

Information Retrieval bei Gesetzestexten:
TF-IDF vs. fastText

Individuelles Projekt der GymInf-Ausbildung

betreut durch

Dr. Sven Helmer,
Universität Zürich

Author: Oliver Baltisberger
Matrikelnummer: 05-413-588
Adresse: Wiesenstrasse 33
4900 Langenthal
E-Mail: oliver.baltisberger@gmail.com
Datum: 13. Juli 2022

Inhaltsverzeichnis

1	Einleitung	5
2	Einlesen und Vorverarbeitung	6
2.1	Einlesen und Vorverarbeitung der Dokumente	6
2.2	Einlesen und Vorverarbeitung der Abfrage	9
3	TF-IDF	11
3.1	TF-IDF-Vektoren	11
3.2	Kosinus-Ähnlichkeit	15
3.3	Umsetzung in Python	16
4	FastText	19
4.1	Textklassifizierung mit fastText	19
4.2	Textdarstellung mit fastText	20
4.3	Umsetzung in Python	22
5	Evaluation	25
6	Schlussfolgerungen	28
A	Weitere Listings	31
B	Truncated SVD	41
C	Kurzanleitung	43

Abbildungsverzeichnis

1	Art. 40a OR	8
2	Webapplikation	9
3	Webapplikation mit Resultat für Beispielabfrage	10
4	TF-Gewicht	12
5	IDF-Gewicht	13
6	TF-IDF-Gewicht	13
7	Kosinus-Ähnlichkeit zwischen zwei Vektoren	15
8	Zwei Textklassifizierungsbeispiele	19
9	CBOW-Modellarchitektur	20
10	CBOW vs. SKIPGRAM	21
11	True Positive, False Positive, True Negative, False Negative	25
12	Evaluation	27
13	Evaluation (Truncated SVD)	42

Tabellenverzeichnis

1	TF-IDF-Vektoren für das Beispiel (nicht normalisiert)	14
2	TF-IDF-Vektoren für das Beispiel (normalisiert)	14
3	Kosinus-Ähnlichkeiten für das Beispiel TF-IDF	16
4	Top-10-Tupel für die Beispielabfrage (TF-IDF)	18
5	Nearest Neighbors und Wortanalogien mit fastText	22
6	Kosinus-Ähnlichkeiten für das Beispiel TF-IDF mit fastText	23
7	Top-10-Tupel für die Beispielabfrage (fastText)	24
8	Evaluationsabfragen und passende Gesetzesartikel	25

Listings

1	Einlesen und Vorverarbeitung der Gesetzesartikel	7
2	Python-Code zur Webapplikation	10
3	Artikelsortierung nach absteigender Relevanz (TF-IDF)	17
4	Hilfsfunktion für Tupel-Generierung	17
5	Wortvektoren, Nearest Neighbors und Wortanalogien mit fastText	22
6	Beispiel TF-IDF mit fastText	23
7	Artikelsortierung nach absteigender Relevanz (fastText)	24
8	Auslesen des Gesetzesartikels (formatiert)	31
9	Auslesen des Gesetzesartikels (unformatiert)	32
10	Linguistische Vorverarbeitung	33
11	HTML-Code zur Webapplikation	34
12	CSS-Code zur Webapplikation	36
13	Top-n-Artikel und Artikel über Schwellenwert	38
14	Evaluation	39
15	Truncated SVD	41

1 Einleitung

”Everything is Information, and Information is Everything” [11] bringt auf den Punkt, welche Rolle Informationen heute spielen. Firmen wie Google, Facebook und Amazon haben aus dem Sammeln und Auswerten von Daten ein erfolgreiches Geschäftsmodell gemacht. Daten alleine werden aber erst dann wirklich nützlich, wenn man gezielt darauf zugreifen kann, um ein bestimmtes Informationsbedürfnis zu befriedigen. Ebenfalls charakteristisch für die heutige Welt ist, dass immer mehr Daten unstrukturiert (z.B. als Textdokumente, Emails, Websites) anstatt strukturiert (z.B. in einem Datenbanksystem) vorliegen [23]. Entsprechend gewinnt die Informationsabfrage (Information Retrieval), d.h. die semantische Suche in unstrukturierten Dokumenten, gegenüber der Datenabfrage (Data Retrieval), d.h. der syntaktischen Suche in stark strukturierten Daten, immer mehr an Bedeutung.

Manning et al. [15, S. 1] definieren Information Retrieval folgendermassen:

Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers).

Somit kann IR auch so verstanden werden, dass man genau die Information findet, die man braucht, und zwar dann, wann man diese braucht. Es wird schnell klar, dass wir heutzutage – im kleinen Rahmen – täglich IR betreiben (z.B. Suche im Internet, Durchsuchen von Emails).

Auch in der Schule wird immer wieder IR betrieben. Ein typisches Problem im Rechtunterricht besteht bspw. darin, für eine juristische Fragestellung den passenden Gesetzesartikel zu finden (Beispiel: Was ist der Ferienanspruch eines Arbeitnehmers pro Jahr? Antwort: Mindestens 4 Wochen gemäss Art. 329a OR). Dies ist ein klassisches IR-Problem, da Gesetzestexte typischerweise nicht strukturiert in einem Datenbanksystem, sondern unstrukturiert in Textform vorliegen.

Dieses Projekt widmet sich diesem Problem. Konkret wird untersucht, ob fastText [7], eine Bibliothek zur effizienten Klassifizierung und Darstellung von Text, für juristische Nutzerabfragen bessere Gesetzesartikel zurückliefert als eine einfache TF-IDF-Schlagwortsuche. Dabei werden nur das Schweizer Obligationenrecht (OR) und Zivilgesetzbuch (ZGB) berücksichtigt, weil diese beiden Gesetze für den Rechtunterricht am Gymnasium am wichtigsten sind.¹

Der Rest dieses Berichts ist wie folgt aufgebaut: Kapitel 2 beschreibt, wie die Dokumente (d.h. Gesetzesartikel) und die Nutzerabfrage in Python eingelesen und linguistisch vorverarbeitet werden. Kapitel 3 zeigt, wie die Dokumente und Abfrage als TF-IDF-Vektoren dargestellt werden können, um mithilfe der Kosinus-Ähnlichkeit die passendsten Gesetzesartikel für eine Abfrage zu bestimmen. Kapitel 4 macht dasselbe mit fastText: Dokumente und Abfrage werden als fastText-Vektoren dargestellt und mittels Kosinus-Ähnlichkeit verglichen. Kapitel 5 evaluiert die zwei Ansätze (TF-IDF und fastText), währenddessen Kapitel 6 das Wichtigste zusammenfasst und Schlussfolgerungen zieht.

¹Im OR sind u.a. alle kaufmännischen Vertragsarten geregelt (z.B. Kauf-, Miet- und Arbeitsvertrag), währenddessen das ZGB Regelungen zum Personen-, Familien-, Erb- und Sachenrecht enthält.

2 Einlesen und Vorverarbeitung

Um die Ähnlichkeit zwischen den Dokumenten (Gesetzesartikeln) und der Abfrage bestimmen zu können, müssen diese zuerst in Python eingelesen und linguistisch vorverarbeitet werden.

2.1 Einlesen und Vorverarbeitung der Dokumente

Die 2'556 OR- bzw. ZGB-Artikel sind die Dokumente in diesem Projekt. Zuerst habe ich das OR [3] und ZGB [4] als HTML-Datei auf meinem Computer gespeichert. Danach habe ich die beiden Gesetze in Python eingelesen und linguistisch vorverarbeitet (siehe Listing 1).

In Listing 1 werden zuerst die gespeicherten HTML-Dateien mithilfe von BeautifulSoup [18] in Python eingelesen (Zeilen 11-13). Dabei werden nur die article-Tags berücksichtigt, da diese die einzelnen Gesetzesartikel enthalten (siehe Abbildung 1b). Anschliessend werden die ID und der Name des Artikels bestimmt (Zeilen 18-20) und nur die relevanten Artikel berücksichtigt (Zeilen 22-29). Es sind nämlich nicht alle Artikel relevant, weil am Ende der Gesetze immer noch Übergangs- bzw. Schlussbestimmungen angehängt sind. Da die Artikelnummerierung bei diesen Übergangs- bzw. Schlussbestimmungen wieder bei 1 anfängt, enden die relevanten Artikel sobald die aktuelle Artikelnummer kleiner ist als die vorherige Artikelnummer. Die Funktion `get_formatted_article()` (siehe Listing 8) liefert den Gesetzesartikel als formatierten String zurück (Zeile 31). Falls der formatierte Gesetzesartikel nicht leer (d.h. nicht aufgehoben) ist, wird mit der Funktion `get_article_text()` (siehe Listing 9) der Artikeltext ausgelesen (Zeile 35) und mit der Funktion `get_preprocessed_text()` (siehe Listing 10) linguistisch vorverarbeitet (Zeile 36). Schlussendlich wird der Artikel in Dictionaries eingefügt (Zeilen 38-39) und diese als JSON gespeichert (Zeilen 42-47).

Die linguistischen Vorverarbeitung ist bewusst einfach gehalten und besteht im Wesentlichen aus folgenden Schritten:

- Alles in Kleinbuchstaben umwandeln (z.B. Verträge → verträge).
- Sonderzeichen entfernen (z.B. Zeilenumbrüche, Non-Breaking Spaces, Soft-Hyphens).
- Stoppwörter entfernen (z.B. aber, alle, ... , zur, zwar, zwischen).
- Daten verbinden (z.B. 2. april 1908 → 2april1908). Dies stellt sicher, dass Daten bei der Tokenisierung als zusammengehörende Einheit behandelt werden.
- Satzzeichen entfernen (z.B. art. → art).
- Umlaute ersetzen (z.B. verträge → vertraege).

```

1 # Gesetze und Pfad
2 laws = ['or', 'zgb']
3 law_path = '/home/oliver/Nextcloud/GymInf-Projekt/Gesetze/'
4
5 # Dictionaries bereitstellen
6 articles_formatted = {}
7 articles_preprocessed = {}
8
9 for law in laws:
10     # HTML-Datei einlesen (nur article-Tags relevant)
11     article_tags = SoupStrainer('article')
12     with open(law_path + law + '.html') as file:
13         articles = BeautifulSoup(file, 'html.parser', parse_only=article_tags)
14     # Artikel in Dictionary speichern
15     last_art_number = 0
16     for art in articles:
17         # ID und Name des Artikels bestimmen
18         art_id = art['id']
19         a_tag = art.find('a')
20         art_name = a_tag['name']
21         # Relevante Artikel bestimmen (z.B. keine Übergangsbestimmungen)
22         art_number = ''
23         for m in art_name:
24             if m.isdigit():
25                 art_number = art_number + m
26         art_number = int(art_number)
27         if art_number < last_art_number:
28             break
29         last_art_number = art_number
30         # Artikel als formatierten String auslesen
31         art_formatted = get_formatted_article(art.div)
32         # Ausgelesener String nicht leer
33         if art_formatted != '':
34             # Artikel als linguistisch vorverarbeiteten String auslesen
35             art_text = get_article_text(art.div)
36             art_preprocessed = get_preprocessed_text(art_text)
37             # Artikel in Dictionaries einfügen
38             articles_formatted[law + '_' + art_id] = art_formatted
39             articles_preprocessed[law + '_' + art_id] = art_preprocessed
40
41 # Dictionaries als JSON speichern
42 f = open('articles_formatted.json', 'w')
43 json.dump(articles_formatted, f)
44 f.close()
45 f = open('articles_preprocessed.json', 'w')
46 json.dump(articles_preprocessed, f)
47 f.close()

```

Listing 1: Einlesen und Vorverarbeitung der Gesetzesartikel

Art. 40a⁸

¹ Die nachfolgenden Bestimmungen sind auf Verträge über bewegliche Sachen und Dienstleistungen, die für den persönlichen oder familiären Gebrauch des Kunden bestimmt sind, anwendbar, wenn:

- a. der Anbieter der Güter oder Dienstleistungen im Rahmen einer beruflichen oder gewerblichen Tätigkeit gehandelt hat und
- b. die Leistung des Kunden 100 Franken übersteigt.

² Die Bestimmungen gelten nicht für Rechtsgeschäfte, die im Rahmen von bestehenden Finanzdienstleistungsverträgen gemäss Finanzdienstleistungsgesetz vom 15. Juni 2018⁹ durch Finanzinstitute und Banken abgeschlossen werden.¹⁰

^{2bis} Für Versicherungsverträge gelten die Bestimmungen des Versicherungsvertragsgesetzes vom 2. April 1908^{11,12}.

³ Bei wesentlicher Veränderung der Kaufkraft des Geldes passt der Bundesrat den in Absatz 1 Buchstabe b genannten Betrag entsprechend an.

⁸ Eingefügt durch Ziff. I des BG vom 5. Okt. 1990, in Kraft seit 1. Juli 1991 (AS 1991 846; BBl 1986 II 354).

⁹ SR 950.1

¹⁰ Fassung gemäss Ziff. II des BG vom 19. Juni 2020, in Kraft seit 1. Jan. 2022 (AS 2020 4969; BBl 2017 5089).

¹¹ SR 221.229.1

¹² Eingefügt durch Ziff. II des BG vom 19. Juni 2020, in Kraft seit 1. Jan. 2022 (AS 2020 4969; BBl 2017 5089).

(a) Text auf Website

```
<article id="art_40_a">
  <a name="a40a"></a>
  <h6 class="heading" role="heading">...</h6>
  <div class="collapseable">
    <p>...</p>
    <dl>
      <dt>a. </dt>
      <dd>...</dd>
      <dt>b. </dt>
      <dd>die Leistung des Kunden 100 Franken
        übersteigt.</dd>
    </dl>
    <p>...</p>
    <p>...</p>
    <p>...</p>
    <div class="footnotes">...</div>
  </div>
</article>
```

(b) HTML

Abbildung 1: Art. 40a OR

Anhand von Art. 40a OR wird nun kurz veranschaulicht, wie ein Gesetzesartikel in Python eingelesen und linguistisch vorverarbeitet wird.

Wie in Panel 1a ersichtlich ist, besteht Art. 40a OR aus vier Absätzen, die den vier p-Tags in Panel 1b entsprechen. Die Aufzählung (a., b.) in Panel 1a entspricht dem dl-Tag in Panel 1b. Der Code aus Listing 1 iteriert über alle Gesetzesartikel und speichert jeden Artikel als formatierten und linguistisch vorverarbeiteten String in einem Dictionary. Für Art. 40a OR liefert Listing 1 den folgenden Output.

Art. 40a OR (formatiert)

1 Die nachfolgenden Bestimmungen sind auf Verträge über bewegliche Sachen und Dienstleistungen, die für den persönlichen oder familiären Gebrauch des Kunden bestimmt sind, anwendbar, wenn:

- a. der Anbieter der Güter oder Dienstleistungen im Rahmen einer beruflichen oder gewerblichen Tätigkeit gehandelt hat und
- b. die Leistung des Kunden 100 Franken übersteigt.

2 Die Bestimmungen gelten nicht für Rechtsgeschäfte, die im Rahmen von bestehenden Finanzdienstleistungsverträgen gemäss Finanzdienstleistungsgesetz vom 15. Juni 2018 durch Finanzinstitute und Banken abgeschlossen werden.

2bis Für Versicherungsverträge gelten die Bestimmungen des Versicherungsvertragsgesetzes vom 2. April 1908.

3 Bei wesentlicher Veränderung der Kaufkraft des Geldes passt der Bundesrat den in Absatz 1 Buchstabe b genannten Betrag entsprechend an.

Art. 40a OR (linguistisch vorverarbeitet)

nachfolgenden bestimmungen verträge bewegliche sachen dienstleistungen persönlichen familien gebrauch kunden bestimmt anwendbar anbieter gueter dienstleistungen rahmen beruflichen gewerblichen tätigkeit gehandelt leistung kunden 100 franken uebersteigt bestimmungen gelten rechtsgeschäfte rahmen bestehenden finanzdienstleistungsverträgen gemäß finanzdienstleistungsgesetz 15juni2018 finanzinstitute banken abgeschlossen versicherungsverträge gelten bestimmungen versicherungsvertragsgesetzes 2april1908 wesentlicher veränderung kaufkraft geldes passt bundesrat absatz 1 buchstabe b genannten betrag entsprechend

2.2 Einlesen und Vorverarbeitung der Abfrage

Der Nutzer kann seine Abfrage in einer einfachen, auf Flask [8] basierenden Webapplikation eingeben (siehe Abbildung 2).² Dabei kann der Nutzer zwischen zwei Anzeigeeoptionen wählen:

- Top 10: Es werden die 10 passendsten Gesetzesartikel zur Abfrage angezeigt.
- Predicted Positive: Es werden alle Gesetzesartikel angezeigt, die für die Abfrage als passend klassifiziert werden.³

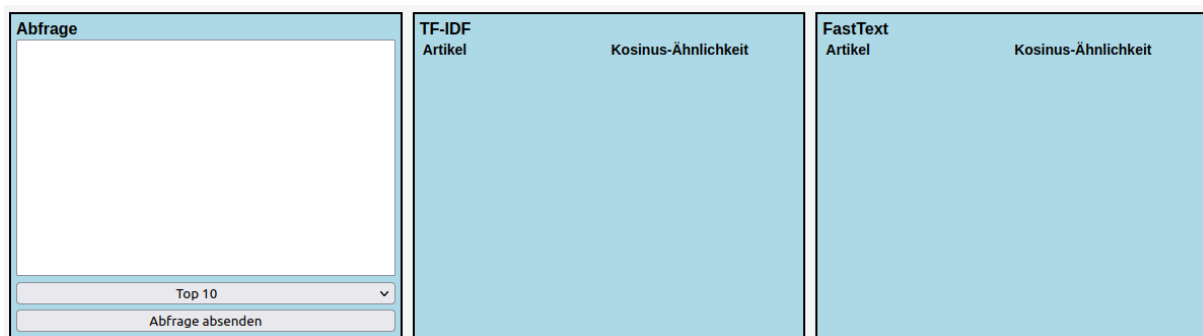


Abbildung 2: Webapplikation

Listing 2 zeigt den Python-Code zur Webapplikation. Wird die Webapplikation aufgerufen (GET request), so liefert Python `index.html` (siehe Listing 11) ohne Einträge für `query`, `display`, `tfidf` und `fasttext` zurück (Zeilen 6-7). Drückt der Nutzer den Knopf “Abfrage absenden” (POST request), so wird der eingegebene Abfragetext und die gewählte Anzeigeeoption an Python übermittelt (Zeilen 10-11). Danach werden die linguistisch vorverarbeiteten Gesetzesartikel geladen (Zeilen 12-13). Falls der Abfragetext nicht leer ist (Zeile 14), wird dieser linguistisch vorverarbeitet (Zeile 15). Anschliessend werden die Artikel in Bezug auf die Abfrage mit TF-IDF bzw. fastText nach absteigender Relevanz sortiert (Zeilen 16-17). Je nach gewählter Anzeigeeoption werden dann die 10 passendsten oder die als passend klassifizierten Gesetzesartikel bestimmt (Zeilen 18-23), bevor Python `index.html` mit den gewünschten Einträgen für `query`, `display`, `tfidf` und `fasttext` zurückliefert (Zeilen 24-28). Falls der Abfragetext leer ist, liefert Python `index.html` ohne Einträge für `query`, `display`, `tfidf` und `fasttext` zurück (Zeilen 29-30).

²Flask ist ein leichtgewichtiges Framework (Web Server Gateway Interface) für Webanwendungen in Python. Es gibt diverse Online-Tutorials zu Flask, z.B. jenes von Visual Studio Code Code [2].

³Ein Artikel wird als passend klassifiziert, wenn dessen Kosinus-Ähnlichkeit zur Abfrage über einem gewissen Schwellenwert liegt (siehe Kapitel 5).

```

1 app = Flask(__name__)
2
3 @app.route('/', methods=['GET', 'POST'])
4 def home():
5     # GET request
6     if request.method == 'GET':
7         return render_template('index.html')
8     # POST request
9     if request.method == 'POST':
10        query_text = request.form['queryText']
11        display_method = request.form['mySelect']
12        with open('articles_preprocessed.json') as json_file:
13            docs = json.load(json_file)
14        if len(query_text) > 0:
15            preprocessed_query = get_preprocessed_text(query_text)
16            tfidf_sorted = tfidf_sort_articles(docs, preprocessed_query)
17            ft_sorted = fasttext_sort_articles(docs, preprocessed_query)
18            if display_method == 'top10':
19                tfidf_art = top_n_articles(tfidf_sorted, 10)
20                ft_art = top_n_articles(ft_sorted, 10)
21            if display_method == 'predictedPositive':
22                tfidf_art = articles_over_threshold(tfidf_sorted, 0.3)
23                ft_art = articles_over_threshold(ft_sorted, 0.65)
24            return render_template('index.html',
25                                   query=query_text,
26                                   display=display_method,
27                                   tfidf=tfidf_art,
28                                   fasttext=ft_art)
29        else:
30            return render_template('index.html')
31
32 app.run(debug=True)

```

Listing 2: Python-Code zur Webapplikation

Abbildung 3 zeigt das Resultat für eine Beispielabfrage mit Anzeigeeoption Top 10. Wie die passendsten Gesetzesartikel für eine Abfrage mit TF-IDF und fastText bestimmt werden, wird in den Kapiteln 3 und 4 erläutert.

<div><div>Abfrage</div><div>Ferienanspruch Arbeitnehmer pro Jahr</div><div><div>Top 10</div><div>Abfrage absenden</div></div></div>	<div><div>TF-IDF</div><table><thead><tr><th>Artikel</th><th>Kosinus-Ähnlichkeit</th></tr></thead><tbody><tr><td>or_art_329_a</td><td>0.3352583927969111</td></tr><tr><td>or_art_329_d</td><td>0.3011674303170548</td></tr><tr><td>or_art_360_e</td><td>0.29315975002442995</td></tr><tr><td>or_art_335_d</td><td>0.2856317074734242</td></tr><tr><td>or_art_322_d</td><td>0.2852504807801393</td></tr><tr><td>or_art_329_h</td><td>0.27995317988016555</td></tr><tr><td>or_art_329_b</td><td>0.2718356595284474</td></tr><tr><td>or_art_321</td><td>0.2677390882077537</td></tr><tr><td>or_art_329_c</td><td>0.26530238297867337</td></tr><tr><td>or_art_337_c</td><td>0.2627533565883097</td></tr></tbody></table></div>	Artikel	Kosinus-Ähnlichkeit	or_art_329_a	0.3352583927969111	or_art_329_d	0.3011674303170548	or_art_360_e	0.29315975002442995	or_art_335_d	0.2856317074734242	or_art_322_d	0.2852504807801393	or_art_329_h	0.27995317988016555	or_art_329_b	0.2718356595284474	or_art_321	0.2677390882077537	or_art_329_c	0.26530238297867337	or_art_337_c	0.2627533565883097	<div><div>FastText</div><table><thead><tr><th>Artikel</th><th>Kosinus-Ähnlichkeit</th></tr></thead><tbody><tr><td>or_art_329_h</td><td>0.82308817</td></tr><tr><td>or_art_329_b</td><td>0.7859092</td></tr><tr><td>or_art_329_i</td><td>0.7598336</td></tr><tr><td>or_art_329_a</td><td>0.7550054</td></tr><tr><td>or_art_335_c</td><td>0.7534696</td></tr><tr><td>or_art_329_d</td><td>0.74196553</td></tr><tr><td>or_art_335_d</td><td>0.7366861</td></tr><tr><td>or_art_324_a</td><td>0.73498315</td></tr><tr><td>or_art_336_c</td><td>0.73396575</td></tr><tr><td>or_art_330_b</td><td>0.7336576</td></tr></tbody></table></div>	Artikel	Kosinus-Ähnlichkeit	or_art_329_h	0.82308817	or_art_329_b	0.7859092	or_art_329_i	0.7598336	or_art_329_a	0.7550054	or_art_335_c	0.7534696	or_art_329_d	0.74196553	or_art_335_d	0.7366861	or_art_324_a	0.73498315	or_art_336_c	0.73396575	or_art_330_b	0.7336576
Artikel	Kosinus-Ähnlichkeit																																													
or_art_329_a	0.3352583927969111																																													
or_art_329_d	0.3011674303170548																																													
or_art_360_e	0.29315975002442995																																													
or_art_335_d	0.2856317074734242																																													
or_art_322_d	0.2852504807801393																																													
or_art_329_h	0.27995317988016555																																													
or_art_329_b	0.2718356595284474																																													
or_art_321	0.2677390882077537																																													
or_art_329_c	0.26530238297867337																																													
or_art_337_c	0.2627533565883097																																													
Artikel	Kosinus-Ähnlichkeit																																													
or_art_329_h	0.82308817																																													
or_art_329_b	0.7859092																																													
or_art_329_i	0.7598336																																													
or_art_329_a	0.7550054																																													
or_art_335_c	0.7534696																																													
or_art_329_d	0.74196553																																													
or_art_335_d	0.7366861																																													
or_art_324_a	0.73498315																																													
or_art_336_c	0.73396575																																													
or_art_330_b	0.7336576																																													

Abbildung 3: Webapplikation mit Resultat für Beispielabfrage

3 TF-IDF

In diesem Kapitel wird mit einer TF-IDF-Schlagwortsuche bestimmt, welche Dokumente am besten zu einer Abfrage passen. Hierfür wird wie folgt vorgegangen:

- Die linguistisch vorverarbeiteten Dokumente (Gesetzesartikel) und die linguistisch vorverarbeitete Abfrage werden als TF-IDF-Vektoren dargestellt.
- Die Kosinus-Ähnlichkeit zwischen dem Abfragevektor und den Dokumentvektoren wird berechnet.
- Die Dokumente werden nach absteigender Kosinus-Ähnlichkeit zur Abfrage rangiert und die besten K Dokumente werden an den Nutzer zurückgegeben (z.B. $K = 10$ für Top 10).

3.1 TF-IDF-Vektoren

Die Grundidee bei TF-IDF (Term Frequency - Inverse Document Frequency) besteht darin, jeden Begriff der Abfrage bzw. Dokumente zu gewichten. Das TF-IDF-Gewicht hängt positiv von der Häufigkeit des Begriffs in der Abfrage bzw. im Dokument (Term Frequency) und dem Informationsgehalt des Begriffs (Inverse Document Frequency) ab (siehe Abbildung 6). Dank der TF-IDF-Gewichte können die Abfrage und Dokumente als TF-IDF-Vektoren dargestellt werden, um deren Ähnlichkeit zu bestimmen.

Die Term Frequency $tf_{t,d}$ zeigt, wie häufig ein Begriff t in einem Dokument d vorkommt. Je häufiger ein Begriff in einem Dokument vorkommt, desto relevanter ist dieser Begriff für dieses Dokument. Jedoch ist es unwahrscheinlich, dass die Relevanz eines Begriffs linear mit dessen Häufigkeit im Dokument steigt (z.B. ist ein Begriff, der in einem Dokument 10mal vorkommt, wichtiger als ein Begriff, der nur 1mal vorkommt, aber nicht 10mal wichtiger). Deshalb wird für die TF-Gewichtung häufig der Logarithmus verwendet ("sublinear tf scaling"). In Anlehnung an Manning et al. [15, S. 126] berechne ich das TF-Gewicht wie folgt:

$$w_{t,d} = \begin{cases} 1 + \ln(tf_{t,d}) & \text{falls } tf_{t,d} > 0 \\ 0 & \text{sonst} \end{cases} \quad (1)$$

Somit bekommt ein Begriff s , der in einem Dokument nur 1mal vorkommt, ein TF-Gewicht von 1, währenddessen ein Begriff t , der im gleichen Dokument 100mal vorkommt, mit 5.6 gewichtet wird (siehe Abbildung 4).

Die Document Frequency df_t sagt aus, in wie vielen Dokumenten der Begriff t in einer Sammlung von N Dokumenten vorkommt. Somit ist df_t ein inverses Mass für den Informationsgehalt eines Begriffs: Ein Begriff t , der in vielen Dokumenten vorkommt (df_t ist hoch), ist weniger charakteristisch für das jeweilige Dokument als ein Begriff u , der nur in wenigen Dokumenten vorkommt (df_u ist tief). Oder anders formuliert: Häufig vorkommende Begriffe haben einen tieferen Informationsgehalt als seltene Begriffe. Nimmt man jedoch die Inverse Document Frequency N/df_t , so erhält man ein Mass für den Informationsgehalt des Begriffs t , das ebenfalls in die Gewich-

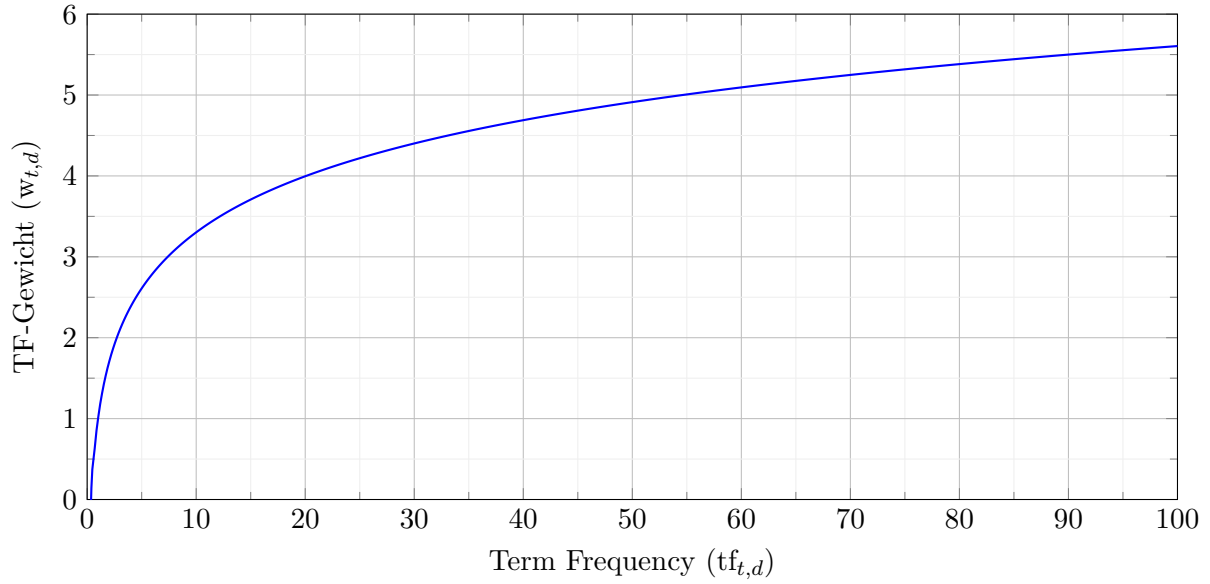


Abbildung 4: TF-Gewicht

tung von Begriff t einfließen soll: Je höher der Informationsgehalt des Begriffs, desto höher soll sein Gewicht sein. Aus ähnlichen Überlegungen wie bei der TF-Gewichtung, wird auch bei der IDF-Gewichtung häufig der Logarithmus verwendet [19]:

$$\text{idf}_t = 1 + \ln \left(\frac{1 + N}{1 + \text{df}_t} \right) \quad (2)$$

Im Gegensatz zu Manning et al. [15] wird in Gleichung (2) im Zähler und Nenner die Konstante 1 addiert, als ob ein zusätzliches Dokument vorhanden wäre, das jeden Begriff in der Sammlung einmal enthält. Dies verhindert eine Division durch 0 bei Begriffen, die in keinem Dokument vorkommen. Zusätzlich wird das IDF-Gewicht um 1 erhöht, sodass ein Begriff s , der in allen Dokumenten vorkommt, ein IDF-Gewicht von 1 anstatt 0 erhält. Ein Begriff t , der in 20 von 100 Dokumenten vorkommt, bekommt hingegen ein IDF-Gewicht von 2.6 (siehe Abbildung 5).

Kombiniert man Gleichungen (1) und (2), so kann das TF-IDF-Gewicht von Begriff t in Dokument d als Produkt des TF- und IDF-Gewichts berechnet werden:

$$\text{tfidf}_{t,d} = [1 + \ln(\text{tf}_{t,d})] \cdot \left[1 + \ln \left(\frac{1 + N}{1 + \text{df}_t} \right) \right] \quad (3)$$

Abbildung 6 zeigt das TF-IDF-Gewicht in Abhängigkeit der Term und Document Frequency für $N = 100$. Es ist deutlich zu sehen, dass das TF-IDF-Gewicht positiv von der Term Frequency und negativ von der Document Frequency abhängt.

Nachfolgend wird anhand eines einfachen Beispiels erklärt, wie TF-IDF-Vektoren berechnet werden, um eine Abfrage mit den Dokumenten zu vergleichen.

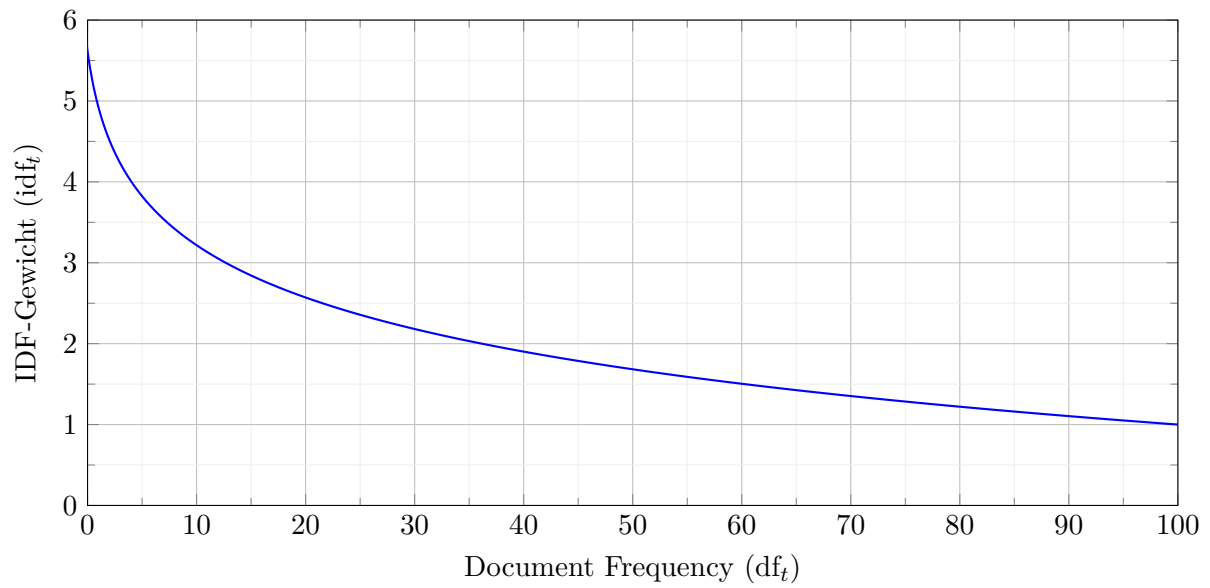


Abbildung 5: IDF-Gewicht ($N = 100$)

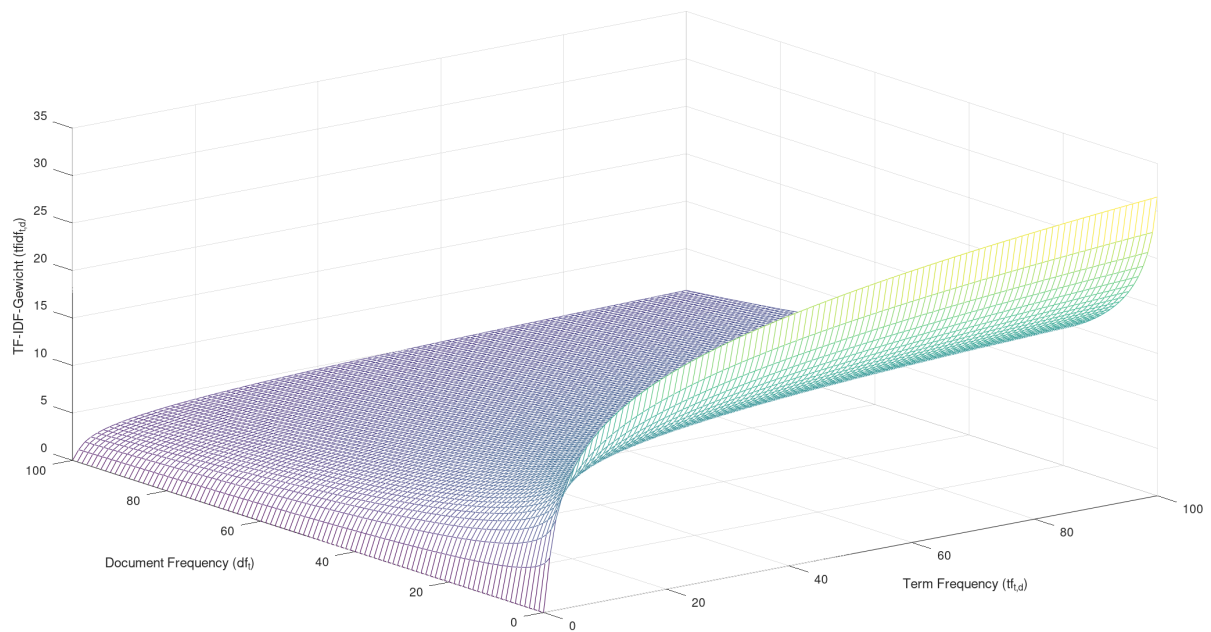


Abbildung 6: TF-IDF-Gewicht ($N = 100$)

Beispiel TF-IDF

Nehmen wir an, es gibt nur zwei Dokumente ($N = 2$) und die einzelnen Wörter stellen die Begriffe dar. Die Abfrage q und die zwei Dokumente d_1 und d_2 seien gegeben durch:

$q = \text{“what is information retrieval”}$

$d_1 = \text{“information is the new gold”}$

$d_2 = \text{“everything is information and information is everything”}$

Tabelle 1 zeigt für jeden Begriff t der beiden Dokumente die DF, das IDF-Gewicht, die TF, das TF-Gewicht und das TF-IDF-Gewicht. Man beachte, dass sich die DF und das IDF-Gewicht auf die einzelnen Begriffe beziehen (d.h. für beide Dokumente gleich sind), währenddessen sich die TF, das TF-Gewicht und das TF-IDF-Gewicht auf die einzelnen Begriffe und Dokumente beziehen (d.h. für beide Dokumente unterschiedlich sein können).

Begriff t	df_t	idf_t	tf_{t,d_1}	w_{t,d_1}	$tfidf_{t,d_1}$	tf_{t,d_2}	w_{t,d_2}	$tfidf_{t,d_2}$
and	1	1.4055	0	0	0	1	1	1.4055
everything	1	1.4055	0	0	0	2	1.6931	2.3797
gold	1	1.4055	1	1	1.4055	0	0	0
information	2	1	1	1	1	2	1.6931	1.6931
is	2	1	1	1	1	2	1.6931	1.6931
new	1	1.4055	1	1	1.4055	0	0	0
the	1	1.4055	1	1	1.4055	0	0	0

Tabelle 1: TF-IDF-Vektoren für das Beispiel (nicht normalisiert)

Die TF-IDF-Vektoren in Tabelle 1 sind noch nicht normalisiert (d.h. die Vektoren können unterschiedlich lang sein). Eine gängige Normalisierung besteht darin, die Summe der Quadrate der TF-IDF-Vektorelemente auf 1 zu setzen.

Auch für die Abfrage kann ein normalisierter TF-IDF-Vektor berechnet werden, wobei das Vokabular durch die Begriffe der Dokumente vorgegeben ist. Tabelle 2 zeigt die normalisierten TF-IDF-Vektoren für die Abfrage und die beiden Dokumente. In diesem einfachen Beispiel lassen sich also die Abfrage und Dokumente als 7-dimensionale TF-IDF-Vektoren darstellen, wobei die Dimensionalität des Vektorraums von der Anzahl unterschiedlicher Begriffe in den Dokumenten bestimmt wird.

Begriff t	$tfidf_{t,q}$	$tfidf_{t,d_1}$	$tfidf_{t,d_2}$
and	0	0	0.3844
everything	0	0	0.6508
gold	0	0.4992	0
information	0.7071	0.3552	0.4630
is	0.7071	0.3552	0.4630
new	0	0.4992	0
the	0	0.4992	0

Tabelle 2: TF-IDF-Vektoren für das Beispiel (normalisiert)

Im Beispiel TF-IDF wurden Wörter als Begriffe (Tokens) verwendet. Motiviert durch die Resultate von Hollink et al. [10], verwende ich in meinem Projekt jedoch nicht Wörter als Begriffe, sondern Character 5-Grams.⁴ Die Autoren zeigen nämlich, dass Character n-Grams für Deutsch bessere IR-Resultate liefern als bspw. Stemming oder Compound-Splitting. Gegenüber ihrem simplen Baseline-Szenario (Indexierung der Wörter) verbessert die Verwendung von Character 4- bzw. 5-Grams (innerhalb von Wortgrenzen) die Resultate um 20.3% bzw. 20.9%.⁵

Die Verwendung von Character 5-Grams führt dazu, dass der TF-IDF-Vektorraum in diesem Projekt eine Dimensionalität von 18'741 hat (d.h. es gibt 18'741 verschiedene Character 5-Grams in den 2'556 Gesetzesartikeln).

3.2 Kosinus-Ähnlichkeit

Da wir nun wissen, wie man die Abfrage und Dokumente als TF-IDF-Vektoren darstellen kann, soll nun in einem nächsten Schritt mithilfe der Kosinus-Ähnlichkeit bestimmt werden, welcher Dokumentvektor dem Abfragevektor am ähnlichsten ist.

Abbildung 7 zeigt für einen 2-dimensionalen Vektorraum, dass die Kosinus-Ähnlichkeit zwischen den zwei (normalisierten) Dokumentvektoren d_1 und d_2 als Kosinus des einschliessenden Winkels θ berechnet wird. Falls die Vektoren orthogonal sind ($\theta = 90^\circ$), so ist $\cos\text{-sim}(d_1, d_2) = 0$. Sind die Vektoren allerdings identisch ($\theta = 0^\circ$), so ist $\cos\text{-sim}(d_1, d_2) = 1$.

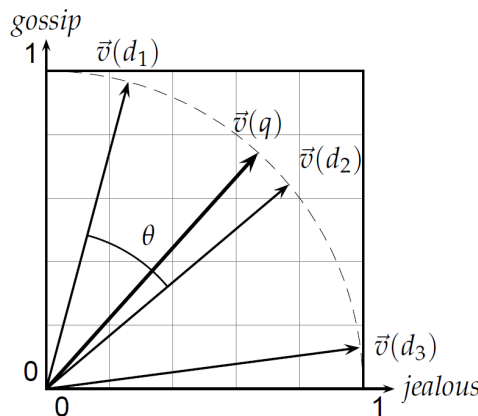


Abbildung 7: Kosinus-Ähnlichkeit zwischen zwei Vektoren: $\cos\text{-sim}(d_1, d_2) = \cos(\theta)$. Abbildung übernommen aus Manning et al. [15, S. 121]

Mathematisch wird die Kosinus-Ähnlichkeit zwischen den Vektoren a und b als Verhältnis zwischen dem Standardskalarprodukt und der euklidischen Norm berechnet [24]:

$$\cos\text{-sim}(a, b) = \cos(\theta) = \frac{a \cdot b}{\|a\| \|b\|} = \frac{\sum_{i=1}^n a_i \cdot b_i}{\sqrt{\sum_{i=1}^n (a_i)^2} \cdot \sqrt{\sum_{i=1}^n (b_i)^2}} \quad (4)$$

⁴Das Wort "vertrag" besteht aus den Character 5-Grams *vert, vertr, ertra, rtrag, trag*, wobei die Sterne den Wortanfang bzw. das Wortende markieren.

⁵Neben den guten IR-Resultaten für Deutsch besitzen Character n-Grams noch einen weiteren Vorteil: Tippfehler des Nutzers bei der Abfrageeingabe werden teilweise kompensiert. Beschränken sich die Tippfehler nämlich nur auf einzelne Wortbereiche, so sind zumindest gewisse Character 5-Grams identisch.

Beispiel TF-IDF (Fortsetzung)

In Tabelle 2 wurden die normalisierten TF-IDF-Vektoren für das Beispiel TF-IDF berechnet. Nun soll kurz gezeigt werden, wie für dieses einfache Beispiel die Kosinus-Ähnlichkeit zwischen dem Abfragevektor und den beiden Dokumentvektoren berechnet werden kann.

Zur Erinnerung werden die Abfrage q und die zwei Dokumente d_1 und d_2 für das Beispiel TF-IDF nochmals wiedergegeben:

q = “what is information retrieval”

d_1 = “information is the new gold”

d_2 = “everything is information and information is everything”

Gemäss Gleichung (4) kann die Kosinus-Ähnlichkeit zwischen zwei Vektoren als Verhältnis zwischen dem Standardskalarprodukt und der euklidischen Norm berechnet werden. Da die TF-IDF-Vektoren bereits normalisiert sind, reduziert sich die Kosinus-Ähnlichkeit zwischen zwei TF-IDF-Vektoren auf deren Standardskalarprodukt. Tabelle 3 zeigt die Kosinus-Ähnlichkeiten zwischen dem Abfragevektor und den beiden Dokumentvektoren.

$\text{cos-sim}(q, d_1)$	$\text{cos-sim}(q, d_2)$
0.5023	0.6548

Tabelle 3: Kosinus-Ähnlichkeiten für das Beispiel TF-IDF

Die Abfrage q ist demnach dem Dokument d_2 ähnlicher als dem Dokument d_1 . Dies lässt sich dadurch erklären, dass die Abfragebegriffe *is* und *information* im Dokument d_2 je zweimal, im Dokument d_1 aber nur je einmal vorkommen. Das führt dazu, dass die TF-Gewichte – und daher auch die TF-IDF-Gewichte – für diese beiden Begriffe im Dokument d_2 höher sind als im Dokument d_1 (siehe Tabelle 2), was wiederum zu einem grösserem Standardskalarprodukt (Kosinus-Ähnlichkeit) zwischen der Abfrage q und dem Dokument d_2 führt.

3.3 Umsetzung in Python

Die Umsetzung von TF-IDF in Python ist dank `TfidfVectorizer` [20] einfach (siehe Listing 3). Übergibt man der Funktion `tfidf_sort_articles()` die Gesetzesartikel (Dokumente) und Abfrage, so werden zuerst die TF-IDF-Dokumentvektoren gebildet (Zeilen 4-7). Dabei werden Character 5-Grams als Tokenisierungsmethode und die Logarithmusvariante für die TF-Gewichtung gewählt (“sublinear tf scaling”). Zudem wird das Dokumentvokabular (d.h. die verschiedenen Character 5-Grams, die in den Gesetzesartikeln vorkommen) für die Erstellung des TF-IDF-Abfragevektors gespeichert (Zeile 8). Danach wird der TF-IDF-Abfragevektor unter Verwendung des Dokumentvokabulars gebildet (Zeilen 10-15) und die Funktion `doc_cossim_url_tuples()` aufgerufen (Zeile 17). Diese Funktion (siehe Listing 4) liefert Tupels (`doc`, `cossim`, `url`) zurück, die nach absteigender Kosinus-Ähnlichkeit (`cossim`) zwischen dem TF-IDF-Abfragevektor und den TF-IDF-Dokumentvektoren sortiert sind.


```

1 def tfidf_sort_articles(docs, query):
2     index_docs = [n for n in docs]
3     # TF-IDF-Vektoren Dokumente
4     vectorizer = TfidfVectorizer(analyzer='char_wb',
5                                 ngram_range=(5,5),
6                                 sublinear_tf=True)
7     tfidf_docs = vectorizer.fit_transform(docs.values())
8     vocab = vectorizer.vocabulary_
9     # TF-IDF-Vektor Abfrage
10    dict_query = {'query': query}
11    vectorizer = TfidfVectorizer(analyzer='char_wb',
12                                vocabulary=vocab,
13                                ngram_range=(5,5),
14                                sublinear_tf=True)
15    tfidf_query = vectorizer.fit_transform(dict_query.values())
16    # Kosinus-Ähnlichkeit zwischen Abfrage und Dokumenten
17    return doc_cossim_url_tuples(index_docs, tfidf_docs, tfidf_query)

```

Listing 3: Artikelsortierung nach absteigender Relevanz (TF-IDF)

```

1 def doc_cossim_url_tuples(docs_index, docs_vect, query_vect):
2     cos_sim = cosine_similarity(query_vect, docs_vect)
3     tuples_list = []
4     i = 0
5     for doc in docs_index:
6         art_id = doc
7         if art_id.startswith('or'):
8             art = art_id[3:]
9             url = 'https://www.fedlex.admin.ch/eli/cc/27/317_321_377/de#' + art
10        if art_id.startswith('zgb'):
11            art = art_id[4:]
12            url = 'https://www.fedlex.admin.ch/eli/cc/24/233_245_233/de#' + art
13        tuple = (art_id, cos_sim[0][i], url)
14        tuples_list.append(tuple)
15        i += 1
16    return sorted(tuples_list, key=lambda x:x[1], reverse=True)

```

Listing 4: Hilfsfunktion für Tupel-Generierung

Nachdem die Funktion `tfidf_sort_articles()` die Gesetzesartikel für eine Abfrage nach absteigender Relevanz (Kosinus-Ähnlichkeit) sortiert hat, können die passendsten 10 Artikel mit der Funktion `top_n_articles()` und die als passenden klassifizierten Artikel mit der Funktion `articles_over_threshold()` (siehe Listing 13) bestimmt werden. Tabelle 4 zeigt beispielhaft die 10 passendsten Tupel für die Beispielabfrage (Ferienanspruch Arbeitnehmer pro Jahr).

doc	cossim	url
or_art_329_a	0.3353	https://www.fedlex.admin.ch/eli/cc/27/317_321_377/de#art_329_a
or_art_329_d	0.3012	https://www.fedlex.admin.ch/eli/cc/27/317_321_377/de#art_329_d
or_art_360_e	0.2932	https://www.fedlex.admin.ch/eli/cc/27/317_321_377/de#art_360_e
or_art_335_d	0.2856	https://www.fedlex.admin.ch/eli/cc/27/317_321_377/de#art_335_d
or_art_322_d	0.2853	https://www.fedlex.admin.ch/eli/cc/27/317_321_377/de#art_322_d
or_art_329_h	0.2800	https://www.fedlex.admin.ch/eli/cc/27/317_321_377/de#art_329_h
or_art_329_b	0.2718	https://www.fedlex.admin.ch/eli/cc/27/317_321_377/de#art_329_b
or_art_321	0.2677	https://www.fedlex.admin.ch/eli/cc/27/317_321_377/de#art_321
or_art_329_c	0.2653	https://www.fedlex.admin.ch/eli/cc/27/317_321_377/de#art_329_c
or_art_337_c	0.2628	https://www.fedlex.admin.ch/eli/cc/27/317_321_377/de#art_337_c

Tabelle 4: Top-10-Tupel für die Beispielabfrage (TF-IDF)

In diesem Kapitel wurde gezeigt, wie mit TF-IDF die passendsten Gesetzesartikel zu einer Abfrage bestimmt werden. TF-IDF ist eine populäre und sehr einfache Technik für IR-Aufgaben, die aber auch gewisse Limitierungen aufweist. So wird bei TF-IDF bspw. die semantische Ähnlichkeit von Wörtern nicht berücksichtigt (z.B. ist die Kosinus-Ähnlichkeit zwischen den TF-IDF-Wortvektoren Mann und Frau null, obwohl diese Wörter semantisch sehr ähnlich sind). Im nächsten Kapitel wird erklärt, wie mit fastText die passendsten Gesetzesartikel zu einer Abfrage bestimmt werden. Im Gegensatz zu TF-IDF ist fastText ein neuronales Netz (shallow neural network), das bspw. die semantische Ähnlichkeit von Wörtern berücksichtigt.

4 FastText

FastText wurde von Facebook AI Research entwickelt, um die Klassifizierung und Darstellung von Text effizient umzusetzen [7]:

FastText is an open-source, free, lightweight library that allows users to learn text representations and text classifiers. It works on standard, generic hardware. Models can later be reduced in size to even fit on mobile devices.

In diesem Kapitel werden zuerst die zwei Anwendungsbereiche von fastText (Textklassifizierung und Textdarstellung) vorgestellt. Anschliessend wird gezeigt, wie fastText in Python implementiert werden kann.

4.1 Textklassifizierung mit fastText

Textklassifizierung ist das Kernproblem bei vielen Anwendungen (z.B. Spam-Erkennung, Stimmungsanalyse, intelligente Antworten). Das Ziel von Textklassifizierung besteht darin, Dokumente (z.B. Emails) einer oder mehreren Kategorien (Labels) zuzuordnen (z.B. Spam vs. Nicht-Spam). Heutzutage werden solche Textklassifizierer meist mithilfe von Machine Learning erstellt, d.h. die Klassifizierungsregeln werden aus Beispielen gelernt. Hierfür werden Trainingsdaten benötigt, die aus Dokumenten und ihren entsprechenden Kategorien bestehen. Abbildung 8 zeigt zwei Textklassifizierungsbeispiele.⁶

Starshmit (born Finlay Dow-Smith 8 July 1988 Bromely England) is a British songwriter producer remixer and DJ. He studied classical music degree at the University of Surrey majoring in performance on saxophone. He has already received acclaim for the remixes he has created for Lady Gaga Robyn Timbaland Katy Perry Little Boots Passion Pit Paloma Faith Marina and the Diamonds and Frankmusik amongst many others.	Rikkavesi is a medium-sized lake in eastern Finland. At approximately 63 square kilometers (24 sq mi) it is the 66th largest lake in Finland. Rikkavesi is situated in the municipalities of Kaavi Outokumpu and Tuusniemi. Rikkavesi is 101 metres (331 ft) above the sea level. Kaavinjärvi and Rikkavesi are connected by the Kaavinkoski Canal. Ohtaansalmi strait flows from Rikkavesi to Juojärvi.
Label: artist	Label: natural place

Abbildung 8: Zwei Textklassifizierungsbeispiele

In fastText funktioniert die Textklassifizierung ähnlich wie im CBOW-Modell⁷ [16], bei dem die Wahrscheinlichkeit eines Wortes für einen gegebenen Kontext prognostiziert wird. Nehmen wir folgenden Beispielsatz, um die Funktionsweise des CBOW-Modells zu erklären:

The mighty knight Lancelot fought bravely.

Beim CBOW-Modell geht es darum, die Wahrscheinlichkeit für das Zielwort w (knight) gegeben den Kontext C (The, mighty, Lancelot, fought, bravely) vorherzusagen. In anderen Worten, das CBOW-Modell modelliert die Wahrscheinlichkeit $p(w|C)$. Die Modellarchitektur des CBOW-Modells ist in Abbildung 9 dargestellt.

⁶Die englischsprachigen Textbeispiele in diesem Abschnitt stammen aus einem Vortrag von Piotr Bojanowski, der unter <https://www.youtube.com/watch?v=CHcExDsDeHU> zu finden ist.

⁷CBOW steht für Continuous Bag Of Words.

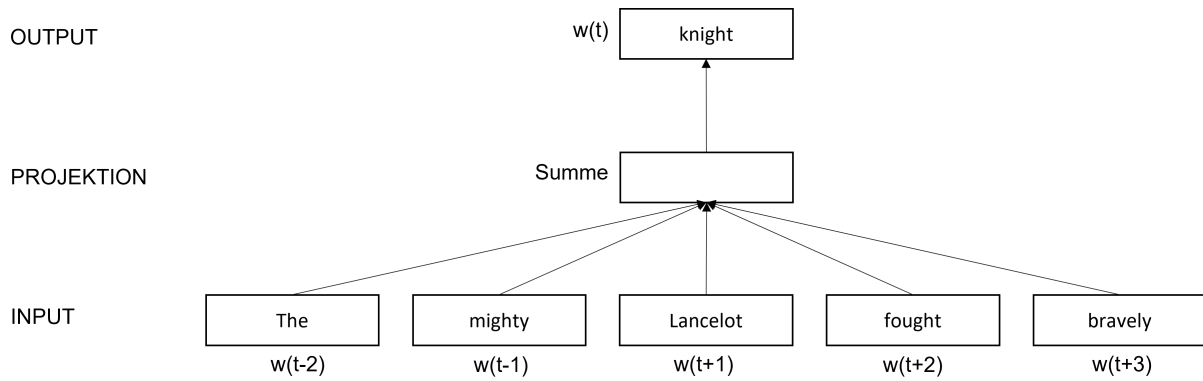


Abbildung 9: CBOW-Modellarchitektur. Eigene Darstellung in Anlehnung an Mikolov et al. [16].

Im Unterschied zum CBOW-Modell, modelliert fastText die Wahrscheinlichkeit eines Labels für einen gegebenen Textabschnitt. In anderen Worten, fastText klassifiziert einen gegebenen Text, indem dieser einem (oder mehreren) Label zugewiesen wird (siehe Abbildung 8).

Mit fastText kann man sehr einfach und effizient Texte klassifizieren. Dank gewissen Eigenschaften (z.B. Klassifizierungstext als “bag of words” darstellen, Beschneiden des Vokabulars, Hashing von Wörtern) ist fastText ein Textklassifizierungsmodell von kleiner Grösse, das vergleichbar gute Resultate wie moderne Deep-Learning-Modelle liefert, aber deutlich schneller zu trainieren und evaluieren ist [12] [13].

Grundsätzlich wäre ein Textklassifizierungsansatz für dieses Projekt möglich gewesen. Hierfür hätte ich selber ein Textklassifizierungsmodell trainieren müssen, um damit Nutzerabfragen einem oder mehreren Gesetzesartikeln (Labels) zuzuordnen. Das Trainieren eines Textklassifizierungsmodells benötigt allerdings viele Trainingsdaten, d.h. Dokumente mit entsprechenden Labels. Da solche Trainingsdaten für das OR und ZGB nicht verfügbar sind und das Erstellen bzw. Sammeln solcher Trainingsdaten extrem zeitaufwändig ist, habe ich mich gegen einen Textklassifizierungs- und für einen Textdarstellungsansatz entschieden.

4.2 Textdarstellung mit fastText

Beim modernen Machine Learning werden Wörter häufig als Vektoren (Word Embeddings) dargestellt. Diese Vektoren erfassen verborgene Informationen über eine Sprache, wie Wortanalogien oder Semantik. Wortvektoren werden auch verwendet, um die Leistung von Textklassifikatoren zu verbessern.

Im Gegensatz zu word2vec [25] basiert fastText auf der Idee, Wortvektoren mit Teilwortinformationen anzureichern. Jedes Wort w wird daher als Bag of Character n-Grams dargestellt. So ist bspw. die Menge der n-Grams für das Wort Vertrag und $n = 3$ gegeben durch:

$$\langle \text{Ve, ver, ert, rtr, tra, rag, ag} \rangle, \langle \text{Vertrag} \rangle$$

Dabei sind \langle und \rangle Begrenzungszeichen, die den Wortanfang bzw. das Wortende kennzeichnen, und $\langle \text{Vertrag} \rangle$ soll es erlauben, auch eine Darstellung für das gesamte Wort zu lernen. Ein Wort wird schlussendlich durch die Summe der Vektordarstellungen seiner n-Grams dargestellt.

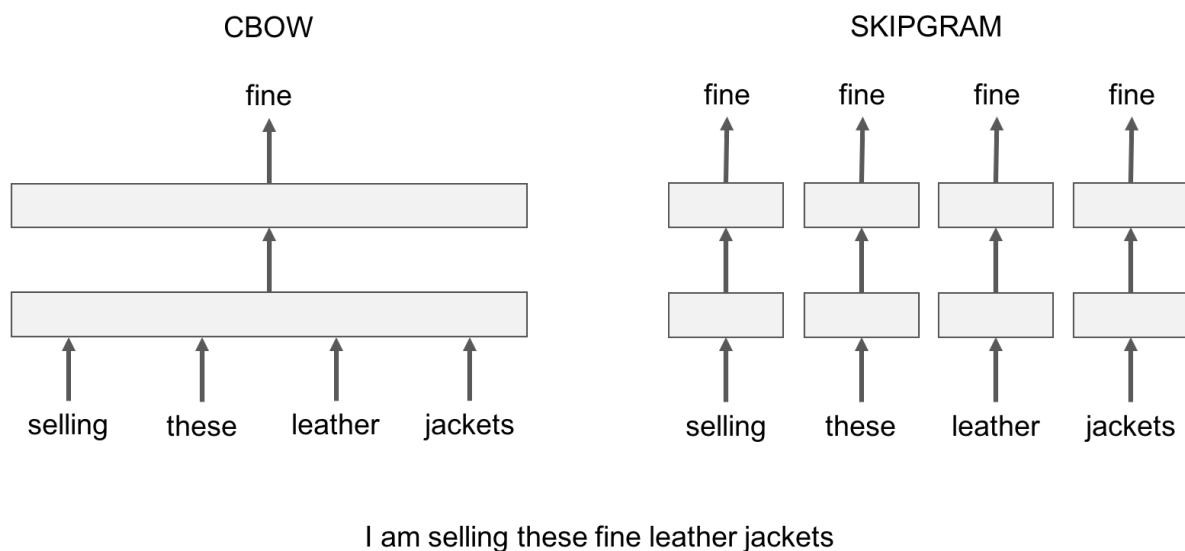


Abbildung 10: CBOW vs. SKIPGRAM. Abbildung übernommen aus Facebook [6]

Dies ermöglicht, Darstellungen zwischen Wörtern zu teilen und somit zuverlässig Darstellungen für seltene Worte zu lernen [1].

FastText bietet zwei Modelle zur Berechnung von Wortdarstellungen: SKIPGRAM und CBOW (siehe Abbildung 10). Das SKIPGRAM-Modell lernt, ein Zielwort anhand eines nahe geliegenen Wortes vorherzusagen, währenddessen das CBOW-Modell das Zielwort anhand seines Kontext voraussagt. Das SKIPGRAM-Modell versucht also, das Zielwort (fine) mit einem zufälligen Wort in der Nähe (z.B. leather, jackets) vorherzusagen. Das CBOW-Modell nutzt alle Kontextwörter (selling, these, leather, jackets) und nutzt die Summe deren Vektoren, um das Zielwort (fine) vorherzusagen. In der Praxis zeigt sich, dass SKIPGRAM-Modelle besser mit Teilwortinformationen arbeiten als CBOW-Modelle [6].

Dank der Einfachheit von fastText können Modelle sehr schnell trainiert werden. Zudem liefert fastText bessere Resultate als Methoden, die keine Teilwortinformationen berücksichtigen, und Ansätze, die auf einer morphologischer Analyse⁸ beruhen. Besonders interessant für dieses Projekt ist, dass fastText – wegen der Verwendung von Teilwortinformationen – ziemlich gut für morphologisch reiche Sprachen wie Deutsch zu funktionieren scheint [1].

Auf der Website von fastText stehen vortrainierte Wortvektoren für 157 Sprachen zur Verfügung, u.a. auch für Deutsch [5]. Diese Wortvektoren wurden auf sehr grossen Datensätzen von Wikipedia (Deutsch: 1'384'170'636 Tokens, 3'005'294 Wörter) und dem Common Crawl Projekt (Deutsch: 65'648'657'780 Tokens, 19'767'924 Wörter) trainiert, weshalb sie von hoher Qualität sind [9]. Dies ist auch der Grund, weshalb ich nicht selbst Wortvektoren für dieses Projekt trainiert, sondern die vortrainierten Wortvektoren von fastText verwendet habe.

⁸Die Morphologie befasst sich mit der Struktur und dem Aufbau von Wörtern. Dies erlaubt bspw. die zwei Wörter *Autofahrer* und *Autodach* in Zusammenhang zu bringen, da beide Wörter das Präfix *Auto* enthalten.

4.3 Umsetzung in Python

Mit einem vortrainierten fastText-Modell können nicht nur Wortvektoren, sondern auch Nearest Neighbors und Wortanalogien bestimmt werden (siehe Listing 5).

```
1 import fasttext
2
3 # Vortrainiertes Modell laden
4 ft = fasttext.load_model('fastText/cc.de.300.bin')
5 ft.get_dimension()
6
7 # Wortvektoren und Nearest Neighbors
8 words = ['Hallo', 'Banane']
9 for word in words:
10     ft.get_word_vector(word)
11     ft.get_nearest_neighbors(word)
12
13 # Wortanalogien
14 ft.get_analogies('Berlin', 'Deutschland', 'Frankreich')
```

Listing 5: Wortvektoren, Nearest Neighbors und Wortanalogien mit fastText

In Listing 5 wird zuerst das vortrainierte fastText-Modell geladen und dessen Dimensionalität bestimmt. Die Dimensionalität von vortrainierten fastText-Modellen ist standardmässig 300 (d.h. alle Wortvektoren haben 300 Einträge), diese kann aber bei Bedarf reduziert werden, um das Modell bspw. auf Smartphones zu verwenden. Anschliessend werden die Wortvektoren und die Nearest Neighbors für die Wörter Hallo und Banane bestimmt (siehe Tabelle 5, Spalten 1 und 2). Ein weiteres Feature von fastText sind Wortanalogien. Für die Wortanalogie

Berlin → Deutschland
? → Frankreich

gibt fastText Paris als beste Antwort zurück (siehe Tabelle 5, Spalte 3).

Hallo	Banane	Berlin → Deutschland ? → Frankreich
0.8294 Hallöchen	0.7592 Bananen	0.7424 Paris
0.8284 hallo	0.6774 Ananas	0.6406 Marseille
0.8149 Moin	0.6750 banane	0.6256 Montpellier
0.7683 Huhu	0.6598 Gurke	0.6073 Toulouse
0.7607 Hallihallo	0.6522 Bananenmilch	0.5821 Lyon
0.7484 Hey	0.6265 Zitrone	0.5769 Charlottenburg
0.7365 .Hallo	0.6258 Erdbeere	0.5727 Südfrankreich
0.7288 Hallöle	0.6190 Südfrucht	0.5683 Amiens
0.7184 Hallo.	0.6173 Bananenschale	0.5682 Nantes
0.7132 hallöchen	0.6168 Möhre	0.5676 Nizza

Tabelle 5: Nearest Neighbors und Wortanalogien mit fastText

Mit fastText kann man jedoch nicht nur Wortvektoren, sondern auch Vektoren für ganze Sätze bzw. Textabschnitte bilden. Für das Beispiel TF-IDF kommt auch fastText zum Schluss, dass Dokument d_2 der Abfrage q ähnlicher ist als Dokument d_1 (siehe Listing 6 und Tabelle 6).

```

1 # Beispiel TF-IDF mit fastText
2 docs = {
3     'd1': 'information is the new gold',
4     'd2': 'everything is information and information is everything'
5 }
6
7 query = 'what is information retrieval'
8
9 # fastText-Vektoren Dokumente
10 fasttext_docs = []
11 for doc in docs.values():
12     vect_doc = ft.get_sentence_vector(doc)
13     fasttext_docs.append(vect_doc)
14
15 # fastText-Vektor Abfrage
16 fasttext_query = ft.get_sentence_vector(query)
17 fasttext_query = fasttext_query.reshape(1, -1)
18
19 # Kosinus-Ähnlichkeit zwischen Abfrage und Dokumenten
20 cossim = cosine_similarity(fasttext_query, fasttext_docs)

```

Listing 6: Beispiel TF-IDF mit fastText

cos-sim(q, d_1)	cos-sim(q, d_2)
0.7850	0.8671

Tabelle 6: Kosinus-Ähnlichkeiten für das Beispiel TF-IDF mit fastText

Die Funktion `fasttext_sort_articles()` (siehe Listing 7) sortiert die Gesetzesartikel für eine gegebene Abfrage nach absteigender Relevanz (Kosinus-Ähnlichkeit). Anschliessend können mit den Funktionen `top_n_articles()` und `articles_over_threshold()` die 10 passendsten Artikel und die als passenden klassifizierten Artikel bestimmt werden (siehe Listing 13). Tabelle 7 zeigt beispielhaft die Top-10-Tupel für die Beispielabfrage (Ferienanspruch Arbeitnehmer pro Jahr).

```

1 def fasttext_sort_articles(docs, query):
2     index_docs = [n for n in docs]
3     ft = fasttext.load_model('fastText/cc.de.300.bin')
4     # fastText-Vektoren Dokumente
5     fasttext_docs = []
6     for doc in docs.values():
7         vect_doc = ft.get_sentence_vector(doc)
8         fasttext_docs.append(vect_doc)
9     # fastText-Vektoren Abfrage
10    fasttext_query = ft.get_sentence_vector(query)
11    fasttext_query = fasttext_query.reshape(1, -1)
12    # Kosinus-Ähnlichkeit zwischen Abfrage und Dokumenten
13    return doc_cossim_url_tuples(index_docs, fasttext_docs, fasttext_query)

```

Listing 7: Artikelsortierung nach absteigender Relevanz (fastText)

doc	cossim	url
or_art_329_h	0.8231	https://www.fedlex.admin.ch/eli/cc/27/317_321_377/de#art_329_h
or_art_329_b	0.7859	https://www.fedlex.admin.ch/eli/cc/27/317_321_377/de#art_329_b
or_art_329_i	0.7598	https://www.fedlex.admin.ch/eli/cc/27/317_321_377/de#art_329_i
or_art_329_a	0.7550	https://www.fedlex.admin.ch/eli/cc/27/317_321_377/de#art_329_a
or_art_335_c	0.7535	https://www.fedlex.admin.ch/eli/cc/27/317_321_377/de#art_335_c
or_art_329_d	0.7420	https://www.fedlex.admin.ch/eli/cc/27/317_321_377/de#art_329_d
or_art_335_d	0.7367	https://www.fedlex.admin.ch/eli/cc/27/317_321_377/de#art_335_d
or_art_324_a	0.7350	https://www.fedlex.admin.ch/eli/cc/27/317_321_377/de#art_324_a
or_art_336_c	0.7340	https://www.fedlex.admin.ch/eli/cc/27/317_321_377/de#art_336_c
or_art_330_b	0.7337	https://www.fedlex.admin.ch/eli/cc/27/317_321_377/de#art_333_b

Tabelle 7: Top-10-Tupel für die Beispielabfrage (fastText)

5 Evaluation

Leider gibt es keine Benchmark-Datensätze für das OR und ZGB. Deshalb habe ich mir typische Nutzerabfragen überlegt und die dazu passenden Gesetzesartikel definiert (siehe Tabelle 8), um zu evaluieren, ob TF-IDF oder fastText bessere Resultate liefert.

Evaluationsabfragen	Passende Gesetzesartikel
Widerrufsrecht bei Haustürgeschäften	OR 40a bis OR 40f
Erfüllungsort bei Geldschulden	OR 74
Transportkosten beim Kaufvertrag	OR 189
Wohnung untervermieten	OR 262
Fristlose Kündigung des Arbeitsvertrags	OR 337 bis OR 337c
Konkurrenzverbot im Arbeitsvertrag	OR 340 bis OR 340c
Mindestkapital AG	OR 621
Auflösung einer Verlobung	ZGB 91 bis ZGB 93
Scheidung einer Ehe	ZGB 111 bis ZGB 115
Altersunterschied bei Adoption	ZGB 264d
Enterbung eines Kindes	ZGB 477 bis ZGB 480

Tabelle 8: Evaluationsabfragen und passende Gesetzesartikel

Für die Evaluation werden Kenngrössen verwendet (z.B. Recall, Precision), die alle auf True Positive, False Positive, True Negative und False Negative basieren (siehe Abbildung 11):

- True Positive: Ein Artikel ist für eine Abfrage passend (Actual Positive) und wird auch als passend klassifiziert (Predicted Positive).
- False Positive: Ein Artikel ist für eine Abfrage unpassend (Actual Negative), wird aber als passend klassifiziert (Predicted Positive).
- True Negative: Ein Artikel ist für eine Abfrage unpassend (Actual Negative) und wird auch als unpassend klassifiziert (Predicted Negative).
- False Negative: Ein Artikel ist für eine Abfrage passend (Actual Positive), wird aber als unpassend klassifiziert (Predicted Negative).

	Predicted Positive	Predicted Negative
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

Abbildung 11: True Positive, False Positive, True Negative, False Negative

Die Accuracy zeigt den Anteil der korrekt klassifizierten Artikel unter allen Artikeln [22]:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}} \quad (5)$$

Der Recall zeigt den Anteil der korrekt klassifizierten Artikel unter den als passenden Artikeln:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad (6)$$

Die Precision zeigt den Anteil der korrekt klassifizierten Artikel unter den passend klassifizierten Artikeln:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \quad (7)$$

Natürlich wäre ein hoher Recall und eine hohe Precision wünschenswert. In der Praxis gibt es allerdings einen Recall-Precision-Tradeoff: Der Recall lässt sich auf Kosten einer tieferen Precision erhöhen, und umgekehrt [14]. Daher kombiniert F1 Score den Recall und die Precision in eine einzige Kenngrösse:

$$\text{F1 Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \quad (8)$$

Für die Klassifizierung, ob ein Artikel für eine Abfrage passend oder unpassend ist, braucht es einen Schwellenwert: Liegt die Kosinus-Ähnlichkeit zwischen der Abfrage und dem Artikel über diesem Schwellenwert, so wird der Artikel als passend klassifiziert (Predicted Positive). Andernfalls wird der Artikel als unpassend klassifiziert (Predicted Negative). Für TF-IDF wurde ein Schwellenwert von 0.3, für fastText von 0.65 gewählt (siehe Abbildung 12).

Die durchschnittlichen Kenngrössen für die Evaluationsabfragen (siehe Tabelle 8) werden in Listing 14 für verschiedene Schwellenwerte berechnet und in Abbildung 12 grafisch dargestellt. Es fällt auf, dass – zumindest im limitierten Evaluationssetting dieses Projekts – TF-IDF deutlich bessere Resultate als fastText liefert: Der durchschnittliche F1 Score für TF-IDF beträgt maximal 0.6181 (Schwellenwert 0.3), währenddessen der durchschnittliche F1 Score für fastText nur 0.1225 erreicht (Schwellenwert 0.65). Nimmt man also den F1 Score als Vergleichsmass, so sind die TF-IDF-Evaluationsresultate ca. 6mal besser als jene von fastText.

Da die Evaluationsresultate für fastText ernüchternd sind, habe ich zusätzlich getestet, ob diese mithilfe einer linearen Dimensionalitätsreduktion (Truncated SVD) verbessert werden können. Im Gegensatz zur Principal Component Analysis (PCA) zentriert die Truncated SVD die Daten nicht, bevor die Singulärwertzerlegung berechnet wird, wodurch dünnbesetzte Matrizen (z.B. TF-IDF-Matrix) effizient verarbeitet werden können [21]. Listing 15 zeigt, welche Code-Anpassungen für Truncated SVD nötig waren. Leider ist jedoch aus Abbildung 13 ersichtlich, dass Truncated SVD die Evaluationsresultate nicht verbessert.

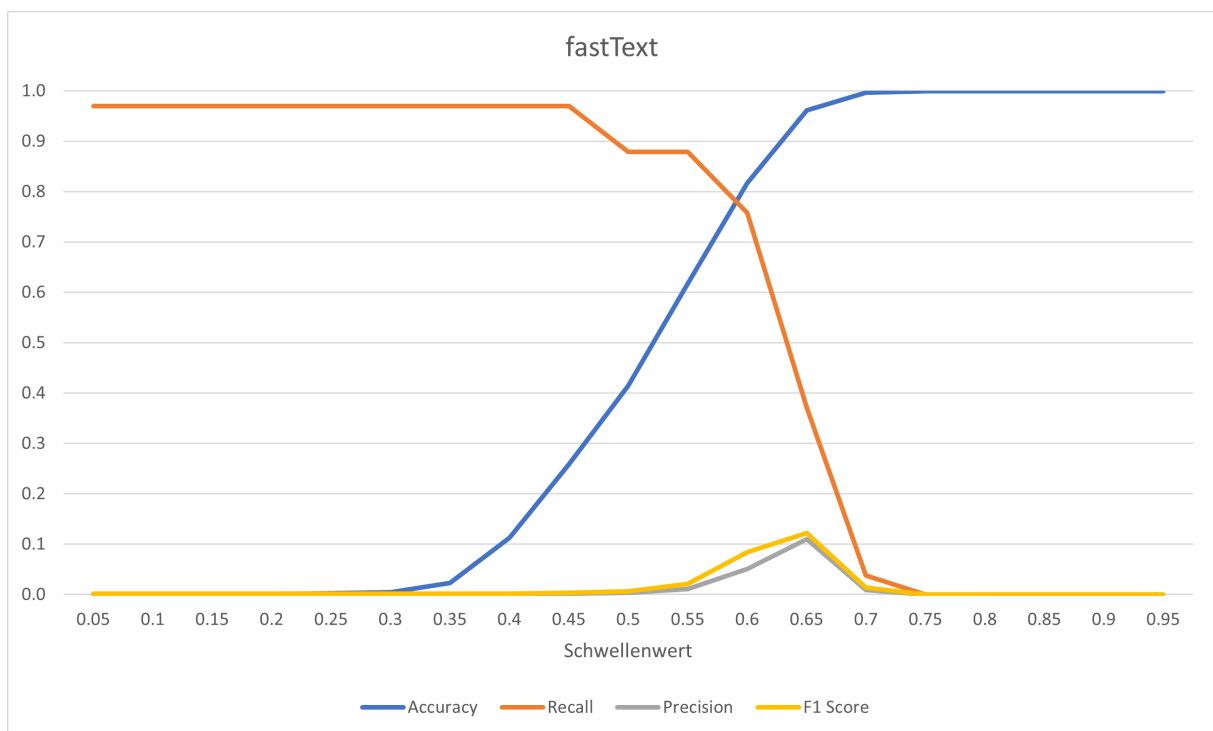


Abbildung 12: Evaluation

6 Schlussfolgerungen

In diesem Bericht wurden zwei Ansätze vorgestellt (TF-IDF und fastText), um Information Retrieval bei Gesetzestexten zu betreiben. Konkret ging es darum, für eine juristische Nutzerabfrage die passenden Gesetzesartikel im OR bzw. ZGB zu finden, ein typisches Problem im gymnasialen Rechtunterricht.

Für die Eingabe der Nutzerabfrage wurde eine einfache, Flask-basierte Webapplikation entwickelt. Auf Knopfdruck werden die Abfrage und Dokumente als TF-IDF- bzw. fastText-Vektoren dargestellt, um – basierend auf der Kosinus-Ähnlichkeit – die 10 passendsten bzw. die als passend klassifizierten Gesetzesartikel zu bestimmen und anzuzeigen.

Währenddessen TF-IDF akzeptable Evaluationsresultate liefert (maximaler F1 Score von 0.6), sind die fastText-Resultate bescheiden (maximaler F1 Score von 0.1). Somit kann die Schlussfolgerung gezogen werden, dass – zumindest im limitierten Evaluationssetting dieses Projekts – TF-IDF ca. 6mal bessere Resultate als fastText liefert.

Der Grund für die bescheidenen fastText-Resultate dürfte darin liegen, dass fastText primär für Wortvektoren entwickelt wurde und daher die einzelnen Begriffe (Character n-Grams) nicht gewichtet. Somit erhalten bspw. Begriffe, die in vielen Dokumenten vorkommen und daher einen tiefen Informationsgehalt haben, das gleiche Gewicht wie seltene Begriffe mit hohem Informationsgehalt. Aufgrund der Länge der Gesetzesartikel (ein typischer Gesetzesartikel ist mehrere Sätze lang), werden die fastText-Dokumentvektoren daher durch irrelevant Begriffe (Noise) dominiert, was es schwierig macht, die passenden Artikel zu einer Abfrage zu finden. TF-IDF leidet dank der Gewichtung der Begriffe nicht unter diesem Problem, was die besseren Evaluationsresultate von TF-IDF erklärt.

Literatur

- [1] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomáš Mikolov. Enriching word vectors with subword information. 2017.
- [2] Visual Studio Code. Flask tutorial in Visual Studio Code, 2022. URL <https://code.visualstudio.com/docs/python/tutorial-flask>.
- [3] Schweizerische Eidgenossenschaft. Bundesgesetz betreffend die Ergänzung des Schweizerischen Zivilgesetzbuches (Fünfter Teil: Obligationenrecht), 2022. URL https://www.fedlex.admin.ch/eli/cc/27/317_321_377/de?print=true.
- [4] Schweizerische Eidgenossenschaft. Schweizerisches Zivilgesetzbuch, 2022. URL https://www.fedlex.admin.ch/eli/cc/24/233_245_233/de?print=true.
- [5] Facebook. Word vectors for 157 languages, 2022. URL <https://fasttext.cc/docs/en/crawl-vectors.html>.
- [6] Facebook. CBOW vs. SKIPGRAM, 2022. URL <https://fasttext.cc/docs/en/unsupervised-tutorial.html>.
- [7] Facebook. fastText - library for efficient text classification and representation learning, 2022. URL <https://fasttext.cc>.
- [8] Flask. Flask documentation, 2022. URL <https://flask.palletsprojects.com/en/2.0.x/>.
- [9] Edouard Grave, Piotr Bojanowski, Prakhar Gupta, Armand Joulin, and Tomáš Mikolov. Learning word vectors for 157 languages. *CoRR*, abs/1802.06893, 2018. URL <http://arxiv.org/abs/1802.06893>.
- [10] Vera Hollink, Jaap Kamps, Christof Monz, and Maarten de Rijke. Monolingual document retrieval for european languages. *Information Retrieval*, 7:33–52, 2004.
- [11] David Jones. Everything is information, and information is everything, 2018. URL <https://www.kmworld.com/Articles/White-Paper/Article/Everything-is-Information-and-Information-is-Everything-123561.aspx>.
- [12] Armand Joulin, Edouard Grave, Piotr Bojanowski, Matthijs Douze, H erve J egou, and Tom as Mikolov. Fasttext.zip: Compressing text classification models. 2016.
- [13] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tom as Mikolov. Bag of tricks for efficient text classification. 2016.
- [14] Joos Korstanje. The F1 score, 2022. URL <https://towardsdatascience.com/the-f1-score-bec2bbc38aa6>.
- [15] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Sch utze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

- [16] Tomáš Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. 2013.
- [17] O'Reilly. Text vectorization and transformation pipelines, 2022. URL <https://www.oreilly.com/library/view/applied-text-analysis/9781491963036/ch04.html>.
- [18] Leonard Richardson. Beautiful soup documentation, 2020. URL <https://www.crummy.com/software/BeautifulSoup/bs4/doc/>.
- [19] Scikit-learn. TfidfTransformer, 2022. URL https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfTransformer.html.
- [20] Scikit-learn. TfidfVectorizer, 2022. URL https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html.
- [21] Scikit-learn. Truncatedsvd, 2022. URL <https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.TruncatedSVD.html>.
- [22] Koo Ping Shung. Accuracy, precision, recall or F1?, 2018. URL <https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9>.
- [23] Christine Taylor. Structured vs. unstructured data, 2022. URL <https://www.datamation.com/big-data/structured-vs-unstructured-data/>.
- [24] Wikipedia. Kosinus-Ähnlichkeit, 2022. URL <https://de.wikipedia.org/wiki/Kosinus-Ähnlichkeit>.
- [25] Wikipedia. Word2vec, 2022. URL <https://en.wikipedia.org/wiki/Word2vec>.

A Weitere Listings

```
1 def get_formatted_article(tag):
2     # Alle Links bzw. Fussnoten eliminieren
3     for a in tag.find_all('a'):
4         a.decompose()
5     text = ''
6     for child in tag.children:
7         # Absätze (z.B. OR 6)
8         if child.name == 'p':
9             text = text + get_replaced_text(child) + '\n'
10        # Aufzählungen (z.B. OR 40a oder OR 271a)
11        if child.name == 'dl':
12            for c in child.children:
13                if c.name == 'dt':
14                    text = text + get_replaced_text(c)
15                if c.name == 'dd':
16                    dl = c.find('dl')
17                    if dl == None:
18                        text = text + get_replaced_text(c) + '\n'
19                    else:
20                        text_in_dl = dl.extract()
21                        text = text + get_replaced_text(c) + '\n'
22                        for d in text_in_dl.children:
23                            if d.name == 'dt':
24                                text = text + get_replaced_text(c)
25                            if d.name == 'dd':
26                                text = text + get_replaced_text(c) + '\n'
27        # Tabellen (z.B. OR 361)
28        if child.name == 'div':
29            rows = child.find_all('tr')
30            for row in rows:
31                td = row.find_all('td')
32                for t in td:
33                    text = text + get_replaced_text(t) + ' '
34            text = text + '\n'
35        # Letzten Zeilenumbruch entfernen
36        text = text[:-1]
37    return text
38
39 # Hilfsfunktion für get_formatted_text()
40 def get_replaced_text(tag):
41     text = tag.get_text()
42     # Zeilenumbrüche
43     text = text.replace('\n', '')
44     # Soft-Hyphens
45     text = text.replace('\xad', '')
46     # Non-Breaking Spaces
47     text = text.replace('\xa0', ' ')
48     return text
```

Listing 8: Auslesen des Gesetzesartikels (formatiert)

```

1 def get_article_text(tag):
2     # sup-Tags entfernen (z.B. Absatzbeschriftung)
3     for sup in tag.find_all('sup'):
4         sup.decompose()
5     # dt-Tags entfernen (z.B. Aufzählungszeichen)
6     for dt in tag.find_all('dt'):
7         dt.decompose()
8     # Fussnoten entfernen
9     footnotes = tag.find('div', {'class': 'footnotes'})
10    if footnotes != None:
11        footnotes.decompose()
12    # Leerzeichen vor/nach dd- und td-Tags einfügen
13    dd_tags = tag.find_all('dd')
14    for dd in dd_tags:
15        dd.insert_before(' ')
16        dd.insert_after(' ')
17    td_tags = tag.find_all('td')
18    for td in td_tags:
19        td.insert_before(' ')
20        td.insert_after(' ')
21    return tag.get_text()

```

Listing 9: Auslesen des Gesetzesartikels (unformatiert)


```

1 def get_preprocessed_text(text):
2     # Leerzeichen/Tab am Anfang entfernen
3     if re.match(r'\s', text):
4         text = text[1:]
5     # In Lowercase umwandeln
6     text = text.lower()
7     # Sonderzeichen entfernen
8     text = text.replace('\n', '')          # Zeilenumbrüche
9     text = text.replace('\xa0', ' ')       # Non-Breaking Spaces
10    text = text.replace('\u00ad', '')       # Soft-Hyphens
11    text = text.replace('\u00ab', '')       # Left-Point Double Angle Quotation Mark
12    text = text.replace('\u00bb', '')       # Right-Point Double Angle Quotation Mark
13    text = text.replace('\u2013', ' ')      # En Dash
14    text = text.replace('\u2011', '')       # Non-Breaking Hyphen
15    # Stoppwörter entfernen
16    words = word_tokenize(text)
17    german_stop_words = stopwords.words('german')
18    filtered_text = [w for w in words if not w in german_stop_words]
19    text = " ".join(filtered_text)
20    # Daten verbinden (z.B. OR 40a: '2. april 1908' -> '2april1908')
21    text = re.sub(r'(\d{1,2})\.\s+(\w{3,9})\s+(\d{4})', r'\1\2\3', text)
22    # Satzzeichen entfernen
23    text = text.translate(str.maketrans('', '', string.punctuation))
24    text = text.replace('...', ' ')
25    # Umlaute ersetzen
26    text = text.replace('ä', 'ae')
27    text = text.replace('ö', 'oe')
28    text = text.replace('ü', 'ue')
29    # Mehrfache Leerzeichen entfernen
30    text = re.sub(r'(\s)+', ' ', text)
31    return text

```

Listing 10: Linguistische Vorverarbeitung

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4     <meta charset="utf-8" />
5     <title>GymInf Projekt</title>
6     <link rel="stylesheet" type="text/css" href="{{url_for('static', filename='site.
css')}}"/>
7 </head>
8 <body>
9     <div class="body-content">
10         <h2>GymInf Projekt</h2>
11         <div class="grid-container">
12             <div class="grid-child" name="abfrage">
13                 <b>Abfrage</b>
14                 <form method="POST">
15                     <textarea rows="15" name="queryText" id="queryText">{{query}}</
textarea>
16                     <div>
17                         <select name="mySelect" id="mySelect">
18                             <option value="top10">Top 10</option>
19                             <option value="predictedPositive">Predicted Positive</
option>
20                         </select>
21                     </div>
22                     <input type="submit" value="Abfrage absenden" id="submitButton">
23                     {% if display %}
24                     <script>
25                         let d = "{{display}}";
26                         document.getElementById("mySelect").value = d;
27                     </script>
28                     {% endif %}
29                 </form>
30             </div>
31             <div class="grid-child" name="tfidf">
32                 <b>TF-IDF</b>
33                 <table class="output">
34                     <tr>
35                         <th>Artikel</th>
36                         <th>Kosinus-Ähnlichkeit</th>
37                     </tr>
38                     {% for tuple in tfidf %}
39                     <!-- tuple = (article, cosine similarity, url) -->
40                     <tr>
41                         <td><a href="{{tuple[2]}}" target="_blank">{{tuple[0]}}</a><
/td>
42                         <td>{{tuple[1]}}</td>
43                     </tr>
44                     {% endfor %}
45                 </table>
46             </div>
47             <div class="grid-child" name="fasttext">
48                 <b>FastText</b>

```

```

49         <table class="output">
50             <tr>
51                 <th>Artikel</th>
52                 <th>Kosinus-Ähnlichkeit</th>
53             </tr>
54             {% for tuple in fasttext %}
55             <!-- tuple = (article, cosine similarity, url) -->
56             <tr>
57                 <td><a href="{{tuple[2]}}" target="_blank">{{tuple[0]}}</a><
58                 <td>{{tuple[1]}}</td>
59             </tr>
60             {% endfor %}
61         </table>
62     </div>
63 </div>
64 </div>
65 </body>
66 </html>

```

Listing 11: HTML-Code zur Webapplikation

```

1 * {
2     box-sizing: border-box;
3 }
4
5 body {
6     margin: 0;
7     font-family: Arial, Helvetica, sans-serif;
8     background-color: whitesmoke;
9 }
10
11 a:link, a:visited {
12     color: blue;
13     text-decoration: none;
14 }
15
16 table {
17     width: 100%;
18 }
19
20 th, td {
21     text-align: left;
22     width: 50%;
23 }
24
25 footer {
26     font-size: 0.875em;
27 }
28
29 .body-content {
30     padding: 10px 20px;
31 }
32
33 .grid-container {
34     display: grid;
35     grid-template-columns: 1fr 1fr 1fr;
36     grid-gap: 10px;
37 }
38
39 .grid-child {
40     background-color: lightblue;
41     border: 2px solid black;
42     padding: 5px;
43 }
44
45 .output {
46     font-size: 0.875em;
47     margin: 0;
48 }
49
50 #queryText {
51     width: 100%;
52     outline: none;

```

```
53     resize: none;
54 }
55
56 #mySelect {
57     width: 100%;
58     margin: 5px 0px;
59     text-align: center;
60 }
61
62 #sendButton {
63     width: 100%;
64 }
```

Listing 12: CSS-Code zur Webapplikation

```
1 def top_n_articles(list, n):
2     return list[0:n]
3
4 def articles_over_threshold(list, threshold):
5     n = 0
6     for tuple in list:
7         if tuple[1] > threshold:
8             n += 1
9         else:
10            break
11    return list[0:n]
```

Listing 13: Top-n-Artikel und Artikel über Schwellenwert

```

1 with open('articles_preprocessed.json') as json_file:
2     docs = json.load(json_file)
3
4 N = len(docs)
5
6 test_queries = {
7     'Widerrufsrecht bei Haustürgeschäften': ['or_art_40a', 'or_art_40b', '
8         or_art_40_c', 'or_art_40_d', 'or_art_40_e', 'or_art_40_f'],
9     'Erfüllungsort bei Geldschulden': ['or_art_74'],
10    'Transportkosten beim Kaufvertrag': ['or_art_189'],
11    'Wohnung untervermieten': ['or_art_262'],
12    'Fristlose Kündigung des Arbeitsvertrags': ['or_art_337', 'or_art_337_a', '
13        or_art_337_b', 'or_art_337_c'],
14    'Konkurrenzverbot im Arbeitsvertrag': ['or_art_340', 'or_art_340_a', 'or_art_340_b
15        ', 'or_art_340_c'],
16    'Mindestkapital AG': ['or_art_621'],
17    'Auflösung einer Verlobung': ['zgb_art_91', 'zgb_art_92', 'zgb_art_93'],
18    'Scheidung einer Ehe': ['zgb_art_111', 'zgb_art_112', 'zgb_art_114', 'zgb_art_115'
19        ],
20    'Altersunterschied bei Adoption': ['zgb_art_264_d'],
21    'Enterbung eines Kindes': ['zgb_art_477', 'zgb_art_478', 'zgb_art_479', '
22        zgb_art_480'],
23 }
24
25 methods = ['tfidf', 'fasttext']
26 thresholds = [0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, 0.6,
27     0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95]
28
29 average_accuracy = {}
30 average_recall = {}
31 average_precision = {}
32 average_f1_score = {}
33
34 for method in methods:
35     print('Method:', method)
36     for threshold in thresholds:
37         accuracies = []
38         recalls = []
39         precisions = []
40         f1_scores = []
41         for query in test_queries.keys():
42             query_preprocessed = get_preprocessed_text(query)
43             sorted_articles = []
44             # Predicted Positives
45             if method == 'tfidf':
46                 sorted_articles = tfidf_sort_articles(docs, query_preprocessed)
47             if method == 'fasttext':
48                 sorted_articles = fasttext_sort_articles(docs, query_preprocessed)
49             predicted_positives_tuples = articles_over_threshold(sorted_articles,
50                 threshold)
51             predicted_positives = len(predicted_positives_tuples)
52             predicted_positives_articles = []

```

```

46     for tuple in predicted_positives_tuples:
47         predicted_positives_articles.append(tuple[0])
48     # Predicted Negatives
49     predicted_negatives = N - predicted_positives
50     # Actual Positives
51     actual_positives_articles = test_queries[query]
52     actual_positives = len(actual_positives_articles)
53     # Actual Negatives
54     actual_negatives = N - actual_positives
55     # True Positives
56     true_positives = 0
57     for actual_positive in actual_positives_articles:
58         if actual_positive in predicted_positives_articles:
59             true_positives += 1
60     # False Positive
61     false_positives = predicted_positives - true_positives
62     # True Negative
63     true_negatives = actual_negatives - false_positives
64     # False Negatives
65     false_negatives = actual_positives - true_positives
66     # Accuracy, Recall, Precision, F1 Score
67     accuracy = (true_positives + true_negatives) / N
68     if true_positives == 0:
69         recall = 0
70         precision = 0
71     else:
72         recall = true_positives / (true_positives + false_negatives)
73         precision = true_positives / (true_positives + false_positives)
74     if recall == 0 and precision == 0:
75         f1_score = 0
76     else:
77         f1_score = 2 * (precision * recall) / (precision + recall)
78     accuracies.append(accuracy)
79     recalls.append(recall)
80     precisions.append(precision)
81     f1_scores.append(f1_score)
82     # Average Accuracy, Recall, Precision, F1 Score
83     average_accuracy[threshold] = mean(accuracies)
84     average_recall[threshold] = mean(recalls)
85     average_precision[threshold] = mean(precisions)
86     average_f1_score[threshold] = mean(f1_scores)

```

Listing 14: Evaluation

B Truncated SVD

```
1 from sklearn.decomposition import TruncatedSVD
2
3 def tfidf_sort_articles(docs, query):
4     index_docs = [n for n in docs]
5     # TF-IDF-Vektoren Dokumente
6     vectorizer = TfidfVectorizer(analyzer='char_wb', ngram_range=(5,5), sublinear_tf
7                                 =True)
8     tfidf_docs = vectorizer.fit_transform(docs.values())
9     vocab = vectorizer.vocabulary_
10    svd = TruncatedSVD(n_components=100, n_iter=7, random_state=42)
11    svd.fit(tfidf_docs)
12    tfidf_docs_svd = svd.transform(tfidf_docs)
13    # TF-IDF-Vektor Abfrage
14    dict_query = {'query': query}
15    vectorizer = TfidfVectorizer(analyzer='char_wb', vocabulary=vocab, ngram_range
16                                =(5,5), sublinear_tf=True)
17    tfidf_query = vectorizer.fit_transform(dict_query.values())
18    tfidf_query_svd = svd.transform(tfidf_query)
19    # Kosinus-Ähnlichkeit zwischen Abfrage und Dokumenten
20    return doc_cossim_url_tuples(index_docs, tfidf_docs_svd, tfidf_query_svd)
21
22 def fasttext_sort_articles(docs, query):
23     index_docs = [n for n in docs]
24     ft = fasttext.load_model('fastText/cc.de.300.bin')
25     # fastText-Vektoren Dokumente
26     fasttext_docs = []
27     for doc in docs.values():
28         vect_doc = ft.get_sentence_vector(doc)
29         fasttext_docs.append(vect_doc)
30     svd = TruncatedSVD(n_components=100, n_iter=7, random_state=42)
31     svd.fit(fasttext_docs)
32     fasttext_docs_svd = svd.transform(fasttext_docs)
33     # fastText-Vektoren Abfrage
34     fasttext_query = ft.get_sentence_vector(query)
35     fasttext_query = fasttext_query.reshape(1, -1)
36     fasttext_query_svd = svd.transform(fasttext_query)
37     # Kosinus-Ähnlichkeit zwischen Abfrage und Dokumenten
38     return doc_cossim_url_tuples(index_docs, fasttext_docs_svd, fasttext_query_svd)
```

Listing 15: Truncated SVD

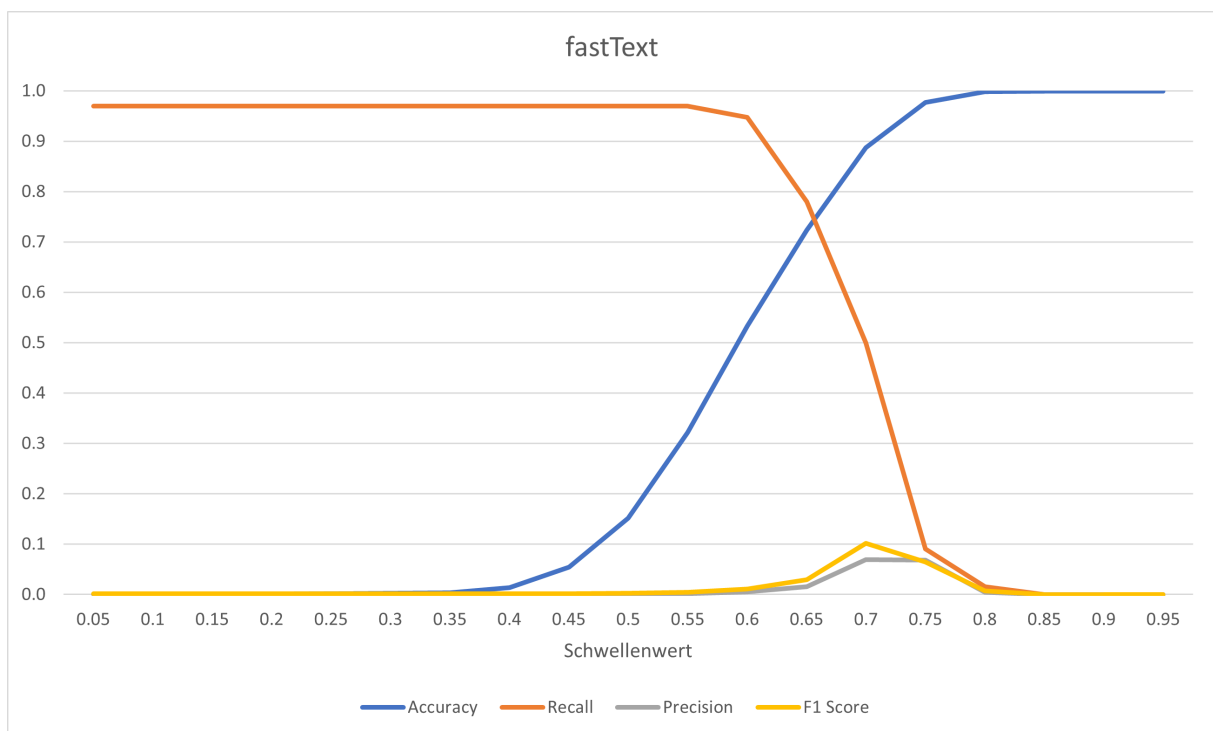
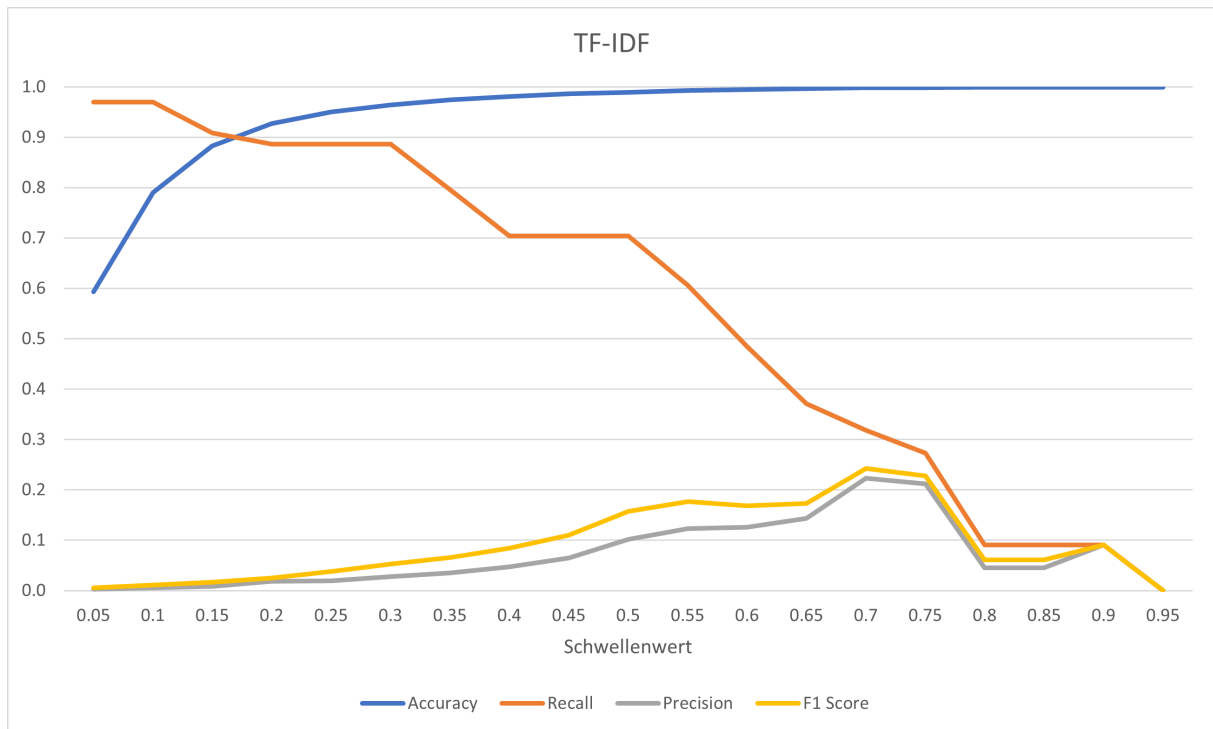


Abbildung 13: Evaluation (Truncated SVD)

C Kurzanleitung

Um die Resultate dieses Berichts zu reproduzieren, können folgende Schritte ausgeführt werden:

1. Die Daten und den Code zu diesem Projekt von GitHub herunterladen (<https://github.com/baltisberger/gyminf-projekt>).
2. FastText installieren (<https://fasttext.cc/docs/en/support.html>) und das vortrainierte fastText-Modell für Deutsch (cc.de.300.bin) herunterladen (<https://fasttext.cc/docs/en/crawl-vectors.html>).
3. In `functions.py` die Funktion `fasttext_sort_articles()` öffnen und den Pfad bei `fasttext.load_model()` entsprechend anpassen.
4. Die Webapplikation mit `app.py` starten. In der Webapplikation eine beliebige Abfrage eingeben, die gewünschte Anzeigeoption wählen (Top 10 oder Predicted Positive) und auf “Abfrage absenden” drücken.
Leider dauert das Anzeigen der Resultate eine gewisse Zeit, da das Laden des fastText-Modells (7.2 GB) eine gewisse Zeit braucht.
5. Die Evaluationsresultate können mit `evaluation.py` reproduziert werden.