8567	0.0602	0.0831
Neutral	0.2571	0.2708	0.4720
Positiv	0.0249	0.0235	0.9516

Tabelle 3: Konfusionsmatrix ohne Klassengewichtung

## 4.6 Word2Vec-CNN-Modell

Ein *convolutional neural network*, kurz CNN, ist in der Lage lokale Muster in einer Sequenz zu erlernen und diese später wiederzuerkennen, auch wenn diese an einer anderen Stelle auftreten.

In diesem Modell wird eine Reihe an Wortvektoren an das CNN übergeben, das Extrahieren der Merkmale wird somit Teil des Optimierungsproblems.

### 4.6.1 Implementierung

Aufgrund des sehr hohen Speicherbedarfs wird der im Kapitel 4.3.1 geschriebene Generator verwendet.

Hierfür muss jedoch zunächst die Funktion `getSentenceVectorCNN` implementiert werden.

Listing 29: Datentransformation

```

0 import numpy as np
1 from gensim import utils
2 from w2v_yelp_model import getWordVecModel
3
4 model_path = "full_yelp_w2v_model"
5 modelW2V = getWordVecModel(model_path)
6
7 def getSentenceVectorCNN(sentence):
8     split = utils.simple_preprocess(sentence)
9     wordVecs = np.zeros((72,100))
10    i=0
11    for word in split:
12        if i == 72: break
13        try:
14            wordVecs[i] = modelW2V.wv[word]
15            i += 1
16        except:

```

```

17         pass
18     if np.all(wordVecs[5:] == 0):
19         raise Exception('not enough words found in w2v
           model')
20     return wordVecs

```

Zuerst wird in Zeile 5 die im Listing 22 geschriebene Funktion `getWordVecMode` aufgerufen, um das Word2Vec-Modell zu erhalten.

Die Funktion `getSentenceVectorCNN` in Zeile 7 nimmt eine Rezension entgegen und transformiert diese in eine für das CNN verarbeitbare Form.

Dafür wird die Rezension in Zeile 8 tokenisiert, sodass in der Variable `split` eine Liste der Rezensionswörter gespeichert ist. Die Vorverarbeitung und Trennung der Wörter erfolgt hier durch die Funktion `utils.simple_preprocess`.

Danach wird in Zeile 9 das Numpy-Array `wordVecs` mit Nullen initialisiert. Dieses Numpy-Array speichert die ersten 72 Wortvektoren der Rezension, die Zahl 72 wurde aufgrund des im Kapitel 4.1.3 berechneten Median der Reviewlänge gewählt.

Die for-Schleife in Zeile 11 durchläuft alle Wörter der Liste `split` und fügt diese, falls sie im word2Vec-Modell vertreten sind, in das `wordVecs` Array ein.

Sind mehr als 72 Wörter in der Liste `split` wird die Schleife abgebrochen, wenn das Array vollständig gefüllt ist.

Sollten sich weniger als 72 Wörter in der Liste befinden, so ist das Array bereits mit Nullen initialisiert.

Dies kann als *Truncating* und *Padding* bezeichnet werden und ist notwendig, da das CNN eine feste Dimension der Eingangsdaten benötigt.

Anschließend wird in Zeile 18 geprüft, ob weniger als 5 Wortvektoren in dem Array stehen. Ist dies der Fall, wird eine Exception geworfen.

Nach dem Implementieren der Funktion `getSentenceVectorCNN` muss das CNN erstellt werden.

Listing 30: CNN

```

0 import numpy as np
1 from tensorflow.keras.models import Sequential
2 from tensorflow.keras.layers import Dense, Flatten
3 from tensorflow.keras.layers import Conv1D, MaxPooling1D
4 from tensorflow import keras
5

```



```
6 modelNN = Sequential()
7
8 modelNN.add(Conv1D(150, kernel_size=5, activation='relu',
    input_shape=((72, 100))))
9 modelNN.add(MaxPooling1D(pool_size=4))
10 modelNN.add(Conv1D(100, kernel_size=3, activation='relu'))
11 modelNN.add(MaxPooling1D(pool_size=4))
12 modelNN.add(Flatten())
13 modelNN.add(Dense(300, activation='relu'))
14 modelNN.add(Dense(100, activation='relu'))
15 modelNN.add(Dense(50, activation='relu'))
16 modelNN.add(Dense(10, activation='relu'))
17 modelNN.add(Dense(3, activation='softmax'))
18 modelNN.compile(optimizer='adam', loss='
    sparse_categorical_crossentropy', metrics=["
    sparse_categorical_accuracy"])
19 modelNN.summary()
```

Hier wurden dem Netzwerk zwei *Conv1D-Schichten* hinzugefügt, mit der Aktivierungsfunktion `relu`. Die erste Schicht hat hier 150 Filter und eine Kernelgröße von fünf. Die zweite Schicht hat hingegen 100 Filter und eine Kernelgröße von drei.

Die *MaxPooling1D-Schichten* dienen dazu die Informationen nach den Faltungen zu verdichten.

Die Matrix, die als Ergebnis der zweiten *MaxPooling1D-Schicht* entsteht, wird anschließend in Zeile 12 in einen Vektor umgewandelt. Dieser Vektor wird als Eingang des dichten neuronalen Netzes benutzt. Für das dichte neuronale Netz wird hier wieder der Ansatz verfolgt, die Schichten inkrementell zu verkleinern, um eine Verdichtung der Information zu erzwingen.

Zuletzt muss das neuronale Netz noch kompiliert werden, was in Zeile 18 geschieht. Hierbei wird wie gehabt der optimizer `adam` gewählt und die Fehlerfunktion `sparse_categorical_crossentropy` sowie die Metrik `sparse_categorical_accuracy` verwendet.

Die Zusammenfassung des Modells, die durch Zeile 19 erzeugt wurde, befindet sich im Listing 31.

Listing 31: CNN - Summary

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
conv1d (Conv1D)	(None, 68, 150)	75150
max_pooling1d (MaxPooling1D)	(None, 17, 150)	0
conv1d_1 (Conv1D)	(None, 15, 100)	45100
max_pooling1d_1 (MaxPooling1D)	(None, 3, 100)	0
flatten (Flatten)	(None, 300)	0
dense (Dense)	(None, 300)	90300
dense_1 (Dense)	(None, 100)	30100
dense_2 (Dense)	(None, 50)	5050
dense_3 (Dense)	(None, 10)	510

Um das Modell mit Generatoren zu trainieren, muss aufgrund der für das Netzwerk unbekannten Größe der Trainingsmenge angegeben werden, wie viele Schritte pro Epoche erfolgen sollen.

Listing 32: CNN - Fitting

```

20 from hdf5 import hdf5Generator
21 num_rows = 4.8E6
22 batchSize = 256
23 steps = num_rows/batchSize
24 #early stop
25 earlystop = keras.callbacks.EarlyStopping(monitor='
    val_sparse_categorical_accuracy',patience=8,verbose=
    False,restore_best_weights=True)
26 cbList = [earlystop]
27 trainData = hdf5Generator("w2vCNN.hdf5", batchSize, "Train"
    )
28 valData = hdf5Generator("w2vCNN.hdf5", batchSize, "Val")
29
30 ###fit
31 cW = {0:4.18,1:9.51,2:1.52}

```

```
32 hist = modelNN.fit(trainData, validation_data=valData,
    epochs=50, class_weight=cW, steps_per_epoch=steps,
    validation_steps=int(steps/3), callbacks=cbList
33
34 testData = hdf5Generator(path + "w2vCNN.hdf5", batchSize, "
    Test", loop=False)
35 modelNN.evaluate(testData)
```

Hierfür wird die Größe der Trainingsmenge durch die BatchSize geteilt (Zeile 24). Um ein Overfitting zu verhindern wird, wie beim Mean-Modell auch schon in Zeile 25, ein Earlystop definiert. Dieser unterbricht das Training vorzeitig, falls acht Epochen lang keine Verbesserung auf der Validierungsmenge erzielt wird.

Die Generatoren für die Trainings- und Testmenge werden in Zeile 27 und 28 variablen gespeichert.

Die Klassengewichte in Zeile 31 wurden im Listing 33 berechnet. Mit allen Anforderungen erfüllt kann jetzt in Zeile 32 das Modell trainiert werden. Dabei ist zu beachten, dass die Anzahl der Schritte pro Epoche für die Validierungsmenge gedrittelt wird, da die Trainingsmenge  $\frac{3}{5}$  des Datensatzes ausmacht die Validierungsmenge hingegen nur  $\frac{1}{5}$  wird trotzdem nach jeder Epoche die gesamte Validierungsmenge evaluiert.

Nach dem Training des Netzes ist es an der Zeit das Modell zu evaluieren. Dafür wird in Zeile 34 zunächst wieder ein Generator angelegt. Mit dem Parameter `loop=False` wird gewährleistet, dass der Generator den Datensatz nur einmal durchläuft. Dieser Generator kann nun in Zeile 35 an die Methode `evaluate` übergeben werden. Das Evaluieren zeigt, dass dieses Modell zu 81.44 % die Klassen der Testmenge richtig klassifiziert.

Listing 33: CNN - Klassengewichte

```
Y_train=[]
gen = hdf5Generator(path + "w2vCNN.hdf5", batchSize, "Test"
    ,loop=False)
for (x,y) in gen:
    Y_train.append(y)
Y_train = np.array(Y_train).flatten()
count = np.unique(Y_train,return_counts=True)[1]
cWeight = 1/(count/Y_train.size)
```

### 4.6.2 Konfusionsmatrix

Um einen besseren Eindruck über die Qualität des Netzes zu erhalten, wird auch bei diesem Modell wieder eine Konfusionsmatrix erstellt.

Listing 34: CNN - Konfusionsmatrix

```

36 from sklearn.metrics import confusion_matrix
37 tD = hdf5Generator(path + "w2vCNN.hdf5", batchSize, "Test",
    loop=False)
38 y_pred = np.argmax(modelNN.predict(tD),axis=-1)
39 y_test=[]
40 for (x,y) in hdf5Generator(path + "w2vCNN.hdf5", batchSize,
    "Test",loop=False):
41     y_test.append(y)
42 y_test = np.array(y_test).flatten()
43
44 confusion_matrix(y_test,y_pred,normalize='true')
```

Dafür werden die vom Netz vorhergesagten Klassen der Testmenge in dem Vektor `y_pred` gespeichert und die tatsächlichen Klassen der Testmenge in dem Vektor `y_test`. Beide Vektoren werden der Funktion `confusion_matrix` übergeben, welche die in der Tabelle 4 dargestellte Konfusionsmatrix zurückliefert.

-	Negativ	Neutral	Positiv
Negativ	0.8314	0.1377	0.0308
Neutral	0.1970	0.6354	0.1676
Positiv	0.0316	0.1316	0.8367

Tabelle 4: Konfusionsmatrix mit Klassengewichtung

Aus der Tabelle 4 lässt sich ablesen, dass negative Bewertungen zu 83.14 % richtig als negative Bewertung klassifiziert werden, zu 13.77 % als neutrale Bewertung und zu 3.08 % als positive Bewertung.

Neutrale Bewertungen werden zu 63.54 % richtig als neutrale Bewertung klassifiziert, zu 19.70 % als positive Bewertung und zu 16.76 % als negative Bewertung.

Positive Bewertungen werden zu 83.67 % richtig als positive Bewertung klassifiziert, zu 13.16 % als neutrale Bewertung und zu 3.16 % als negative Bewertung.

## 5 Ergebnisse

### 5.1 Subjektivität

Wichtig für machine learning algorithms ist es, einen möglichst objektiven Datensatz zu benutzen, auf dem letztlich trainiert wird. Je nach konkretem Szenario und Fragestellung kann sich dies als schwierig gestalten. So ist z.B. *Racial Profiling* durch künstliche Intelligenz ein viel diskutiertes Thema in diesem Sektor.

Die in dieser Arbeit verwendete Methode Word2Vec ist besonders anfällig für solche Probleme. Schließlich sollen die Wörter anhand ihrer Bedeutung gruppiert und in Relation zueinander eingeordnet werden. Ist der Datensatz auf dem diese Cluster-Bildung basiert durch Vorurteile oder gar Diskriminierung betroffen, spiegelt sich dies auch in den wordembeddings wieder. Dafür muss dies nicht einmal gezielt in den Ausgangsdaten vorliegen, auch ein unvollständiger Datensatz kann zu solchen Ergebnissen führen. Ein direktes Beispiel dafür findet sich in Kapitel 3.1.4 in Abbildung 1. Hier wurden die Begriffe erfolgreich in männlich und weiblich unterteilt. Dazu gehören aber auch die verwendeten Adjektive *wise* und *pretty*, welche laut des Programms eher weibliche Begriffe seien, während *strong* ein männlicher Begriff sei. Dies stellt der Datensatz letztlich auch so dar. Es ist anzunehmen, dass diese Begriffe auch von Menschen vermutlich in dieser Weise eingeordnet worden wären. Mit einem umfassenden Datensatz wären diese Tendenzen aber möglicherweise schwächer und allgemein etwas neutraler ausgefallen.

Gleichzeitig sind diese Subjektivitäten aber auch in vielen Fällen wichtig. Sprache ist in der Regel meinungsbehaftet und viele machine learning Projekte beschäftigen sich damit, wie auch dieses Projekt zu einem gewissen Grad. Um Reviews in entsprechende Kategorien einzuteilen, müssen die Wörter anhand ihrer gebräuchlichen Nutzung eingeordnet werden. Um kontextspezifische Unterschiede zu vermeiden sollte sich idealerweise die Erstellung der Wortvektoren aufgrund dieser Nutzung auf das vorliegende Szenario beziehen. Wenn ein Wort also entgegengesetzt der ursprünglichen Bedeutung in einem völlig anderen Kontext verwendet wird, stellt dies eine wichtige Information dar, welche für die Auswertung beachtet werden muss. Jedoch ist es aufwändig entsprechend zugeschnittene wordembeddings zu erstellen. Weiterhin sind davon viele Wörter nicht betroffen, wodurch ein verhältnismäßig großer Mehraufwand für eine recht geringe Menge an Wörtern entsteht. Man kann ebenfalls davon ausgehen, dass das Modell in vielen Fällen diese Probleme entsprechend optimiert, sodass diese Wörter letztlich korrekt interpretiert werden.

Schreibstile unterscheiden sich von Person zu Person. Besonders in einem informellen Raum wie es das Internet häufig ist zeigt sich dies. Gewisse Wortwahl und Zeichensetzung ist für die einen ein eindeutiges Zeichen für Sarkasmus, während andere die-

se Aussagen ernst nehmen. Auch Akzente und Dialekte spiegeln sich teilweise durch Slangwörter oder sonstige Besonderheiten im schriftlichen wieder. Dies kann verschiedene Auswirkungen auf die Kategorisierung von Texten haben. Eine mögliche Szenario wäre, dass ein spezieller Dialekt häufig in negativen Reviews vorkommt. Die Ursache dafür könnte dabei einen spezifischen Grund haben oder auch rein zufällig sein. Unabhängig davon kann dies jedoch dazu führen, dass das Programm diese *Korrelation* entdeckt und eine *Kausalität* herstellt. Folglich werden alle Reviews, welche diesen Dialekt enthalten als negativ kategorisiert, darunter auch solche, welche tatsächlich positiv sind.

## 5.2 Auswertung

Im Folgenden werden die Ergebnisse der Mean- und der CNN-Methode ausgewertet und gegenübergestellt. Zunächst wird jedoch der Einfluss der Trainingsgewichte untersucht.

### 5.2.1 Gewichte

Das Mean-Vektor-Klassifikationsmodell wurde sowohl mit als auch ohne Trainingsgewichte getestet. Dabei hat die Variante ohne extra Gewichte ein allgemein besseres Ergebnis mit 85.70 % gegenüber 80.02 % ohne Gewichte. Die Konfusionsmatrizen aus den Tabellen 2 und 3 zeigen jedoch, dass die Verteilung der richtig bestimmten Reviews mit Gewichten gleichmäßiger ist. Ohne Gewichte werden mehr negative und positive Reviews richtig klassifiziert, aber die Anzahl der richtig bestimmten neutralen Reviews beträgt gerade einmal knapp über 27 % gegenüber 68 % mit Gewichten. Außerdem werden mehr als dreimal so viele neutrale Reviews als positiv identifiziert, wenn die Gewichte weg gelassen werden. Unter Berücksichtigung der ursprünglichen Distribution des Datensatzes, wie zuvor in Bild 2 gezeigt, wird auch deutlich, wieso diese 'Entscheidung' für den Algorithmus von Vorteil ist. Es gibt eine hohe Anzahl an positiven Reviews, während es verhältnismäßig wenig negative, dafür aber nur einen Bruchteil an neutralen Reviews gibt. Wie bereits in Kapitel 4.2.1 angesprochen entsteht dadurch unweigerlich ein Bias zu einer Identifizierung als positive Review, da dies im Gesamtergebnis zu einem größeren Erfolg führt. Es lässt sich also erkennen, dass die Gewichtung der Daten durchaus den gewünschten Effekt erzielt, jedoch darunter das Gesamtergebnis etwas gelitten hat.

### 5.2.2 Mean vs. CNN

Für die Gegenüberstellung der verschiedenen Methoden werden nur die Ergebnisse der Mean Methode mit Gewichten berücksichtigt, um so ein ausgeglicheneres Ausgangsszenario zu erhalten. Hier lässt sich zwar erkennen, dass die CNN Methode geringfügig

besser abgeschnitten hat, mit 81.44 % gegenüber 80.02 %, jedoch ist dieser Unterschied minimal. Die Methoden können daher als gleichwertig betrachtet werden. Auch die Konfusionsmatrizen 2 und 4 sind annähernd identisch. Das CNN Modell hat lediglich wieder weniger Wert auf neutrale Reviews gelegt und stattdessen eine positive und negative Kategorisierung bevorzugt.

Aufgrund der ähnlichen Ergebnisse ist jedoch die Mean Methode deutlich zu bevorzugen. Zum einen benötigt diese in der Regel weniger Vorbereitung, hier in Form der Generatoren. Das liegt daran, dass die Mean Methode schwächer skalierende Anforderungen an die Hardware hat. Dadurch ist auch die Trainingszeit um ein vielfaches schneller mit ca. 10s pro Epoche gegenüber ungefähr 7min. Der Großteil dieses Unterschieds entsteht dabei durch das Lesen von der Festplatte durch die Generatoren anstelle des Arbeitsspeichers. Aber auch Konfigurationsunterschiede sind dabei zu beachten, z. .B nutzt der Mean eine wesentlich größere batchsize von 2048. Im Vergleich arbeitet das CNN nur mit einer Größe von 256.

Allerdings bietet das CNN Modell mehr Möglichkeiten an der Struktur zu arbeiten als die Mean Methode. Während am Mean nur wenige Änderungen vorgenommen werden könnten, bietet das CNN ein wesentlich größeres Potenzial. Es besteht also durchaus die Möglichkeit, dass ein ausführlicheres Experimentieren mit dem Aufbau des CNN zu konstant besseren Ergebnissen führt.

### 5.2.3 Problemstellung

Die relativ geringe Erfolgsquote der Modelle, aber z. B. auch die Schwierigkeiten mit neutralen Reviews zeigen sicherlich, dass es noch Optimierungsmöglichkeiten gibt. Dennoch liegt ein großer Teil der Herausforderung auch an der eigentlichen Problemstellung. Die meisten Menschen hätten vermutlich auch Schwierigkeiten dabei eine 2 Sterne Review, sprich eine negative, von einer 3 Sterne, also neutralen, zu unterscheiden. Zum einen sind die Übergänge recht fließend. Zum Anderen sind Menschen aber auch einfach unterschiedlich: Der gleiche Text mit den identischen Argumenten könnte für eine Person eine 3 Sterne und für die nächste eine 4 Sterne Review sein. Aus der Sicht des Modells gäbe es also 2 Einträge mit identischen Merkmalen, welche jedoch unterschiedlichen Kategorien zugeordnet werden.

### 5.3 Fazit

Zusammenfassend lässt sich sagen, dass Word2Vec ein valider erster Schritt zur Klassifizierung von Reviews ist. Die Ergebnisse zeigen jedoch, dass noch Optimierungsbedarf herrscht. Dabei ist es aber unklar, ob für die genutzten Modelle idealere Strukturen möglich sind oder ob diese Werte eine Obergrenze für diese Problemstellung in Kombination mit ebendiesen Modellen darstellen. Davon ausgehend, dass die Ergebnisse eine Obergrenze darstellen, ist die Mean Methode aufgrund der in Kapitel 5.2.2 besprochenen Punkte deutlich dem CNN Ansatz vorzuziehen.

Das deutliche Hauptproblem der Klassifizierung stellen die neutralen Reviews dar. Ein möglicher Ansatz wäre die Fragestellung auf eine Kategorisierung in positiv und negativ zu beschränken. Dadurch wird das Problem jedoch nicht gelöst sondern lediglich verschoben. Eine weitere Alternative wäre das Ignorieren von neutralen Reviews, sodass nur mit 1 und 2 sowie 4 und 5 Sterne Reviews gearbeitet wird. Dies wäre aber nur sinnvoll, wenn der geplante Nutzen dieser Klassifizierung das zulässt. Dadurch, dass die Anzahl der neutralen Reviews jedoch ohnehin verhältnismäßig sehr gering ist könnte dies z. B. in der Marketinganalyse durchaus eine Option sein.



## Literatur

- [Cho18] François Chollet. *Deep learning mit Python und Keras: das Praxis-Handbuch vom Entwickler der Keras-Bibliothek*. mitp, 2018.
- [KBB20] Jon Krohn, Grant Beyleveld und Aglaé Bassens. *Deep Learning illustriert: eine anschauliche Einführung in Machine Vision, Natural Language Processing und Bilderzeugung für Programmierer und Datenanalysten*. dpunkt, 2020.
- [WS19] Ludwig Wittgenstein und Joachim Schulte. *Philosophische Untersuchungen*. Suhrkamp Verlag, 2019.

Das Projekt und die zugehörigen Dateien sind im folgenden Git zu finden:

<https://gitlab.cvh-server.de/w2v/w2vp>

## Eidesstattliche Erklärung

Hiermit versichern wir, dass wir die Hausarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Datum

8.8.21

Unterschrift der Verfasser

J. Schump

Johannes Schump

Schump