



Lerneinheit: Preprocessing mit NLTK

Mareike Schumacher  ¹

Michael Vauth 

1. Universität Regensburg

forTEXT

| | | | |
|--------------------|---|-----------------------|---|
| Thema: | Korpusbildung | DOI: | 10.48694/fortext.3809 |
| Jahrgang: | 1 | Ausgabe: | 2 |
| Erscheinungsdatum: | 12-06-2024 | Erstveröffentlichung: | 2022-04-04 auf fortext.net |
| Lizenz: |  | | open  access |

Allgemeiner Hinweis: Rot dargestellte *Begriffe* werden im Glossar am Ende des Beitrags erläutert. Alle externen Links sind auch am Ende des Beitrags aufgeführt.

Eckdaten der Lerneinheit

- Anwendungsbezug: Textvorverarbeitung von Goethes *Die Leiden des jungen Werthers*
- Methodik: Korpusbildung und **Preprocessing**
- Angewendetes Tool: NLTK
- Lernziele: Einzelne Schritte zur Textbereinigung, wie z.B. Löschen von Leerzeilen, Entfernen von Stoppwörtern (vgl. **Stoppwortliste**) oder Tokenisierung (vgl. **Type/Token**) auswählen und durchführen können
- Dauer der Lerneinheit: ca. 60 Minuten
- Schwierigkeitsgrad des Tools: mittel

1. Anwendungsbeispiel

Anhand einer Reintextversion (vgl. **Reintext-Version**) von Goethes *Die Leiden des jungen Werthers* erlernen Sie in dieser Lerneinheit einige grundlegende Arbeitsschritte der Aufbereitung (vgl. **Preprocessing**) von Texten für digitale Analysen. Die Arbeitsschritte können unabhängig von Form und Inhalt des Textes oder des Korpus durchgeführt werden, das Sie analysieren möchten. Sie benötigen lediglich eine TXT-Version des Textes, den Sie untersuchen möchten. Achten Sie bei der Auswahl der Funktionen für eigene digitale Analysen aber darauf, ob sie zu den gewählten Methoden passen. Nutzen Sie z.B. Named Entity Recognition (Schumacher 2024a; Schumacher 2024b) für maschinelles Lernen, so müssen Sie Ihr Korpus tokenisieren, d.h. in Ein-Wort-Listen umwandeln. Wenn Sie dagegen Topic Modeling (Horstmann 2024) anwenden möchten, brauchen Sie diesen Schritt nicht. Stattdessen ist es wichtig, Stoppwortlisten anwenden zu können.

2. Vorarbeiten

Wir haben für diese Lerneinheit ein Jupyter Notebook (Kluyver u. a. 2016) vorbereitet, das Sie sowohl zum Erlernen als auch zum späteren Ausführen der Funktionen im Rahmen eigener Projekte nutzen können. Ein solches Notebook ermöglicht es Ihnen, die vorbereiteten kurzen Code-Einheiten per Klick aufzurufen. Installieren Sie bitte vorab **Anaconda** (Anaconda Software Distribution 2020). Haben Sie Anaconda installiert, so nutzen Sie dieses Programm, um NLTK (Bird, Klein und Loper 2009) zu installieren. Starten Sie dafür Anaconda und wechseln Sie links im Menü zu „Environments“. Geben Sie dann oben rechts in die Suchleiste „NLTK“ ein und wählen Sie im Drop-Down-Menü oben in der Mitte „All“. Im Suchergebnis erscheint nun eine Zeile für „NLTK - Natural Language Toolkit“, vor der links ein leerer Kasten steht.

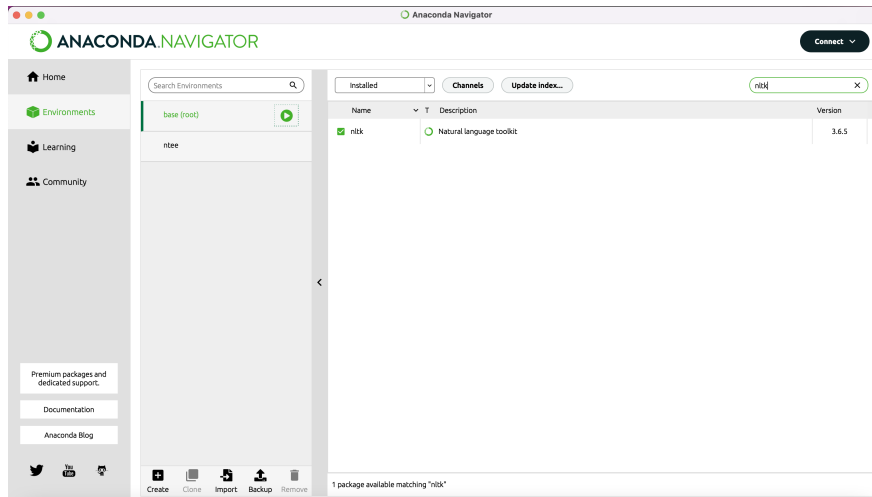


Abb. 1: Suche nach „NLTK“ innerhalb von Anaconda

Klicken Sie in diesen Kasten, sodass dort ein kleiner Pfeil erscheint (vgl. Abb. 1) und gehen Sie dann auf den Button „Apply“ unten rechts. Es kann einen Moment dauern bis das Programm Ihnen anzeigt, was die Installation von NLTK bedeutet, bzw. welche anderen Packages zusätzlich installiert oder aktualisiert werden müssen. Sobald Ihnen die Liste angezeigt wird, können Sie mit „Apply“ die Installation bestätigen (vgl. Abb. 2).

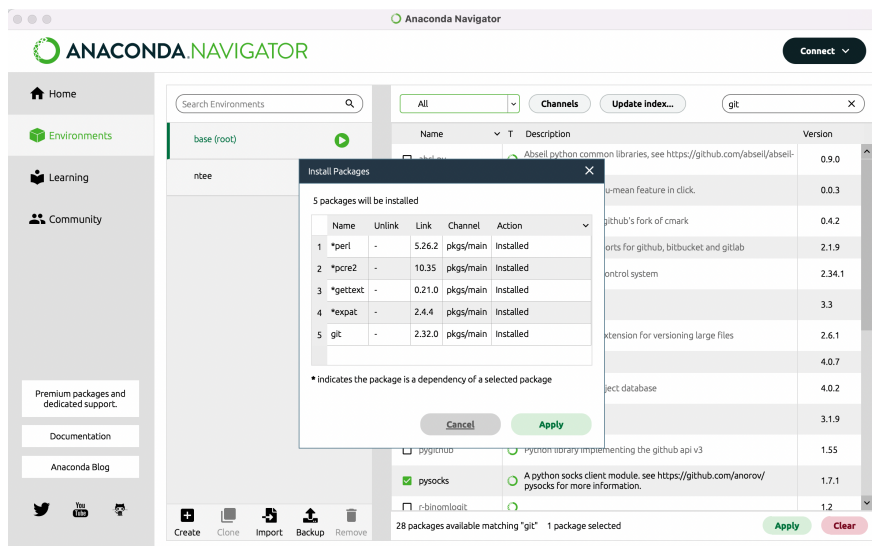


Abb. 2: Installation innerhalb von Anaconda

Laden Sie nun den von uns vorbereiteten Goethe-Text und das Jupyter Notebook zum Preprocessing mit NLTK herunter. Gehen Sie dazu in unser [forTEXT.net-GitHub-Repository](#) und dort zunächst auf den grünen Button „Code“ und dann auf „Download ZIP“ (vgl. Abb. 3).

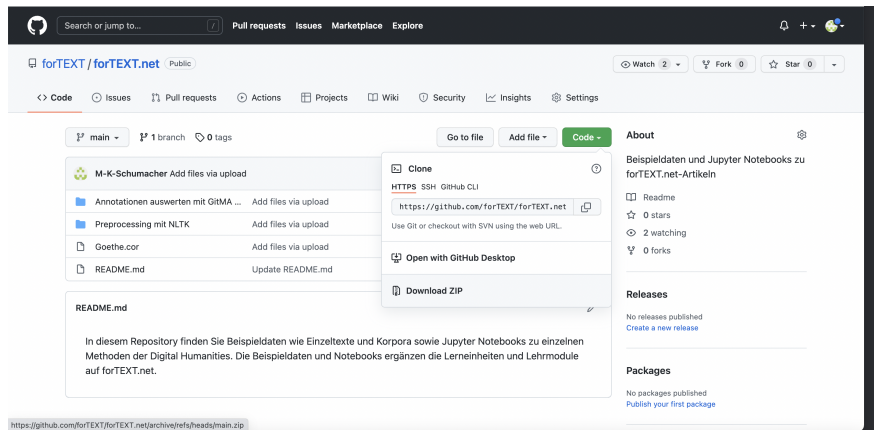


Abb. 3: *forTEXT.net* GitHub-Repository herunterladen

Entpacken Sie dann das ZIP-Archiv, indem Sie es in Ihrem Ordnersystem ausfindig machen und dieses doppelt anklicken. Gehen Sie nun zurück zu Anaconda. Klicken Sie links im Menü auf „Home“ und dann in der Kachel „Jupyter Notebooks“ auf „Launch“ (vgl. Abb. 4).

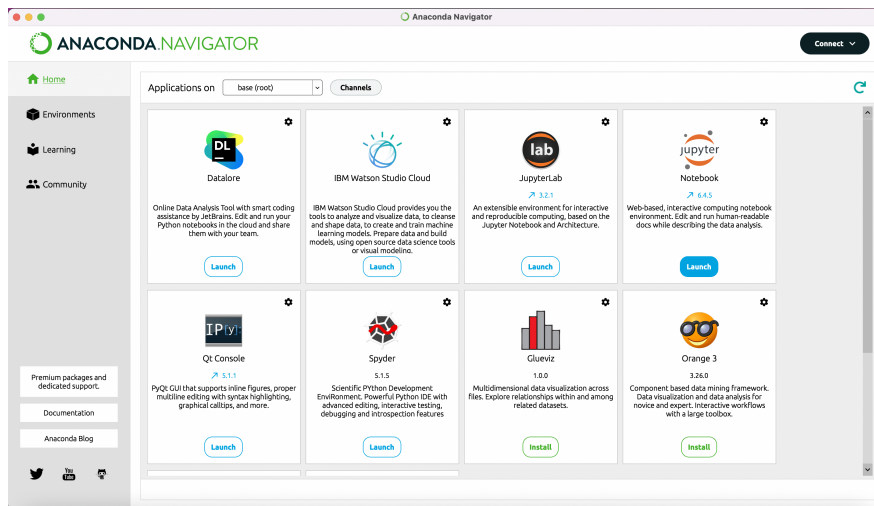


Abb. 4: *Jupyter Notebook* in Anaconda starten

Es öffnet sich dann ein Browser-Fenster, in welchem Sie Ihre Ordner-Struktur sehen. Navigieren Sie zu Ihrem Downloads-Ordner und dann in den gerade entpackten Ordner „Preprocessing mit NLTK“. Klicken Sie hier auf die Datei „Preprocessing mit NLTK.ipynb“, um das Notebook zu öffnen. Sie können nun der Anleitung im Notebook folgen. Falls Sie eine Auswahl unserer Codebeispiele lieber zu einem eigenen Notebook zusammenfassen wollen, finden Sie die einzelnen Schritte im Folgenden dokumentiert.

3. Funktionen

Für diese Lerneinheit haben wir das Preprocessing, mit dem Sie Texte für die digitale Analyse mit anderen Tools vorbereiten können, in zwei Komplexe unterteilt.

Im ersten Komplex finden Sie Funktionen, mit denen Sie die Form eines Textes verändern können, ohne dabei inhaltlich einzugreifen. Mit diesen Funktionen können Sie

- einen Text in einzelne Wörter oder Sätze unterteilen (Tokenisierung (vgl. **Type/Token**) auf Wort- oder Satzebene)
- den gesamten Text in Kleinschreibung umwandeln
- doppelte Leerzeichen und Leerzeilen entfernen.

Im zweiten Komplex finden Sie folgende Funktionen, mit denen Sie einzelne Elemente aus einem Text herauslösen können: Wörter entfernen, die weniger als drei Zeichen haben. So können Sie damit meist solche Fehler eliminieren, die auf ein suboptimales **OCR** zurückzuführen sind. Sie können außerdem:

- Zahlen und Satzzeichen aus einem Text entfernen
- Sätze mit einer bestimmten Wortanzahl aus dem Text entfernen.
- eine Liste von Stoppwörtern (vgl. [Stoppwortliste](#)) (Wörter, die Sie in Ihrer Analyse nicht mit einbeziehen möchten) aus dem Text entfernen.

Die Lerneinheit ist so aufgebaut, dass Sie jede Funktion einzeln ausführen können. Suchen Sie sich die Funktionen heraus, die Sie für Ihre Korpusvorbereitung (vgl. [Preprocessing](#)) brauchen. Funktionen, die Sie nicht benötigen, können Sie einfach überspringen. Um die Kombination unterschiedlicher Preprocessing-Schritte möglichst unkompliziert zu ermöglichen, stellen wir am Ende der Lerneinheit drei Preprocessing-Pipelines vor, die Sie ebenfalls benutzen können.

NLTK laden

Bevor Sie mit dem Preprocessing Ihres Textes beginnen können, müssen Sie das NLTK-Package (Natural Language Processing Toolkit) und ein paar zusätzliche Packages laden. Diesen Schritt müssen Sie immer ausführen, unabhängig davon welchen Schritt oder welche Schritte des Preprocessings Sie übernehmen möchten. Sie können den folgenden Code einzeln ausführen oder vor den Code einer Funktion setzen.

```
import nltk
import re
import os
```

Daten laden

Als nächstes legen wir fest, welche Textdatei die nachfolgenden Preprocessing-Schritte nutzt und in welchem Verzeichnis diese liegt. In das Datenverzeichnis werden auch die bearbeiteten Dateien gespeichert.

```
data_dir = 'Daten'
text_file = '1774-Werther.txt'
```

Funktionen zur Anpassung der Textform

Manche digitalen Methoden und Tools zur Textanalyse benötigen vorverarbeitete Dokumente, die eine ganz bestimmte Form aufweisen. Manchmal benötigen Sie z.B. eine Darstellung, in der in jeder Zeile ein Wort steht oder pro Zeile ein Satz. In der digitalen Stilometrie kann es manchmal sinnvoll sein, Texte so vorzubereiten, dass sie ausschließlich Kleinschreibung aufweisen, um die Großschreibung häufiger Wörter wie „der“, „die“ oder „das“ am Satzanfang herauszunehmen. In diesem Abschnitt finden Sie Funktionen des Python-Packages NLTK, mit denen Sie solche Operationen durchführen können, die nicht in den eigentlichen Text, sondern nur in dessen Form eingreifen.

Tokenisierung auf Wortebene

Mit der Funktion zur Tokenisierung können Sie Elemente in einem Text vereinzeln bzw. einzeln markieren. Mit NLTK können Sie Tokenisierung sowohl auf Wort- als auch auf Satzebene durchführen. Um die Tokenisierung auf Wortebene durchzuführen und den tokenisierten Text in einer neuen Datei zu speichern, klicken Sie nun in die unten stehende Box. Klicken Sie dann oben im Menü auf „Run“, um den Vorgang zu starten. Sobald links in den eckigen Klammern eine kleine Zahl erscheint, müsste das Programm eine neue Datei mit Ihren Daten in Ihrem Ordnersystem abgelegt haben.

Dabei wird jedes Token in einer neuen Textzeile ausgegeben.

```
input_file = os.path.join(data_dir, text_file)
basename = os.path.splitext(input_file)[0]

with open('Downloads/1774-Werther.txt', 'r', encoding='utf-8') as input:
    text = input.read()
```

```

# Tokenisierung
tokens = nltk.word_tokenize(text, 'german')

# Der join() Befehl verbindet Elemente einer Liste mit einem String.
# Wir verwenden hier den Zeilenumbruch "\n" als Verbindungselement, um eine
  ↪ vertikale Wortliste zu erstellen.

output_str = '\n'.join(tokens)

with open('Downloads/Werther_Tokens.txt', 'w', encoding='utf-8') as output:
    output.write(output_str)

```

Tokenisierung auf Satzebene

Um die Tokenisierung auf Satzebene durchzuführen und den tokenisierten Text in einer neuen Datei zu speichern, klicken Sie nun in die unten stehende Box. Klicken Sie dann oben im Menü auf „Run“, um den Vorgang zu starten. Sobald links in den eckigen Klammern eine kleine Zahl erscheint, müsste das Programm eine neue Datei mit Ihren Daten in Ihrem Ordnersystem abgelegt haben.

Dabei wird jeder Satz in einer neuen Textzeile ausgegeben.

```

with open(input_file, 'r', encoding='utf-8') as input:
    text = input.read()

sentences = nltk.sent_tokenize(text, language='german')
output_str = '\n'.join(sentences)

with open(basename + '_Saetze.txt', 'w', encoding='utf-8') as output:
    output.write(output_str)

```

Umwandlung in Kleinschreibung

Für manche Methoden und / oder manche Sprachen kann es sinnvoll sein, den gesamten Text in Kleinschreibung zu transformieren. Wenn Sie Ihren Text auf diese Weise umwandeln und das Ergebnis in einer neuen Datei speichern möchten, nutzen Sie dazu diese Code-Zeilen. Das Programm legt dann eine neue Datei mit dem unten aufgeführten Namen in Ihrem Ordnersystem ab.

```

with open(input_file, 'r', encoding='utf-8') as input:
    text = input.read()

lower_text = text.lower()

with open(basename + '_Saetze_lowercase.txt', 'w', encoding='utf-8') as output:
    output.write(lower_text)

```

Doppelte Leerzeichen und Leerzeilen entfernen

Manchmal kommen durch die Vorverarbeitung oder auch durch die Digitalisierung Leerzeichen und -zeilen in Texte, die Sie vielleicht bei Ihren weiteren Verarbeitungsschritten stören können. Die folgenden Funktionen helfen Ihnen dabei, doppelte Leerzeichen und Leerzeilen zu entfernen.

Doppelte Leerzeichen entfernen

Wenn Sie doppelte Leerzeichen in Ihrem Text entfernen möchten, nutzen Sie dazu den folgenden Code. Ihr Ergebnis wird dann in einer neuen Datei gespeichert.

```

with open(input_file, 'r', encoding='utf-8') as input:
    text = input.read()

# Mit der re.sub() Funktion können wir reguläre Ausdrücke verwenden, um
    ↪ bestimmte Textelemente zu entfernen.
# Mehrfachvorkommen von Leerzeichen werden mit dem regulärem Ausdruck ' +'
    ↪ gefunden.
# Um reguläre Ausdrücke auszuprobieren, kann man zum Beispiel diese Website
    ↪ verwenden: <https://regexr.com/>

clean_text = re.sub('_+', # der reguläre Ausdruck mit dem wir die Zeichen
    ↪ finden, die entfernt werden sollen.
'_ ', # Was eingefügt werden soll. In diesem Fall ein einfaches Leerzeichen
text # Der Text den wir bereinigen wollen.
)

with open(basename + '_einfacheLeerzeichen.txt', 'w', encoding='utf-8') as
    ↪ output:
    output.write(clean_text)

```

Leerzeilen entfernen

Wenn Sie Leerzeilen in Ihrem Text entfernen möchten, nutzen Sie dazu den folgenden Code. Sobald links in den eckigen Klammern eine Zahl erscheint, wurde Ihr Ergebnis in einer neuen Datei gespeichert.

```

with open(input_file, 'r', encoding='utf-8') as input:
    text = input.read()

# Mit diesem regulären Ausdruck suchen wir nach Mehrfachvorkommen von
    ↪ Zeilenumbrüchen.
clean_text = re.sub('\n+', '_ ', text)

```

```
with open(basename + '_ohneLeerzeilen.txt', 'w', encoding='utf-8') as output: output.write(clean_text)
```

Funktionen zum Herauslöschten einzelner Elemente

Sie wissen nun, wie Sie formale Elemente Ihres Textes verändern können. Sie können auf Satz- und Wortebene tokenisieren, durchgängige Kleinschreibung bewirken und doppelte Leerzeichen sowie Leerzeilen entfernen. Für manche Methoden, wie z.B. das Topic Modeling oder auch word2vec (@ Horstmann 2024; Schumacher 2024c) kann es sinnvoll sein, bei der Vorverarbeitung auch in den Text selbst einzugreifen. Von großer Bedeutung ist hier die Entfernung von Stopwörtern, d.h. Wörtern, die nicht in die Analyse einbezogen werden sollen. Auch das Entfernen von Zahlen, Interpunktionszeichen oder kurzen Wörtern oder Sätzen kann für einige Methoden der Digital Humanities sinnvoll sein. Wählen Sie sich die Funktionen, die Sie brauchen einfach aus den folgenden Blöcken aus und überspringen Sie diejenigen, die für Sie nicht sinnvoll sind.

Bei manchen Funktionen können Sie wählen, in welchem Format Sie ihr Ergebnis abspeichern wollen. In diesem Fall finden Sie zu einer Funktion mehrere Abschnitte.

Entfernen von Stopwörtern

Das NLTK-Package stellt auch eine Reihe von Stopwortlisten für verschiedene Sprachen zur Verfügung. Im folgenden Beispiel führen Sie die Entfernung von Stopwörtern mit der Liste deutscher Stopwörter durch.

```

from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

```

```

with open(input_file, 'r', encoding='utf-8') as input:
    text = input.read()

language = 'german'
token = word_tokenize(text, language)

# Wir verwenden eine List Comprehension, mit der nur die Token in die neue
  ↳ Liste übernommen werden, die nicht in der Stopwords Liste sind.

filtered_token = [token for token in token if token not in stopwords.words(
  ↳ language)]

# Erneut erstellen wir eine vertikale Tokenliste, die in die neue Datei
  ↳ ausgegeben wird.
clean_token = '\n'.join(filtered_token)

with open(basename + '_ohneStopwords.txt', 'w', encoding='utf-8') as output:
    output.write(clean_tokens)

```

Wenn Sie Stoppwörter in einer anderen Sprache entfernen wollen, geben Sie in oben stehendem Code statt ‚german‘ einfach die Sprache ein, die Sie benötigen. Um eine Liste aller Sprachen zu bekommen, für die NLTK Stoppwortlisten bereitstellt, können Sie den folgenden Code nutzen.

```

from nltk.corpus import stopwords
print(stopwords.fileids())

```

Entfernen von Satzzeichen und Zahlen

Je nach Fragestellung die Sie untersuchen wollen, kann es sinnvoll sein, Satzzeichen und Zahlen, also alle nicht-alphabetischen Elemente, aus einem Text herauszufiltern. Wenn Sie Ihren Text oder Ihr Korpus auf diese Weise vorverarbeiten wollen, finden Sie hier dazu den Code.

```

with open(input_file, 'r', encoding='utf-8') as input:
    text = input.read()

from nltk.tokenize import word_tokenize
token = word_tokenize(text)

# Alle nicht-alphabetischen Zeichen werden herausgefiltert
filtered_token = [token for token in token if token.isalpha()]
clean_tokens = '\n'.join(filtered_tokens)

with open(basename + '_ohneSatzzeichenundZahlen.txt', 'w', encoding='utf-8') as
  ↳ output:
    output.write(clean_tokens)

```

Entfernen von Zahlen

Wenn Sie zwar Zahlen, aber keine Satzzeichen entfernen wollen, können Sie dafür den folgenden Code verwenden.

```

with open(input_file, 'r', encoding='utf-8') as input:
    text = input.read()

# Hier verwenden wir wieder die re.sub() Funktion
#'[0-9]' -- Der reguläre Ausdruck für Zahlen

text_without_numbers = re.sub('[0-9]', '', text)

```

```
# Hier tokenisieren wir den bereinigten Text
tokens = nltk.word_tokenize(text_without_numbers, 'german')
clean_tokens = '\n'.join(tokens)

with open(basename + '_ohneZahlen.txt', 'w', encoding='utf-8') as output:
    output.write(clean_tokens)
```

Entfernen von Satzzeichen

Wenn Sie ausschließlich Satzzeichen aus Ihrem Text herausfiltern möchten, können Sie das mit diesem Code tun.

```
import string

with open(input_file, 'r', encoding='utf-8') as input:
    text = input.read()

from nltk.tokenize import word_tokenize
tokens = word_tokenize(text)

# Filtern der Interpunktionszeichen
filtered_tokens = [token for token in tokens if token not in string.punctuation
↔ ]

clean_tokens = '\n'.join(filtered_tokens)

with open(basename + '_ohneSatzzeichen.txt', 'w', encoding='utf-8') as output:
    output.write(clean_tokens)
```

Entfernen kurzer Wörter

Wörter, die nur aus drei Buchstaben oder weniger bestehen, sind häufig Funktionswörter. Ein anderer Grund, aus dem Sie evtl. kurze Wörter bei der Vorverarbeitung Ihres Korpus herausfiltern möchten, ist, dass Fehler, die durch mangelhafte OCR-Erkennung entstanden sein können, auf diese Weise minimiert werden können. OCR-Software erkennt manchmal Bindestriche am Seitenrand nicht und „zerhackt“ dann die betroffenen Wörter.

Wenn Sie aus oben genannten oder anderen Gründen Wörter einer bestimmten Länge aus Ihrem Text herausfiltern möchten, so können Sie das mit unten stehendem Code tun.

Mit diesem Code filtern Sie Wörter aus einem Text heraus, die kürzer als drei Buchstaben sind. Möchten Sie Wörter bis zu einer anderen Länge aus Ihrem Text herausholen, so ersetzen Sie einfach die 3. Wenn Sie lange Wörter herausfiltern möchten statt kurze, so ändern Sie das > Zeichen und machen Sie daraus <. Passen Sie in diesem Fall auch die Zahl an.

```
with open(input_file, 'r', encoding='utf-8') as input:
    text = input.read()

from nltk.tokenize import word_tokenize
tokens = word_tokenize(text)

# Filtern nach der Tokenlänge
filtered_tokens = [token for token in tokens if len(token) > 3]

clean_tokens = '\n'.join(filtered_tokens)

with open(basename + '_nurlangeToken.txt', 'w', encoding='utf-8') as output:
    output.write(clean_tokens)
```

Entfernen kurzer Sätze

Wenn Sie Sätze einer bestimmten Länge aus Ihrem Text herausfiltern möchten, so können Sie dies mit unten stehendem Code tun. Passen Sie dazu den Dateipfad in der ersten Zeile so an, dass dieser zu Ihrem Text oder Textkorpus führt. Wenn Sie mit unserem Beispieltext arbeiten, müssen Sie hier evtl. gar nichts anpassen. Mit unserem Beispielcode können Sie Sätze aus einem Text herausfiltern, die kürzer sind als 4 Wörter; passen Sie die Zahl 4 im Code also ggf. an. Elemente wie „Erster Theil“, die nicht zum eigentlichen Erzähltext gehören, sondern Formelemente sind, bekommen Sie damit aus Ihrem Text herausgerechnet. Bedenken Sie aber auch, dass derselbe Code auch Sätze wie „Aber halt!“ aus dem Text herauswirft, obwohl diese zum eigentlichen Erzähltext gehören. Nutzen Sie diesen Vorverarbeitungsschritt also bewusst und vorsichtig!

Im Folgenden Finden Sie drei Varianten des Codes, die Ihnen unterschiedliche Output-Dateien ausgeben. Das erste Beispiel gibt Ihnen einen Fließtext, in dem alle (übrig gebliebenen) Tokens und Sätze als solche markiert sind.

Entfernen kurzer Sätze und Speichern als Fließtext

Möchten Sie Sätze einer bestimmten Länge aus Ihrem Text herausrechnen und den Text in einer Datei als Fließtext abspeichern, in dem alle Wörter und Sätze als solche markiert sind, so können Sie dazu diesen Code nutzen.

```
from nltk.tokenize import sent_tokenize
    ↪ , word_tokenize

with open(input_file, 'r', encoding='utf-8') as input:
    text = input.read()

sentences = nltk.sent_tokenize(text, 'german')

# Filtern anhand der Tokenanzahl pro Satz
long_sentences = [sentence for sentence in sentences if len(word_tokenize(
    ↪ sentence, 'german')) > 4]

# Hier fügen wir alle getrennten Sätze wieder zusammen.
long_sentences_str = ' '.join(long_sentences)

with open(basename + '_langeSaetze_Fliesstext.txt', 'w', encoding='utf-8') as
    ↪ output:
    output.write(long_sentences_str)
```

Entfernen kurzer Sätze und Speichern mit einem Satz pro Zeile

Möchten Sie Sätze einer bestimmten Länge aus Ihrem Text herausrechnen und den Text in einer Datei abspeichern, in der jeder Satz eine Zeile bildet und Wörter und Sätze nicht markiert sind, so nutzen Sie dazu unten stehenden Code.

```
with open(input_file, 'r', encoding='utf-8') as input:
    text = input.read()

sentences = nltk.sent_tokenize(text, 'german')
filtered_sentences = [sent for sent in sentences if len(nltk.word_tokenize(
    ↪ sentence, 'german')) > 4]
output_str = '\n'.join(filtered_sentences)

with open(basename + '_langeSaetze_Zeilen.txt', 'w', encoding='utf-8') as
    ↪ output:
    output.write(output_str)
```

Sie haben nun einige zentrale Funktionen zur Vorverarbeitung (Preprocessing) Ihrer Texte für eine digitale

Analyse kennengelernt. Sie können die Funktionen alle auch einzeln nutzen. Speichern Sie sich unser Notebook also gerne ab und nutzen es immer dann, wenn Sie einzelne Funktionen davon zum Preprocessing brauchen.

Preprocessing Pipelines

Um die Funktionen zu Textbereinigung, Wort-Tokenisierung und Satz-Tokenisierung übersichtlich und mit wenig Code zu verwenden, können wir auch Preprocessing Pipelines bauen. Wir stellen im folgenden 3 Pipelines vor, die Sie nutzen und weiterentwickeln können.

Text Pipeline

```
# Import des Regular Expression Package
import re

class TextPipeline:
    def __init__(self, input_file: str):
        with open(input_file, 'r', encoding='utf-8') as input:
            self.text = input.read()

    def remove_double_spaces(self):
        self.text = re.sub('_+', '_', self.text)

    def lowercase(self):
        self.text = self.text.lower()

    def remove_blank_lines(self):
        self.text = re.sub('\n+', '_', self.text)

    def save(self, output_file: str = 'clean_text.txt'):
        with open(output_file, 'w', encoding='utf-8') as output:
            output.write(self.text)

# Um die Text-Pipeline zu beginnen, geben wir den Dateinamen unseres Textes an:

text_pipeline = TextPipeline(input_file='Daten/1774-Werther.txt')

# Jetzt können wir alle Bereinigungs-schritte beliebig hinzufügen oder weglassen
text_pipeline.remove_double_spaces()
text_pipeline.lowercase()
text_pipeline.remove_blank_lines()

# Als letztes sichern wir die bereinigte Tokenliste und wählen dafür einen
    ↪ Dateinamen aus:
text_pipeline.save(output_file='Daten/clean_text.txt')

# Wir können uns das Ergebnis aber zur Überprüfung zusätzlich hier ausgeben
    ↪ lassen:
print(text_pipeline.text)
```

Token Pipeline

```
import string
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
```

```

class TokenPipeline:
# Die Klasse "TokenPipeline" enthält alle Bereinigungsfunktionen, die oben
  ↪ erklärt wurden. Weitere Bereinigungsschritte können ergänzt werden. Im
  ↪ ersten Schritt der Token-Pipeline wird der Text geladen und tokenisiert.

    def __init__(self, input_file: str, language: str = 'german'):
        with open(input_file, 'r', encoding='utf-8') as input:
            text = input.read()
            self.tokens = word_tokenize(text, language)
            self.language = language

    def remove_stopwords(self):
        self.tokens = [token for token in self.tokens if token not in stopwords.
            ↪ words(self.language)]

    def remove_short_token(self, min_len: int = 3):
        self.tokens = [token for token in self.tokens if len(token) >= min_len]

    def remove_numbers(self):
        self.tokens = [token for token in self.tokens if not re.findall('[0-9]',
            ↪ token)]

    def remove_interpunctuation(self):
        self.tokens = [token for token in self.tokens if token not in string.
            ↪ punctuation]

    def save(self, output_file: str = 'clean_token.txt'):
        token_str = '\n'.join(self.tokens)
        with open(output_file, 'w', encoding='utf-8') as output:
            output.write(token_str)

# Um die Token-Pipeline zu beginnen, geben wir den Dateinamen unseres Textes an
  ↪ :
token_pipeline = TokenPipeline(input_file='Daten/1774-Werther.txt', language='
  ↪ german')

# Jetzt können wir alle Bereinigungsschritte beliebig hinzufügen oder weglassen
token_pipeline.remove_numbers()
token_pipeline.remove_interpunctuation()
token_pipeline.remove_stopwords()
token_pipeline.remove_short_token(min_len=4)

# Als letztes sichern wir die bereinigte Tokenliste und wählen dafür einen
  ↪ Dateinamen aus:
token_pipeline.save(output_file='Daten/clean_token.txt')

# Wir können uns das Ergebnis aber zur Überprüfung zusätzlich hier ausgeben
  ↪ lassen:
print(token_pipeline.tokens)

```

Satz Pipeline

```

from nltk.tokenize import sent_tokenize, word_tokenize

class SentencePipeline:
    def __init__(self, input_file: str, language: str = 'german'):
        with open(input_file, 'r', encoding='utf-8') as input:
            text = input.read()
            self.sentences = sent_tokenize(text, language)
            self.language = language

```

```

def remove_short_sentences(self, min_len: int = 3):
    self.sentences = [sentence for sentence in self.sentences
                       if len(word_tokenize(sentence, self.language)) > min_len]

# hier können weitere Bereinigungsfunktionen ergänzt werden.

def save_as_list(self, output_file: str = 'sentence_list.txt'):
    sentence_str = '\n'.join(self.sentences)
    with open(output_file, 'w', encoding='utf-8') as output:
        output.write(sentence_str)

def save_as_text(self, output_file: str = 'sentence_list.txt'):
    sentence_str = ' '.join(self.sentences)
    with open(output_file, 'w', encoding='utf-8') as output:
        output.write(sentence_str)

# Um die Sentence-Pipeline zu beginnen, geben wir den Dateinamen unseres Textes
  ↪ an:
sentence_pipeline = SentencePipeline(input_file='Daten/1774-Werther.txt',
  ↪ language='german')

# Jetzt können wir alle Bereinigungsschritte beliebig hinzufügen oder weglassen
  ↪ :
sentence_pipeline.remove_short_sentences()

# Als letztes sichern wir die bereinigte Tokenliste und wählen dafür einen
  ↪ Dateinamen aus:
token_pipeline.save(output_file='Daten/clean_sentences.txt')

# Wir können uns das Ergebnis aber zur Überprüfung zusätzlich hier ausgeben
  ↪ lassen:
print(sentence_pipeline.sentences)

```

Externe und weiterführende Links

- Anaconda: <https://web.archive.org/save/https://www.anaconda.com/products/individual> (Letzter Zugriff: 04.06.2024)
- forTEXT.net-GitHub-Repository: <https://web.archive.org/save/https://github.com/forTEXT/forTEXT.net> (Letzter Zugriff: 04.06.2024)
- REGEX: <https://web.archive.org/save/https://regexr.com/> (Letzter Zugriff: 04.06.2024)

Bibliographie

- Anaconda Software Distribution. 2020. Anaconda Documentation. <https://docs.anaconda.com>.
- Bird, Steven, Ewan Klein und Edward Loper. 2009. *Natural language processing with Python: analyzing text with the natural language toolkit*. O'Reilly Media.
- Horstmann, Jan. 2024. Methodenbeitrag: Topic Modeling. Hg. von Evelyn Gius. *forTEXT* 1, Nr. 8. Topic Modeling (7. Oktober). doi: 10.48694/fortext.3717, <https://fortext.net/routinen/methoden/topic-modeling>.
- Kluyver, Thomas, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, u. a. 2016. Jupyter Notebooks – a publishing format for reproducible computational workflows. In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, hg. von F. Loizides und B. Schmidt, 87–90. IOS Press.
- Schumacher, Mareike. 2024a. Lehrmodul: Named Entity Recognition mit Stanford NER lehren. Hg. von Evelyn Gius. *forTEXT* 1, Nr. 9. Named Entity Recognition (30. Oktober). doi: 10.48694/fortext.3768, <https://fortext.net/routinen/lehrrmodule/named-entity-recognition-mit-stanford-ner-lehren>.
- . 2024c. Methodenbeitrag: word2vec. Hg. von Evelyn Gius. *forTEXT* 1, Nr. 10. word2vec (30. Oktober). doi: 10.48694/fortext.3815, <https://fortext.net/routinen/methoden/word2vec-1>.
- . 2024b. Toolbeitrag: Stanford Named Entity Recognizer. Hg. von Evelyn Gius. *forTEXT* 1, Nr. 9. Named Entity Recognition (30. Oktober). doi: 10.48694/fortext.3767, <https://fortext.net/tools/tools/stanford-named-entity-recognizer>.

Glossar

- CSV** CSV ist die englische Abkürzung für *Comma Separated Values*. Es handelt sich um ein Dateiformat zur einheitlichen Darstellung und Speicherung von einfach strukturierten Daten mit dem Kürzel `.csv`, sodass diese problemlos zwischen IT-Systemen ausgetauscht werden können. Dabei sind alle Daten zeilenweise angeordnet. Alle Zeilen wiederum sind in einzelne Datenfelder aufgeteilt, welche durch Trennzeichen wie Semikola oder Kommata getrennt werden können. In Programmen wie Excel können solche Textdateien als Tabelle angezeigt werden.
- Lemmatisieren** Die Lemmatisierung von Textdaten gehört zu den wichtigen **Preprocessing**-Schritten in der Textverarbeitung. Dabei werden alle Wörter (**Token**) eines Textes auf ihre Grundform zurückgeführt. So werden beispielsweise Flexionsformen wie „schneller“ und „schnelle“ dem Lemma „schnell“ zugeordnet.
- OCR** OCR steht für *Optical Character Recognition* und bezeichnet die automatische Texterkennung von gedruckten Texten, d.h. ein Computer „liest“ ein eingescanntes Dokument, erkennt und erfasst den Text darin und generiert daraufhin eine elektronische Version.
- Preprocessing** Für viele digitale Methoden müssen die zu analysierenden Texte vorab „bereinigt“ oder „vorbereitet“ werden. Für statistische Zwecke werden Texte bspw. häufig in gleich große Segmente unterteilt (*chunking*), Großbuchstaben werden in Kleinbuchstaben verwandelt oder Wörter werden **lemmatisiert**.
- Reintext-Version** Die Reintext-Version ist die Version eines digitalen Textes oder einer Tabelle, in der keinerlei Formatierungen (Kursivierung, Metadatenauszeichnung etc.) enthalten sind. Reintext-Formate sind beispielsweise TXT, RTF und **CSV**.
- Stoppwortliste** Stoppwörter sind hochfrequente Wörter, meist Funktionswörter, die, aufgrund ihrer grammatisch bedingten Häufigkeit, beispielsweise die Ergebnisse von inhaltlichen oder thematischen Analysen verzerren können. Deshalb werden diese Wörter, gesammelt in einer Stoppwortliste, bei digitalen Textanalysen meist nicht berücksichtigt.
- Type/Token** Das Begriffspaar „Type/Token“ wird grundsätzlich zur Unterscheidung von einzelnen Vorkommnissen (Token) und Typen (Types) von Wörtern oder Äußerungen in Texten genutzt. Ein Token ist also ein konkretes Exemplar eines bestimmten Typs, während ein Typ eine im Prinzip unbegrenzte Menge von Exemplaren (Token) umfasst.
- Es gibt allerdings etwas divergierende Definitionen zur Type-Token-Unterscheidung. Eine präzise Definition ist daher immer erstrebenswert. Der Satz „Ein Bär ist ein Bär.“ beinhaltet beispielsweise fünf Worttoken („Ein“, „Bär“, „ist“, „ein“, „Bär“) und drei Types, nämlich: „ein“, „Bär“, „ist“. Allerdings könnten auch vier Types, „Ein“, „ein“, „Bär“ und „ist“, als solche identifiziert werden, wenn Großbuchstaben beachtet werden.