

# 1 NLP

---

## Einführung

Linguistische Datenverarbeitung (Natural Language Processing, NLP) ist ein Teilgebiet der Linguistik und Informatik, das sich vor allem mit der Verarbeitung natürlichsprachlicher Daten, wie Text- oder Sprachinhalten, mithilfe probabilistischer Methoden des maschinellen Lernens beschäftigt. Da Computer nicht in der Lage waren, den Kontext der menschlichen Sprache zu verstehen, bestand ein Bedarf an Methoden, die diese Funktionalität unterstützen und es Computern dadurch ermöglichen, die menschliche Sprache zu verstehen und zu analysieren. Algorithmen für maschinelles Lernen und neuronale Netze wurden verwendet, um ein besseres Verständnis der Korpora und Texte in natürlichen Sprachen zu erlangen.

NLP-Tools sind wichtig für Unternehmen, die mit großen Mengen unstrukturierter Dokumente und Textdateien arbeiten, wie z. B. E-Mails, Sprach- oder Textkonversationen, Online-Chats, Umfragebögen usw.

Die Anwendung von NLP-Methoden in der Wirtschaft ermöglicht es Unternehmen, Daten effizient zu analysieren, relevante Informationen aus verschiedenen Texten und Dokumenten abzuleiten und wichtige Geschäftsentscheidungen zu treffen.

### Wie wird linguistische Datenverarbeitung in Unternehmen eingesetzt?

Wir haben vielleicht einmal den Satz gehört: „Dieser Anruf kann zu Qualitäts- und Schulungszwecken aufgezeichnet werden“, wenn wir anrufen und um einen Service beim Kundendienstcenter bitten. Die durch aufgezeichnete Anrufe oder Überarbeitungen eines bestimmten Produkts gesammelten Daten können für Algorithmen zur linguistischen Datenverarbeitung verwendet werden. Im Geschäftsleben können die Daten einen besseren Einblick in wichtige Informationen ermöglichen und Unternehmen in neue Bereiche führen. Da die meisten der gesammelten Daten unstrukturiert sind, erfordert die Verwendung standardmäßiger Datenanalysetools und -algorithmen die Umwandlung der Daten in stark strukturierte Formate, was zum Verlust wichtiger Informationen führen kann. In vielen Geschäftsbereichen hat die Technologie der linguistischen Datenverarbeitung aufgrund ihrer vielfältigen Anwendungen und der Fähigkeit, den Kontext unstrukturierter Daten zu verstehen, die online und in Anrufaufzeichnungen verfügbar sind, besondere Aufmerksamkeit erlangt.

## Spracherkennung

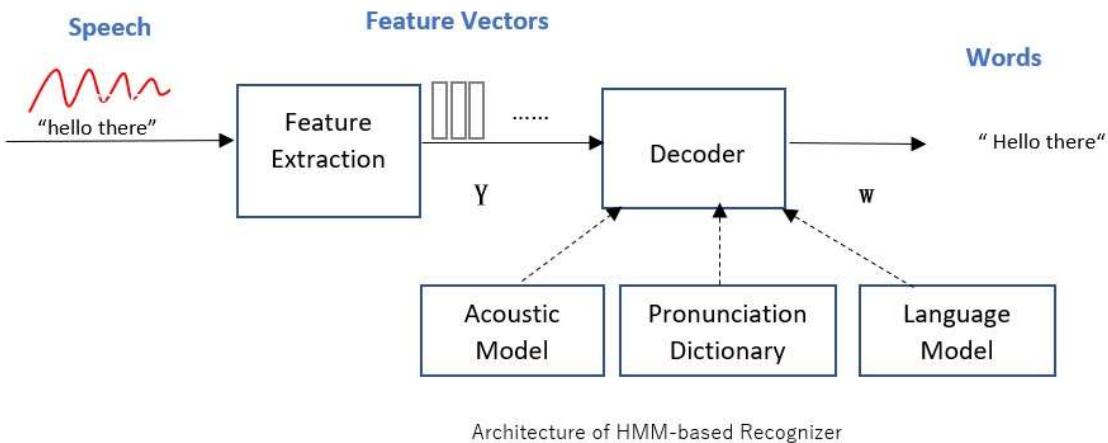
Die Spracherkennungstechnologie ermöglicht das Verstehen von Reden und deren anschließende Übersetzung in lesbaren Text mithilfe von Techniken des maschinellen Lernens und der Verarbeitung natürlicher Sprache.

Der Hauptvorteil der Spracherkennung kann in der Transkription gezeigt werden, dennoch kann sie eine Vielzahl von Anwendungsfällen ansprechen. Der transkribierte Text kann beispielsweise für die sprachbasierte Suche oder sprachbasierte Systeme wie virtuelle Assistenten und intelligente Haushaltsgeräte verwendet werden. ein weiterer Vorteil von Spracherkennung ist ihre Fähigkeit, beträchtliche Daten für Analyse- und Schüraufgaben zu produzieren.

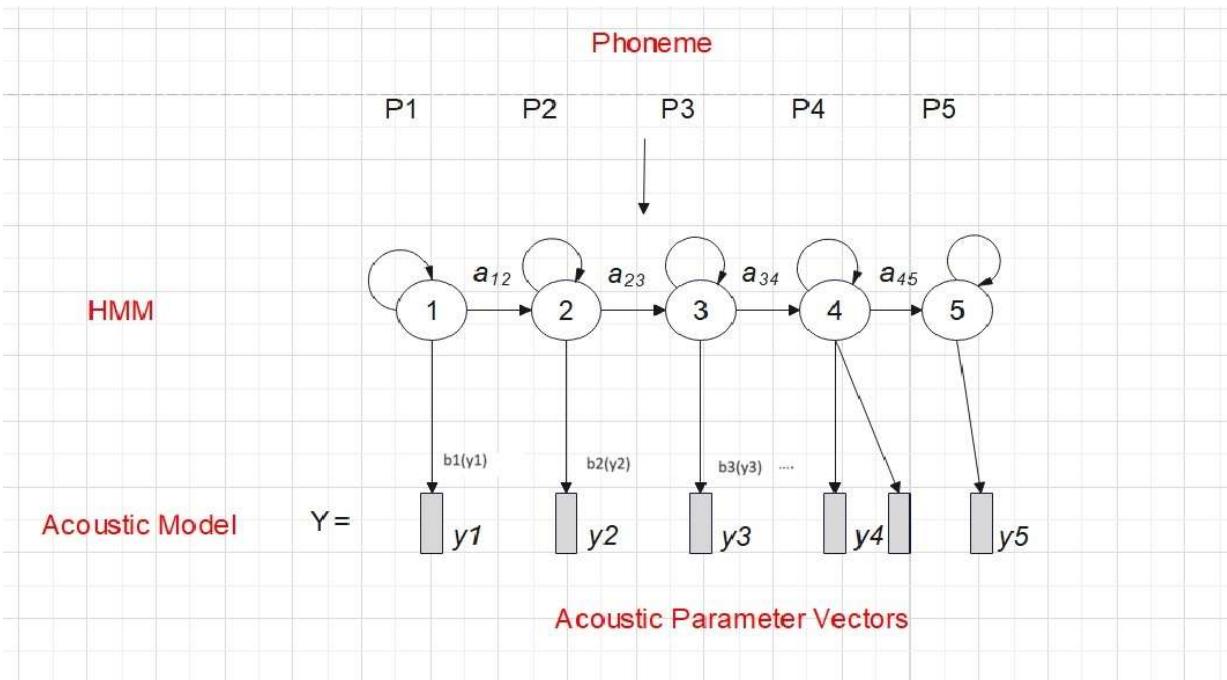
Wie funktioniert die Spracherkennung?

Die Sprache muss von einem physikalischen in ein elektrisches Signal und dann in ein digitales (digitalisiertes) Signal umgewandelt werden. Die meisten modernen Spracherkennungssysteme beruhen auf einem sogenannten Hidden-Markov-Modell (HMM). In einem typischen HMM wird das Sprachsignal in 10-Millisekunden-Fragmente aufgeteilt. Das Leistungsspektrum jedes Fragments, das im Wesentlichen eine Darstellung der Signalleistung als Funktion der Frequenz ist, wird auf einen Vektor aus reellen Zahlen abgebildet, die als Cepstral-Koeffizienten bekannt sind. Die endgültige Ausgabe des HMM ist eine Folge dieser Vektoren. Die Spracherkennung funktioniert durch die folgenden Schritte:

1. Die Eingangsaudiowellenform von einem Mikrofon wird in einem als Merkmalsextraktion bezeichneten Prozess in eine Folge akustischer Vektoren fester Größe umgewandelt.
2. Um Sprache in Text zu decodieren, werden Gruppen der ausgegebenen Vektoren in ein oder mehrere Phoneme (Grundeinheit der Sprache) eingeordnet. Der Decoder versucht dann, die Wortfolge  $w_1:L = w_1, \dots, w_L$  zu finden, die am wahrscheinlichsten Y erzeugt hat, indem er ein akustisches Modell, ein Aussprachewörterbuch und ein Sprachmodell verwendet. Der Decoder arbeitet unter Verwendung eines akustischen Modells, indem er alle möglichen Wortsequenzen durchsucht, indem er eine Beschneidung verwendet, um unwahrscheinliche Hypothesen zu entfernen.
3. Diese Berechnung erfordert Übung, da der Klang eines Phonems von Sprecher zu Sprecher unterschiedlich ist.
4. Ein spezieller Algorithmus wird dann verwendet, um das wahrscheinlichste Wort/Wörter zu bestimmen, die die gegebene Sequenz von Phonemen erzeugen.
5. Neuronale Netze werden verwendet, um das Sprachsignal unter Verwendung von Techniken zur Merkmalstransformation und Dimensionsreduktion vor der HMM-Erkennung zu vereinfachen.
6. Sprachaktivitätsdetektoren (VADs) werden auch verwendet, um ein Audiosignal nur auf die Teile zu reduzieren, die wahrscheinlich Sprache enthalten.



Für jedes gegebene  $w$  wird das entsprechende akustische Modell synthetisiert, indem Lautmodelle verkettet werden, um Wörter zu bilden, wie sie durch ein Aussprachewörterbuch definiert sind. Die Parameter dieser Lautmodelle werden aus Trainingsdaten geschätzt, die aus Sprachwellenformen und ihren orthografischen Transkriptionen bestehen. Das Sprachmodell ist typischerweise ein N-Gramm-Modell, bei dem die Wahrscheinlichkeit jedes Wortes nur von seinen  $N-1$  Vorgängern abhängt.



Diese Abbildung zeigt den Zustandsübergang in einem Hidden-Markov-Modell. Die Wahrscheinlichkeit, einen bestimmten Übergang vom Zustand  $s_i$  zu den Zustand  $s_j$  zu machen, ist durch die Übergangswahrscheinlichkeit  $\{a_{ij}\}$  gegeben. Beim Eintritt in einen Zustand wird ein Merkmalsvektor unter Verwendung der Wahrscheinlichkeit  $\{b_j(y_j)\}$  erzeugt, die dem Zustand zugeordnet ist, in den eingetreten wird.  $\{b_j(y_j)\}$  stellt die Wahrscheinlichkeit dar, dass das Ergebnis  $y_j$  ist.

## Spracherkennung Übung

Um mit Sprachdaten umgehen zu können, müssen wir das entsprechende Python-Modul installieren. Das Spracherkennungsmodul ist eine Python-Bibliothek zur Durchführung der Spracherkennung mit Unterstützung für mehrere Engines und APIs, online und offline. Der folgende Code importiert Daten aus einer Audiodatei "harvard.wav", verarbeitet und transkribiert die Daten in einen lesbaren Text.

Die Datei ist [hier](#) verfügbar.

```
In [1]: #pip install SpeechRecognition
import speech_recognition as sr
# get the version
sr.__version__
r = sr.Recognizer()
har = sr.AudioFile('harvard.wav')
with har as s:
    aud = r.record(s)

r.recognize_google(aud)

result2:
{   'alternative': [   {   'confidence': 0.94823617,
        'transcript': 'the still smell of old beer lingers '
                      'it takes heat to bring out the odour '
                      'a cold dip restores health exist a '
                      'salt pickle taste fine with him as '
                      'well past or my favourite exist for '
                      'food is the hot cross bun'},
        {   'transcript': 'the still smell of old beer lingers '
                      'it takes heat to bring out the odour '
                      'a cold dip restores health exist a '
                      'salt pickle taste fine with him go '
                      'past or my favourite exist for food '
                      'is the hot cross bun'},
        {   'transcript': 'the still smell of old beer lingers '
                      'it takes heat to bring out the odour '
```

Die record Funktion wird verwendet, um Audiodaten aus der Datei zu extrahieren. Duration- und Offset-Argumente werden verwendet, um die Aufzeichnung und Abschaltung (in Sekunden) des Prozesses zu steuern. Diese Werte können die Qualität der Ausgabe beeinflussen und müssen daher fein abgestimmt werden.

```
In [2]: ## working with audio files
har = sr.AudioFile('harvard.wav')

with har as s:
    # extract audio data from the file
    # Records up to duration seconds of audio from source (an AudioSource instance) starting at offset
    # (or at the beginning if not specified) into an AudioData instance, which it returns.
    aud = r.record(s, offset = 4.7, duration =2.2 )
    aud2= r.record(s, offset = 4, duration =5 )
r.recognize_google(aud) # # generate a list of possible transcriptions

result2:
{   'alternative': [   {   'confidence': 0.92281985,
        'transcript': 'exceed to bring out the odour'},
        {'transcript': 'exceed to bring out the owner'},
        {'transcript': 'exceed to bring out the ogre'},
        {'transcript': 'exceed to bring out the ODA'},
        {'transcript': 'make seed to bring out the odour'}],
```

Eine weitere Funktion des Spracherkennungsmoduls ist adjust\_for\_ambient\_noise(). Diese Funktion nimmt die notwendigen Änderungen an den Einstellungen vor, damit die Sprache in einer leicht lauten Umgebung gehört werden kann.

Im folgenden Code extrahieren wir Audiodaten aus einer verrauschten Audiodatei [jackhammer.wav](#):

```
In [3]: jackhammer = sr.AudioFile('jackhammer.wav')
with jackhammer as jh:
    # adjust_for_ambient_noise:this function makes the necessary changes
    # to the settings that allow the speech to be heard in a slightly noisy environment.
    r.adjust_for_ambient_noise(jh, duration = 1)
    aud = r.record(jh)
r.recognize_google(aud, show_all = True)

Out[3]: {'alternative': [['transcript': 'spell smell during windows',
    'confidence': 0.60805857},
    {'transcript': 'spell smell off your fingers'},
    {'transcript': 'still smell up your windows'},
    {'transcript': 'still smell your fingers'},
    {'transcript': 'spell smell']],
'final': True}
```

Mit dem folgenden Code verwenden wir Audioeingangsdaten mithilfe der Mikrofonfunktion, um ein Ratespiel zu programmieren, bei dem der Benutzer das Wort erraten soll, das das Programm zufällig ausgewählt hat. Der Benutzer sollte das Wort laut sprechen, das anschließend erfasst, erkannt, es mit dem ausgewählten verglichen wird. Schließlich wird das Ergebnis an den Benutzer zurückgeliefert. Der Benutzer sollte das Wort in maximal 3 Versuchen erraten, um das Spiel zu gewinnen.

```

r = sr.Recognizer()
response = {"success": True, "error": None, "transcription": None}
wait_for_answer = 5
words = ['milk', 'butter', 'cheese', 'jam', 'bread']
word = random.choice(words)
with sr.Microphone() as source:
    print("guess the word am thinking of from the following list:\n" "{wrds}\n" "You have {n} tries.\n".format(wrds= words, n=num_att))
    for i in range(num_att):
        #r.adjust_for_ambient_noise(source, duration = 1)
        for j in range(wait_for_answer):
            audio = r.listen(source, timeout=5, phrase_time_limit=5)

        try:
            response['transcription'] = r.recognize_google(audio)
        except sr.UnknownValueError:
            print("Could not understand audio")
        except sr.RequestError as e:
            print("Could not request results; {0}".format(e))
        if response['transcription']:
            break
        print('I did not catch what you said: please repeat again')
    response['success'] = response["transcription"].lower() == word.lower()
if response['success']:
    print('you are correct!, the word is {}, Congratulations!\n'.format(word))
    break
elif ( i < num_guess - 1):
    print('Incorrect . please try again . you still have {} attempts\n'.format(num_guess-i))
else:
    print("the word is: {}, sorry! you did not guess the right answer, hard luck!\n".format(word))
break

```

```

1 {
2   "cells": [
3     {
4       "cell_type": "code",
5       "execution_count": 1,
6       "id": "87c85eb5",
7       "metadata": {},
8       "outputs": [
9         {
10           "name": "stdout",
11           "output_type": "stream",
12           "text": [
13             "result2:\n",
14             "{'alternative': [ { 'confidence': 0.94823629,\n",
15               "transcript": 'the still smell of old beer lingers '\n",
16               "it takes heat to bring out the odour '\n",
17               "a cold dip restores health exist a '\n",
18               "salt pickle taste fine with him as '\n",
19               "well past or my favourite exist for '\n",
20               "food is the hot cross bun'},\n",
21               { 'transcript': 'the still smell of old beer lingers '\n",
22                 "it takes heat to bring out the odour '\n",
23                 "a cold dip restores health exist a '\n",
24                 "salt pickle taste fine with him go '\n",
25                 "past or my favourite exist for food '\n",
26                 "is the hot cross bun'},\n",
27               { 'transcript': 'the still smell of old beer lingers '\n",
28                 "it takes heat to bring out the odour '\n",
29                 "a cold dip restores health exist a '\n",
30                 "salt pickle taste fine with him past '\n",
31                 "or my favourite exist for food is '\n",
32                 "the hot cross bun'},\n",
33               { 'transcript': 'the still smell of old beer lingers '\n",
34                 "it takes heat to bring out the older '\n",
35                 "a cold dip restores health exist a '\n",
36                 "salt pickle taste fine with him go '\n",
37                 "past or my favourite exist for food '\n",
38                 "is the hot cross bun'},\n",
39               { 'transcript': 'the still smell of old beer Lingus '\n",
40                 "it takes heat to bring out the odour '\n",
41                 "a cold dip restores health exist a '\n",
42                 "salt pickle taste fine with him as '\n",
43                 "well past or my favourite exist for '\n",
44                 "food is the hot cross bun'}],\n",
45           "final": True}\n"
46         ]
47     },
48     {
49       "data": {
50         "text/plain": [
51           "'the still smell of old beer lingers it takes heat to bring out the odour a cold dip restores health exist a salt pi"
52         ]
53       },
54       "execution_count": 1,
55       "metadata": {},
56       "output_type": "execute_result"
57     },
58   ],
59   "source": [
60     "#pip install SpeechRecognition\n",
61     "import speech_recognition as sr\n",
62     "# get the version\n",
63     "sr.__version__\n",
64     "r =sr.Recognizer()\n",
65     "har = sr.AudioFile('harvard.wav')\n",
66     "with har as s:\n",
67     "    aud = r.record(s)\n",
68     "    \n",
69     "r.recognize_google(aud)      "
70   ],
71 },
72 {
73   "cell_type": "code",

```

```

74     "execution_count": 2,
75     "id": "543f0e2c",
76     "metadata": {},
77     "outputs": [
78     {
79         "name": "stdout",
80         "output_type": "stream",
81         "text": [
82             "result2:\n",
83             "{ 'alternative': [ { 'confidence': 0.9228186, \n",
84             "transcript": 'exceed to bring out the odour'}, \n",
85             " { 'transcript': 'exceed to bring out the owner'}, \n",
86             " { 'transcript': 'exceed to bring out the ogre'}, \n",
87             " { 'transcript': 'exceed to bring out the ODA'}, \n",
88             " { 'transcript': 'make seed to bring out the odour'}], \n",
89             "final": True}\n"
90         ],
91     },
92     {
93         "data": {
94             "text/plain": [
95                 "'exceed to bring out the odour'"
96             ]
97         },
98         "execution_count": 2,
99         "metadata": {},
100        "output_type": "execute_result"
101    },
102 ],
103 "source": [
104     "## working with audio files\n",
105     "har = sr.AudioFile('harvard.wav')\n",
106     "\n",
107     "with har as s:\n",
108         # extract audio data from the file\n",
109         # Records up to duration seconds of audio from source (an AudioSource instance) starting at offset \n",
110         # (or at the beginning if not specified) into an AudioData instance, which it returns.\n",
111         aud = r.record(s, offset = 4.7, duration = 2.2) \n",
112         aud2= r.record(s, offset = 4, duration = 5)\n",
113         "r.recognize_google(aud) # # generate a list of possible transcriptions"
114     ],
115 ],
116 {
117     "cell_type": "code",
118     "execution_count": 31,
119     "id": "1ced8ae1",
120     "metadata": {},
121     "outputs": [],
122     "source": [
123         "jackhammer = sr.AudioFile('jackhammer.wav')\n",
124         "with jackhammer as jh:\n",
125             # adjust_for_ambient_noise:this function makes the necessary changes \n",
126             # to the settings that allow the speech to be heard in a slightly noisy environment.\n",
127             r.adjust_for_ambient_noise(jh, duration = 1)\n",
128             aud = r.record(jh)"
129     ],
130 ],
131 {
132     "cell_type": "code",
133     "execution_count": 17,
134     "id": "8c8e6ba3",
135     "metadata": {},
136     "outputs": [
137     {
138         "data": {
139             "text/plain": [
140                 "[]"
141             ]
142         },
143         "execution_count": 17,
144         "metadata": {},
145         "output_type": "execute_result"
146     },
147 ],
148 "source": [
149     "r.recognize_google(aud, show_all = True)"
150 ],
151 ],
152 {
153     "cell_type": "code",
154     "execution_count": null,
155     "id": "fdd0b308",
156     "metadata": {},
157     "outputs": [],
158     "source": [
159         "## working with microphones\n",
160         "#pip install pyaudio\n",
161         "mic = sr.Microphone()\n",
162         "with mic as mc:\n",
163             r.adjust_for_ambient_noise(mc)\n",
164             aud = r.listen(source=mic, timeout=10, phrase_time_limit=3)\n",
165             "r.recognize_google(aud)"
166     ],
167 ],
168 {
169     "cell_type": "code",
170     "execution_count": 93,
171     "id": "99a668cb",
172     "metadata": {},
173     "outputs": [
174     {
175         "name": "stdout",
176         "output_type": "stream",
177         "text": [

```

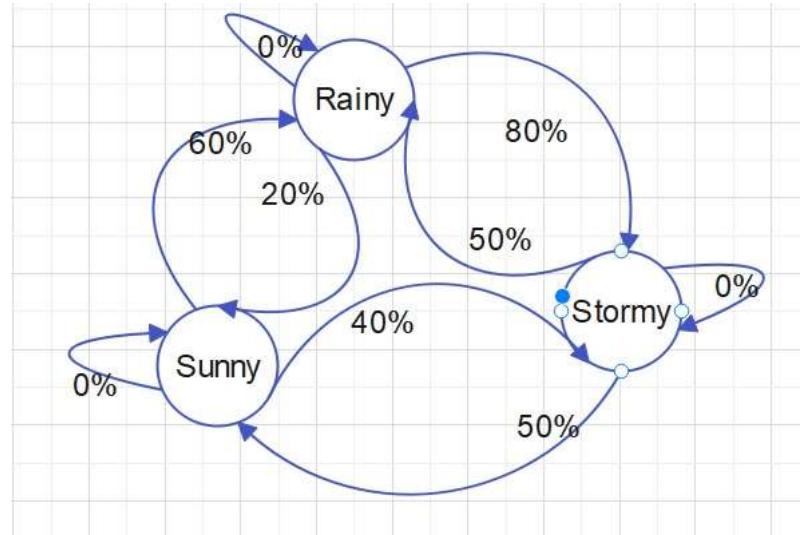
```

178     "guess the word am thinking of from the following list:\n",
179     "['milk', 'butter', 'cheese', 'jam', 'bread']\n",
180     "You have 3 tries.\n",
181     "\n"
182   ],
183 },
184 {
185   "ename": "KeyboardInterrupt",
186   "evalue": "",
187   "output_type": "error",
188   "traceback": [
189     "\u001b[1;31m-----\u001b[0m",
190     "\u001b[1;31mKeyboardInterrupt\u001b[0m"                                     Traceback (most recent call last),
191     "Cell \u001b[1;32mIn[93], line 17\u001b[0m\n\u001b[0;32m      14\u001b[0m \u001b[38;5;28;01mfor\u001b[39;00m i \u001b[3
192     "File \u001b[1;32m~\\anaconda3\\lib\\site-packages\\speech_recognition\\__init__.py:709\u001b[0m, in \u001b[0;36mRecog
193     "File \u001b[1;32m~\\anaconda3\\lib\\site-packages\\speech_recognition\\__init__.py:211\u001b[0m, in \u001b[0;36mMicro
194     "File \u001b[1;32m~\\anaconda3\\lib\\site-packages\\pyaudio.py:612\u001b[0m, in \u001b[0;36mStream.read\u001b[1;34m(se
195     "\u001b[1;31mKeyboardInterrupt\u001b[0m: "
196   ]
197 }
198 ],
199 "source": [
200   "import random\n",
201   "import time\n",
202   "import speech_recognition as sr  \n",
203   "# get audio from the microphone\n",
204   "r = sr.Recognizer() \n",
205   "response = {"success": True, \"error\": None, \"transcription\": None}\n",
206   "wait_for_answer =5\n",
207   "words = ['milk', 'butter', 'cheese', 'jam', 'bread']  \n",
208   "word = random.choice(words)\n",
209   "with sr.Microphone() as source: \n",
210   "    print(\"guess the word am thinking of from the following list:\\n\" \"\{wrds\\}\\n\\\" \"You have {n} tries.\\n\\\".format
211   "    for i in range(num_att):\n",
212   "        #r.adjust_for_ambient_noise(source, duration = 1)\n",
213   "        for j in range(wait_for_answer): \n",
214   "            audio = r.listen(source, timeout=5, phrase_time_limit=5)\n",
215   "            \n",
216   "            try:\n",
217   "                response['transcription'] = r.recognize_google(audio)\n",
218   "            except sr.UnknownValueError:\n",
219   "                print(\"Could not understand audio\")\n",
220   "            except sr.RequestError as e:\n",
221   "                print(\"Could not request results; {0}\").format(e)\n",
222   "            if response['transcription']:\n",
223   "                break\n",
224   "            print('I did not catch what you said: please repeat again')      \n",
225   "            response['success'] = response[\"transcription\"].lower() == word.lower()\n",
226   "            if response['success']:\n",
227   "                print('you are correct!, the word is {}, Congratulations!\\n'.format(word))\n",
228   "                break\n",
229   "            elif ( i < num_guess - 1):\n",
230   "                print('Incorrect . please try again . you still have {} attempts\\n'.format(num_guess-1-i) )\n",
231   "            else:\n",
232   "                print(\"the word is: {}, sorry! you did not guess the right answer, hard luck!\\n\").format(word))  \n",
233   "                break
234   "
235 },
236 {
237   "cell_type": "code",
238   "execution_count": null,
239   "id": "28c47a8e",
240   "metadata": {},
241   "outputs": [],
242   "source": []
243 },
244 {
245   "cell_type": "code",
246   "execution_count": null,
247   "id": "faf7c861",
248   "metadata": {},
249   "outputs": [],
250   "source": []
251 },
252 ],
253 "metadata": {
254   "kernelspec": {
255     "display_name": "Python 3 (ipykernel)",
256     "language": "python",
257     "name": "python3"
258   },
259   "language_info": {
260     "codemirror_mode": {
261       "name": "ipython",
262       "version": 3
263     },
264     "file_extension": ".py",
265     "mimetype": "text/x-python",
266     "name": "python",
267     "nbconvert_exporter": "python",
268     "pygments_lexer": "ipython3",
269     "version": "3.9.13"
270   },
271 },
272 "nbformat": 4,
273 "nbformat_minor": 5
274 }

```

Ein Markov-Prozess ist ein Prozess, der Übergänge zwischen verschiedenen Zuständen durchführen kann, wobei die Wahrscheinlichkeiten der verfügbaren Zustände und Übergänge nur vom aktuellen Zustand des Systems abhängen. Mit anderen Worten: Ein Markov-Prozess hat kein Gedächtnis.

Markov-Ketten, benannt nach Andrey Markov, sind ein mathematisches Modell eines stochastischen Prozesses, der sich je nach Zustand des aktuellen Zustands von einem „Zustand“ in einen anderen bewegt. Beispielsweise wird das Markov-Ketten-Modell verwendet, um den Zustand des nächsten Zustands vorherzusagen (z. B. wird es morgen regnen?) und die Wahrscheinlichkeit des Übergangs vom aktuellen Zustand zum nächsten zu modellieren. Im NLP werden Markov-Ketten bei der Textgenerierung verwendet, um das nächste Wort in einem Text basierend auf dem aktuell eingegebenen Wort vorherzusagen.



Markov-Ketten sind sehr effektiv bei der Modellierung eines Textes, indem sie jedem Wort unterschiedliche Wahrscheinlichkeiten dafür zuordnen, wie wahrscheinlich die anderen Wörter im Text darauf folgen. Daher wurden Markov-Ketten zur Textgenerierung verwendet, indem das nächste Wort in einem Text basierend auf dem aktuell eingegebenen Wort vorhergesagt wird. In einem bestimmten Korpus können Wörter mithilfe von Markov-Ketten mit unterschiedlichen Wahrscheinlichkeiten „verknüpft“ werden. Wenn das aktuelle Wort (der aktuelle Zustand) beispielsweise „Regen“ ist, verketten Markov Modellwahrscheinlichkeiten für alle Wörter im Text und zeigt an, wie wahrscheinlich es ist, dass jedes Wort dem aktuellen folgt. Dementsprechend könnten wir einen „Gewitter“ von 65 % haben, der wahrscheinlich auf „Regen“ folgt, zusammen mit einem „tatsächlichen“ Wert von 20 %, einem „möglicherweise“ von 10 % und so weiter. Dabei wird davon ausgegangen, dass die Wahrscheinlichkeit, dass jedes Wort sich selbst folgt, etwa 0 % betragen sollte, da dies in natürlichsprachlichen Skripten nicht der Fall wäre.

## Markov-Chains Übung

Wir werden drei von Shakespeares Tragödien aus dem NLTK-Korpus des Projekts Gutenberg verwenden, um den Kettengenerator zu trainieren. Markovify ist ein Python-Modul, das zum Erstellen von Markov-Modellen großer Textkorpora und zum Generieren zufälliger Sätze daraus verwendet wird. Der erste Schritt besteht darin, die erforderlichen Pakete zu installieren.

**Anmerkung:** Der folgende Code sollte auf Windows-Betriebssystemen reibungslos funktionieren. Bitte achten Sie bei der Installation der Pakete darauf, die entsprechenden Befehle zu verwenden, falls Sie mit einem anderen Betriebssystem als Windows arbeiten.

```
In [2]: #markovify
!pip install nltk
!pip install markovify
!pip install spacy
!pip install -m spacy download en

Requirement already satisfied: nltk in c:\users\la2022\anaconda3\lib\site-packages (3.7)
Requirement already satisfied: joblib in c:\users\la2022\anaconda3\lib\site-packages (from nltk) (1.1.0)
Requirement already satisfied: click in c:\users\la2022\anaconda3\lib\site-packages (from nltk) (8.0.4)
Requirement already satisfied: tqdm in c:\users\la2022\anaconda3\lib\site-packages (from nltk) (4.64.1)
Requirement already satisfied: regex>=2021.8.3 in c:\users\la2022\anaconda3\lib\site-packages (from nltk) (2022.7.9)
Requirement already satisfied: colorama in c:\users\la2022\anaconda3\lib\site-packages (from click->nltk) (0.4.5)
```

Das Python-Modul nltk.corpus wird zum Herunterladen der Gutenberg-Schriften verwendet, während spacy.load als Wrapper zum Lesen der Pipeline mithilfe von language: 'en' verwendet wird, um ein Sprachobjekt zu erstellen. Schauen wir uns den Gutenberg-Korpus an:

```
In [2]: # importing the libraries
import spacy
#regular expression
import re
import markovify
import nltk
from nltk.corpus import gutenberg
import warnings
warnings.filterwarnings('ignore')
nltk.download('gutenberg')

#inspect Gutenberg corpus
print(gutenberg.fileids())

['austen-emma.txt', 'austen-persuasion.txt', 'austen-sense.txt', 'bible-kjv.txt', 'blake-poems.txt', 'bryant-stories.txt', 'burgess-busterbrown.txt', 'carroll-alice.txt', 'chesterton-ball.txt', 'chesterton-brown.txt', 'chesterton-thursday.txt', 'edgeworth-parents.txt', 'melville-moby_dick.txt', 'milton-paradise.txt', 'shakespeare-caesar.txt', 'shakespeare-hamlet.txt', 'shakespeare-macbeth.txt', 'whitman-leaves.txt']
```

Im nächsten Schritt müssen drei Shakespeares Tragödien importiert werden, um sie zum Trainieren des Generators zu verwenden: „Hamlet“, „Macbeth“ und „Caesar“.

```
In [ ]: #import the plays
macbeth = gutenberg.raw('shakespeare-macbeth.txt')
hamlet = gutenberg.raw('shakespeare-hamlet.txt')
caesar = gutenberg.raw('shakespeare-caesar.txt')

#print the first 100 char of each play to take a look
print('\nmacbeth:\n', hamlet[:250])
print('\ncaesar:\n', caesar[:250])
print('\nhamlet:\n', macbeth[:250])
```

Die Textbereinigung ist ein wichtiger Schritt, um unnötige Informationen wie Satzzeichen und Sonderzeichen aus dem Korpus zu entfernen. Dies funktioniert mit der Engine für reguläre Ausdrücke von Python, die eine Reihe von Zeichenfolgen in einem Text angibt, die einem bestimmten regulären Ausdruck entsprechen. Die Funktion re.sub() entfernt und ersetzt die Teile des Textes, wenn sie mit den angegebenen Zeichenfolgen übereinstimmen. Hier verwenden wir re.sub, um Satzzeichen, Symbole und Klammern zu entfernen, da diese für die Erstellung des Modells möglicherweise irrelevant sind. Kapitelmarkierungen müssen ebenfalls entfernt werden.

```
In [121]: # text cleaning; re.sub() replaces the occurrences of a string by the second argument(repl)
#The r means that the string is to be treated as a raw string, which means all escape codes will be ignored.e.g '\n' will NOT be
# treated as new line but as '\' followed by 'n'.
# '[' creates a regular expression that will match either A or B.
# '\b Matches the empty string, but only at the beginning or end of a word
# '\s Matches Unicode whitespace characters ..
# '\d Matches any Unicode decimal digits
# * Causes the resulting RE to match 0 or more repetitions of the preceding RE, e.g. ab* will match 'a', 'ab', or 'a' followed by
# + Causes the resulting RE to match 1 or more repetitions of the preceding RE, e.g. ab+ will match 'a' followed by any non-zero
# ? Causes the resulting RE to match 0 or 1 repetitions of the preceding RE, e.g. ab? will match either 'a' or 'ab'.
# \ Either escapes special characters (permitting you to match characters like '*', '?', and so forth).
# [] Used to indicate a set of characters. In a set: Characters can be listed individually, e.g. [amk] will match 'a', 'm', or 'k'
# OR Ranges of characters can be indicated by giving two characters and separating them by a '-'. e.g. [0-5][0-9] will match all
# and [0-9A-Fa-f] will match any hexadecimal digit.
# ^ Matches the start of the string
# white space characters: ' - Space. '\t' - Horizontal tab. '\v' - Vertical tab. '\n' - Newline. '\r' - Carriage return. '\f' - Feed
#text = ' cwww,-645e2wkk9875[?-*!lmmmm ^^ --\n\f\v'
#text = re.sub(r'[m+ -- \[*. \??* \d+ ^ \s]', '', text), >> cvewkk=\
def clean_txt(txt):
    #text = re.sub(r'[m+ -- \[*. \??* \d+ ^ \s]', '', txt)
    text = re.sub(r'--', '', txt)
    text = re.sub('[\[\].*?\[\]]', '', text)
    text = re.sub(r'(\b|\s+\-?|^-\?)(\d+|\d*\.\d+ )\b', '', text)
    return text
```

```
In [122]: #remove chapter indicators
hamlet = re.sub(r'Chapter \d+', '', hamlet)
macbeth = re.sub(r'Chapter \d+', '', macbeth)
caesar = re.sub(r'Chapter \d+', '', caesar)
#Cleaning the texts
hamlet = clean_txt(hamlet)
macbeth = clean_txt(macbeth)
caesar = clean_txt(caesar)
print(hamlet[:250])
```

spacy.load( ) wird verwendet, um den bereinigten Text zu lesen und die zum Erstellen des Generators erforderlichen Sprachsubjekte zu erstellen.

```
In [9]: # parsing the cleaned text
# spacy.load() is used as a wrapper to read the pipeline by means of Language: 'en' to construct language object
lang_obj = spacy.load('en_core_web_sm')
hamlet_obj = lang_obj(hamlet)
macbeth_obj = lang_obj(macbeth)
caesar_obj = lang_obj(caesar)
print(hamlet_obj)
```

Actus Primus. Scoena Prima.

Enter Barnardo and Francisco two Centinels.

Barnardo. Who's there?  
Fran. Nay answer me: Stand & vnfold  
your selfe

Bar. Long lieue the King

Fran. Barnardo?  
Bar. He

Ein Sprachobjekt besteht aus Wörtern, die zu Sätzen zusammengefasst werden müssen, die wiederum als Eingabe für den Generator verwendet werden müssen. Die drei Romane werden zusammengeführt und zum Aufbau des Generators verwendet.

```
In [10]: # combining the sentences in the documents, language object consists of 'word' elements, that's why hamlet_obj[:100] is longer than hamlet_sents[:100]
hamlet_sents = ' '.join([sent.text for sent in hamlet_obj.sents if len(sent.text) > 1])
macbeth_sents = ' '.join([sent.text for sent in macbeth_obj.sents if len(sent.text) > 1])
caesar_sents = ' '.join([sent.text for sent in caesar_obj.sents if len(sent.text) > 1])
# combination of three novels
HMC = hamlet_sents + macbeth_sents + caesar_sents
print(len(HMC))
```

376067

In diesem Schritt erstellen wir einen Textgenerator mit der Methode markovify.text(). Das Argument der Zustandsgröße ist die Anzahl der Wörter, von der die Wahrscheinlichkeit eines nächsten Wortes abhängt. Anschließend generieren wir mit dem Generator lange und kurze Sätze.

```
In [16]: #create text generator using markovify
# State size is a number of words the probability of a next word depends on.
# for text generation: we will build Markov model using three of Shakespeares' Tragedies from the Project Gutenberg NLTK corpus.
gen = markovify.Text(HMC, state_size=1)

#generating short and long sentences using make_sentance() and make_short_sentence()
print('long sentences: \n')
for i in range(4):
    print(gen.make_sentence())

print('\n short sentences: \n')
for i in range(4):
    # of max 100 chars
    print(gen.make_short_sentence(100))
```

```
long sentences:

Nay that's honest Ham.
Good night good words That open'd lies he is not, He speakes by and Night?
Ile set me in his eyes a-while, For I had beene there was it away?
There's another: you shall be reueng'd Most throughly for a poyson in Shallowes, and the manner Bru.

short sentences:

Oh wonderfull Sonne, do Malc.
To pricke vs haue no more: And as if he is Rome?
I, A rapsodie of my smooth Body.
I am sick at a time of Angels and her Closset, take vpon's what hoa?
```

Um die Textvorhersage zu verbessern, verwenden wir die Klasse POSifiedText. Dies kann entweder ein NLTK oder ein spaCy-Tagger sein, um ein Markov-Modell zu generieren, das besser zur Satzstruktur passt als ein naives Modell.

Beim POS-Tagging wird ein Wort im Text markiert auf eine bestimmte Wortart basierend auf ihrem Kontext und ihrer Definition beziehen. Mit anderen Worten: Beim POS-Tagging wird ein Wort als Substantiv, Pronomen, Verb, Adjektiv usw. identifiziert.

```
In [12]: # to improve the text prediction we will use POSifiedText class: spaCy tagger to generate a Markov model that comply with
# sentence structure better than a naive model.

#in spacy library POS tagging is the process of marking a word in the text
#to a particular part of speech based on both its context and definition.
#In simple language, we can say that POS tagging is the process of identifying a word as nouns,
#pronouns, verbs, adjectives, etc.

class POSifiedText(markovify.Text):
    def word_split(self, sentence):
        return [':'.join((word.orth_, word.pos_)) for word in lang_obj(sentence)] # add word tags(positions)
    def word_join(self, words):
        sentence = ' '.join(word.split(':')[0] for word in words) # re-build sentences for the model
        return sentence
generator_2 = POSifiedText(HMC, state_size=2)
generator_2
```

```
In [21]: print('short sentences: \n')
for i in range(4):
    print(generator_2.make_short_sentence(max_chars=100))
```

```
short sentences:

With what , did the Cyclops hammers fall
On all deseruers .
We shall be done ?
Enter the Ghost of Banquo he is young ,
And for thy Dowrie .
I Sir , and from henceforth
When greefe and blood ill temper'd too

Mar. Nor I my Lord ?
```

```
In [23]: print('\n long sentences: \n')
for i in range(4):
    print(generator_2.make_sentence())
```

```
long sentences:

Read it great Caesar

Caes Bid them prepare within
Remember that you are a Roman tell me that ,
But with a crafty Madnesse keepes aloofe
And that craves warie walking Crown him that , and by
On this side Tyber , chafing with her .
Must I remember now
Our feares do make vs Med'cines of our nature come
To addre the death of Hamlet our deere Brothers death
The memory be greene and though I
Truly deliuier

For .
Haue after , to his Birth
If that thou bee'st a Roman

Cassi .
Good morrow worthy Caesar , I can no more ,
And tell them that I am not in your teeth .
```

```
1 {
2     "cells": [
3         {
4             "cell_type": "code",
```

```

5   "execution_count": null,
6   "id": "6bca42be",
7   "metadata": {},
8   "outputs": [],
9   "source": [
10     "#markovify \n",
11     "!pip install nltk \n",
12     "!pip install markovify\n",
13     "!pip install spacy\n",
14     "#!pip install -m spacy download en\n",
15     "!python -m spacy download en"
16   ],
17 },
18 {
19   "cell_type": "code",
20   "execution_count": null,
21   "id": "6d76901a",
22   "metadata": {},
23   "outputs": [],
24   "source": [
25     "# importing the libraries\n",
26     "import spacy\n",
27     "#regular expression \n",
28     "import re\n",
29     "import markovify\n",
30     "import nltk\n",
31     "from nltk.corpus import gutenberg\n",
32     "import warnings\n",
33     "warnings.filterwarnings('ignore')\n",
34     "nltk.download('gutenberg')\n",
35     "\n",
36     "#inspect Gutenberg corpus\n",
37     "print(gutenberg.fileids())\n"
38   ],
39 },
40 {
41   "cell_type": "code",
42   "execution_count": null,
43   "id": "cc02d32",
44   "metadata": {},
45   "outputs": [],
46   "source": [
47     "#import the plays\n",
48     "macbeth = gutenberg.raw('shakespeare-macbeth.txt')\n",
49     "hamlet = gutenberg.raw('shakespeare-hamlet.txt')\n",
50     "caesar = gutenberg.raw('shakespeare-caesar.txt')\n",
51     "\n",
52     "#print the first 100 char of each play to take a look\n",
53     "print('\\nmacbeth:\\n', hamlet[:250])\n",
54     "print('\\ncaesar:\\n', caesar[:250])\n",
55     "print('\\nhamlet:\\n', macbeth[:250])"
56   ],
57 },
58 {
59   "cell_type": "code",
60   "execution_count": null,
61   "id": "77741bc5",
62   "metadata": {},
63   "outputs": [],
64   "source": [
65     "# text cleaning; re.sub() replaces the occurrences of a string by the second argument(repl)\n",
66     "#The r means that the string is to be treated as a raw string, which means all escape codes will be ignored.e.g '\\n' w.
67     "# treated as new line but as '\\\' followed by 'n'.\n",
68     "# '|' creates a regular expression that will match either A or B.\n",
69     "# '\\b Matches the empty string, but only at the beginning or end of a word\n",
70     "# '\\s Matches Unicode whitespace characters ..\n",
71     "# '\\d Matches any Unicode decimal digits.,\n",
72     "# * Causes the resulting RE to match 0 or more repetitions of the preceding RE, e.g. ab* will match ?a?, ?ab?, or ?a? f
73     "# + Causes the resulting RE to match 1 or more repetitions of the preceding RE, e.g. ab+ will match ?a? followed by any
74     "# ? Causes the resulting RE to match 0 or 1 repetitions of the preceding RE, e.g. ab? will match either ?a? or ?ab?.\n",
75     "# \\ Either escapes special characters (permitting you to match characters like '*', '?', and so forth).\n",
76     "# [] Used to indicate a set of characters. In a set: Characters can be listed individually, e.g. [amk] will match 'a',
77     "# OR Ranges of characters can be indicated by giving two characters and separating them by a '-'. e.g. [0-5][0-9] will :
78     "# and [0-9A-Fa-f] will match any hexadecimal digit.\n",
79     "# ^ Matches the start of the string \n",
80     "# white space characters: ' ? Space. '\\t' ? Horizontal tab.'\\v' ? Vertical tab.'\\n' ? Newline.'\\r' ? Carriage retu:
81     "#text = cvmm,-645e2wkk9875=[?=!\\"mnm ^ --\\n \\"f\\v'\n",
82     "#text = re.sub(r'[m+ -- \\[*. \\?** \\d+ \\^ \\s]', '', text), >> cvewkk==\\n",
83     "def clean_txt(txt):\n",
84       #text = re.sub(r'[m+ -- \\[*. \\?** \\d+ \\^ \\s]', '', txt)\n",
85       text = re.sub(r'--', '', txt)\n",
86       text = re.sub('[[\\[\\.\\?\\[\\]]]', '', text)\n",
87       text = re.sub(r'(\\\b|\\s+\\?-?|^\\-?) (\\d+\\d*\\.\\d+ )\\b', '', text)\n",
88       return text"
89   ],
90 },
91 {
92   "cell_type": "code",
93   "execution_count": null,
94   "id": "3ea7e207",
95   "metadata": {},
96   "outputs": [],
97   "source": [
98     "#remove chapter indicators\n",
99     "hamlet = re.sub(r'Chapter \\d+', '', hamlet)\n",
100    "macbeth = re.sub(r'Chapter \\d+', '', macbeth)\n",
101    "caesar = re.sub(r'Chapter \\d+', '', caesar)\n",
102    "#cleaning the texts\n",
103    "hamlet = clean_txt(hamlet)\n",
104    "macbeth = clean_txt(macbeth)\n",
105    "caesar= clean_txt(caesar)\n",
106    "print(hamlet[:250])"
107   ],
108 }

```

```

109 },
110 "cell_type": "code",
111 "execution_count": null,
112 "id": "dbdeb146",
113 "metadata": {},
114 "outputs": [],
115 "source": [
116     "# parsing the cleaned text \n",
117     "# spacy.load() is used as a wrapper to read the pipeline by means of language:'en' to construct language object\n",
118     "lang_obj = spacy.load('en_core_web_sm')\n",
119     "hamlet_obj = lang_obj(hamlet)\n",
120     "macbeth_obj = lang_obj(macbeth)\n",
121     "caesar_obj = lang_obj(caesar)\n",
122     "print(hamlet_obj)"
123 ],
124 },
125 {
126     "cell_type": "code",
127     "execution_count": null,
128     "id": "00a15d32",
129     "metadata": {},
130     "outputs": [],
131     "source": [
132         "# combining the sentences in the documents, language object consists of 'word' elements, that's why hamlet_obj[:100] is
133         "# hamlet_sents[:100]\n",
134         "hamlet_sents = ' '.join([sent.text for sent in hamlet_obj.sents if len(sent.text) > 1])\n",
135         "macbeth_sents = ' '.join([sent.text for sent in macbeth_obj.sents if len(sent.text) > 1])\n",
136         "caesar_sents = ' '.join([sent.text for sent in caesar_obj.sents if len(sent.text) > 1])\n",
137         "# combination of three novels \n",
138         "HMC = hamlet_sents + macbeth_sents + caesar_sents\n",
139         "print(len(HMC))"
140     ],
141 },
142 {
143     "cell_type": "code",
144     "execution_count": null,
145     "id": "16092e34",
146     "metadata": {},
147     "outputs": [],
148     "source": [
149         "#create text generator using markovify\n",
150         "# State size is a number of words the probability of a next word depends on.\n",
151         "# for text generation: we will build Markov model using three of Shakespeares' Tragedies from the Project Gutenberg NLT
152         "gen = markovify.Text(HMC, state_size=1)\n",
153         "\n",
154         "#generating short and long sentences  using make_sentance() and make_short_sentence()\n",
155         "print('long sentences: \n')\n",
156         "for i in range(4):\n",
157             "    print(gen.make_sentence())\n",
158             "\n",
159         "print('\n short sentences: \n')\n",
160         "for i in range(4):\n",
161             "# of max 100 chars \n",
162             "    print(gen.make_short_sentence(100))\n"
163     ],
164 },
165 {
166     "cell_type": "code",
167     "execution_count": null,
168     "id": "8d5b8eac",
169     "metadata": {},
170     "outputs": [],
171     "source": [
172         "# to improve the text prediction we will use POSifiedText class: spaCy tagger to generate a Markov model that comply wi
173         "# sentence structure better than a naive model.\n",
174         "\n",
175         "#in spacy library POS tagging is the process of marking a word in the text\n",
176         "#to a particular part of speech based on both its context and definition.\n",
177         "#In simple language, we can say that POS tagging is the process of identifying a word as nouns,\n",
178         "#pronouns, verbs, adjectives, etc.\n",
179         "\n",
180         "class POSifiedText(markovify.Text):\n",
181             "    def word_split(self, sentence):\n",
182                 "        return [':'.join((word.orth_, word.pos_)) for word in lang_obj(sentence)] # add word tags(positions)\n",
183             "    def word_join(self, words):\n",
184                 "        sentence = ' '.join(word.split(':')[0] for word in words)# re-build senetences for the model\n",
185                 "        return sentence\n",
186             "generator_2 = POSifiedText(HMC, state_size=2)\n",
187             "generator_2"
188     ],
189 },
190 {
191     "cell_type": "code",
192     "execution_count": null,
193     "id": "065d21fe",
194     "metadata": {},
195     "outputs": [],
196     "source": [
197         "print('short sentences: \n')\n",
198         "for i in range(4):\n",
199             "    print(generator_2.make_short_sentence(max_chars=100))\n",
200             "
201     ],
202 },
203 {
204     "cell_type": "code",
205     "execution_count": null,
206     "id": "b790a242",
207     "metadata": {},
208     "outputs": [],
209     "source": [
210         "print('\n long sentences: \n')\n",
211         "for i in range(4):\n",
212             "    print(generator_2.make_sentence())\n"

```

```

213     ]
214   }
215 ],
216 "metadata": {
217   "kernelspec": {
218     "display_name": "Python 3 (ipykernel)",
219     "language": "python",
220     "name": "python3"
221   },
222   "language_info": {
223     "codemirror_mode": {
224       "name": "ipython",
225       "version": 3
226     },
227     "file_extension": ".py",
228     "mimetype": "text/x-python",
229     "name": "python",
230     "nbconvert_exporter": "python",
231     "pygments_lexer": "ipython3",
232     "version": "3.9.13"
233   },
234 },
235 "nbformat": 4,
236 "nbformat_minor": 5
237 }

```

## Themenmodellierung

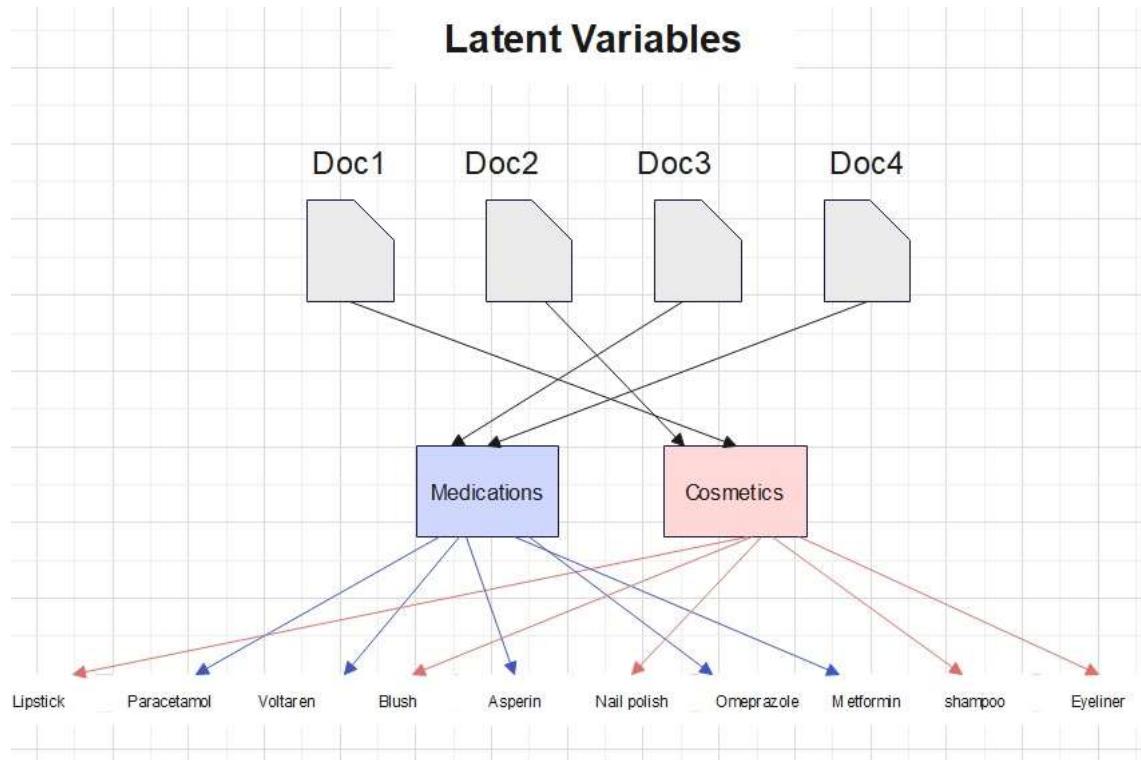
Die Themenmodellierung ist eine unbeaufsichtigte Lernmethode, die darauf abzielt, verborgene Strukturen in einem Text aufzudecken und die in einer Sammlung von Dokumenten vorkommenden Themen zu extrahieren, die die darin enthaltenen Informationen am besten repräsentieren.

Es gibt mehrere vorhandene Algorithmen, die zur Themenmodellierung verwendet werden können. Die häufigsten davon sind Latent Semantic Analysis (LSA/LSI), Probabilistic Latent Semantic Analysis (pLSA) und Latent Dirichlet Allocation (LDA).

LDA ist ein generatives Wahrscheinlichkeitsmodell, das davon ausgeht, dass jedes Thema eine Mischung aus einer Reihe von Wörtern und jedes Dokument eine Mischung aus einer Reihe von Themenwahrscheinlichkeiten ist. Für jedes Dokument „D“ und jedes Wort „w“ berechnet LDA zwei Wahrscheinlichkeitswerte für jedes Thema „k“:

1. P1: der Anteil der Wörter im Dokument (D), die aktuell dem Thema (k) zugeordnet sind.
2. P2: der Anteil der Zuordnungen zum Thema (k) an allen Dokumenten, die aus diesem Wort 'w' stammen.

Das LDA-Modell hat vier Hauptparameter; Alpha ist ein Prior-Parameter, der die Themendichte eines Dokuments darstellt. Höhere Alpha-Werte weisen auf eine größere Verteilung der Themen pro Dokument hin. Während der Beta-Parameter ein Parameter ist, der die Themenworddichte darstellt. Höhere Betawerte weisen darauf hin, dass Themen aus einer großen Anzahl von Wörtern im Korpus bestehen. Anzahl der Themen die aus dem Korpus extrahiert werden sollen ist auch ein wichtiger Parameter, so wie Anzahl der Themenbegriffe, der Anzahl der Begriffe angibt, die in einem einzelnen Thema zusammengefasst sind.



LDA ist ein iterativer Prozess. In der ersten Iteration werden die Themen zufällig jedem Wort im Dokument zugewiesen. LDA stellt nach der ersten Iteration die anfänglichen Dokument-Themen- und Themen-Wort-Matrizen bereit. Die vorliegende Aufgabe besteht darin, diese erzielten Ergebnisse zu erreichen, was LDA durch Iteration über alle Dokumente und alle Wörter durchführt. Bei jeder Iteration geht LDA davon aus, dass alle Themen und Wörter, die bisher zugewiesen wurden, korrekt sind, mit Ausnahme des aktuellen Dokuments und Wortes. Für jedes Dokument und Wort ,LDA schätzt einen neuen Wert ( $p_1 * p_2$ ), der die Wahrscheinlichkeit darstellt, dass das Wort „w“ für das Thema K relevant ist.

## Themenmodellierung Übung

Für die Themenmodellierung werden wir die auf der NeurIPS (NIPS)-Konferenz veröffentlichten Beiträge verwenden. Bevor wir unser Themenmodell erstellen, müssen wir den Text bereinigen, was bedeutet, dass wir Metadaten sowie unnötige Symbole und Satzzeichen entfernen. Darüber hinaus werden nicht alle Papiere für die Modellierung verwendet, sondern wir nehmen 200 Stichproben aus dem Datensatz:

```
In [3]: #we'll use the dataset of papers published in NeurIPS (NIPS) conference
import pandas as pd
import os

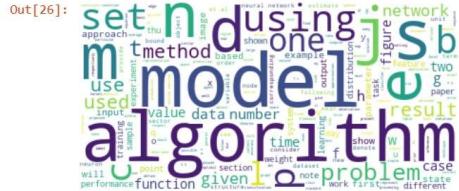
# importing paper data
papers = pd.read_csv('papers.csv')
# head
print(len(papers))
#papers.head()
# removing some metadata columns such as id, event type and pdf name, sampling 200 rows or 200 papers
paper_cleaned = papers.drop(columns = ['id', 'pdf_name', 'event_type'], axis=1).sample(200)
# remove punctuation and upper case
import re
def clean_txt(txt):
    #text = re.sub(r'[m+- -- \/*\/* \?/* \d+ \^ \s]', '', txt)
    text = re.sub('---', '', txt)
    text = re.sub('[[]]*?', '', text)
    text = re.sub('(\b|\s+\-?\^-\?)(\d+\|\d*\.\d+)\b', '', text)
    return text
paper_cleaned['processed_text'] = paper_cleaned['paper_text'].map(lambda x: clean_txt(x))
paper_cleaned['processed_text'] = paper_cleaned['processed_text'].map(lambda x: x.lower())
# print out the first 10 rows of the processed text
paper_cleaned['processed_text'].head(10)
```

7241

```
Out[3]: 1069    learning spike-based correlations and\nconditi...
3237    transduction with matrix completion:\nthree bi...
1140    308\n\ndonnett and smithers\n\nneuronal group ...
```

Jetzt sollten wir die Papiere zu einem Text zusammenfassen. Mithilfe des Wortwolkenmoduls können wir einen Blick auf die Wortverteilung in unserem Korpus werfen:

```
In [26]: # import wordcloud library
# !pip install WordCloud
from wordcloud import WordCloud
# combining the text in all papers, so that we apply text modelling
All_papers = ','.join(list(paper_cleaned['processed_text'].values))
# Create a WordCloud object
wc = WordCloud(background_color="white", max_words=5000, contour_width=3, contour_color='steelblue')
# Generate a word cloud
wc.generate(All_papers)
# Visualize the word cloud
wc.to_image()
```



Vor der Anwendung der LDA-Textmodellierung sollten Wörter im Text in eine Form umgewandelt werden, die das Modell erkennen kann. Dies wird als Wort-Tokenisierung bezeichnet. Dies bedeutet auch Kleinschreibung und Deakzentierung von Wörtern. Das Gensim-Python-Modul wird verwendet, um die Wörter zu tokenisieren und den Akzent zu entfernen. Auch Stopwörter müssen herausgefiltert werden:

```
In [53]: # tokenization of the text before training LDA model
# using gensim simple preprocess function we lowercase, tokenize, de-accent the text
# tokens are unicode strings of a set of characters those have a meaning together.
import gensim
import nltk
from gensim.utils import simple_preprocess
nltk.download('stopwords')
from nltk.corpus import stopwords

# stop words of the English dictionary
stop_w = stopwords.words('English')
#stop_words.extend(['from', 'subject', 're', 'edu', 'use', 'of', 'as', 'by', 'uc'])
# we use yield generator and not return when we want to iterate over a sequence, but don't want to store the entire sequence in memory.

def doc_to_tokens(text):
    for doc in text:
        # deacc=True removes accent
        yield(gensim.utils.simple_preprocess(str(doc), deacc=True))

def rm_stopwords(texts):
    wrds = [[word for word in doc if word not in stop_w] for doc in texts]# for each doc in the text, remove stop words
    return wrds

# joining the paper content in a list of sentences
txt = paper_cleaned.processed_text.values.tolist() # txt is a collection of sentences
tokens = list(doc_to_tokens(txt))
#tokens[0][10]
words = rm_stopwords(tokens)
# print(words[1][0][:30])
```

Der letzte Schritt vor der Modellierung des Textes besteht darin, eine Wortsammlung im Korpus zu erstellen. Ein Wortbeutel repräsentiert Tupel von Wörtern und deren Häufigkeit für alle Wörter im Dokument. Dieser Schritt erfordert die Erstellung eines Wörterbuchs der Wörter in den Dokumenten, was bedeutet, dass eine Zuordnung zwischen Wörtern und ihren ganzzahligen IDs erstellt wird.

```
In [12]: # create the dictionary of the text to be modelled; a mapping between words and their integer ids
import gensim.corpora as corpora

# build dictionary of the text
word_dict = corpora.Dictionary(docs)

# doc2bow : Convert document (a list of words) into the bag-of-words format = List of (token_id, token_count) 2-tuples.
# term frequency
corpus = [word_dict.doc2bow(doc) for doc in docs]

# View
len(corpus)
print(corpus[:1][0][:10])# print out the tuples
```

[(0, 3), (1, 1), (2, 1), (3, 3), (4, 1), (5, 1), (6, 1), (7, 1), (8, 1), (9, 4)]

Jetzt wenden wir die LDA-Themenmodellierung mit der LdaMulticore-Funktion an, die eine Parallelisierung des Trainingsprozesses ermöglicht, um den Aufbau unseres Modells zu beschleunigen. Lassen Sie uns gemäß dem Modell 20 Schlüsselwörter der 10 wichtigsten Themen im Korpus ausdrucken:

```
In [63]: # training the model
# we will build a model with 10 topics,
# where each topic is a combination of keywords, each keyword holds a specific contribution/weight to the topic.
from pprint import pprint
# building LDA text modelling using all CPU cores to parallelize and speed up model training.
lda_mo = gensim.models.LdaMulticore(corpus=corpus,
                                    id2word=id2word,
                                    num_topics=10)
# Printing the Keywords in the 10 topics
# Get the most significant topics
# num_words: The number of keywords to be included per topics (ordered by significance).
print(lda_mo.print_topics( num_words=20))
```

```
[(), ('0.005*"learning" + 0.005*"data" + 0.005*"model" + 0.005*"network" + ' '0.005*"algorithm" + 0.005*"set" + 0.004*"time" + 0.004*"function" + ' '0.004*"using" + 0.004*"one"'), (1, ('0.006*"algorithm" + 0.005*"data" + 0.005*"model" + 0.005*"set" + ' '0.005*"learning" + 0.004*"function" + 0.004*"number" + 0.004*"using" + ' '0.004*"one" + 0.003*"used"'), (2, ('0.006*"learning" + 0.005*"model" + 0.005*"algorithm" + 0.004*"function" + ' '0.004*"data" + 0.004*"set" + 0.004*"using" + 0.004*"time" + 0.003*"problem" + 0.003*"also"'), (3, ('0.007*"model" + 0.005*"function" + 0.005*"algorithm" + 0.005*"learning" + ' '0.004*"data" + 0.004*"using" + 0.004*"set" + 0.004*"training" + ' '0.004*"problem" + 0.003*"time"'),
```

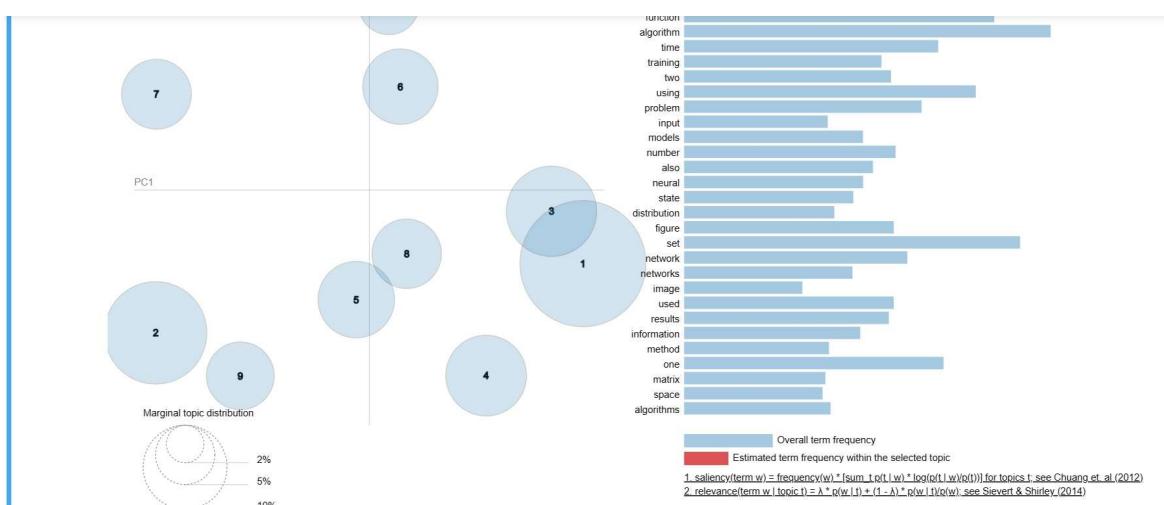
Jetzt würden wir das Ergebnis der Themenmodellierung mithilfe der Prepare-Funktion aus dem pyLDAvis-Modul visualisieren. Diese Funktion wandelt die Themenmodellverteilungen und zugehörigen Korpusdaten in die für die Visualisierung erforderlichen Strukturen um. Da dieser Schritt zeitaufwändig ist, können wir die Ausgabe in eine Datei schreiben und sie jedes Mal zur Ausgabevisualisierung verwenden, ohne die Strukturen erneut berechnen zu müssen.

Das linke Diagramm zeigt die marginale Themenverteilung der Top-10-Themen, die die „Bedeutung“ jedes Themas für das gesamte Korpus darstellt: den Prozentsatz, den das Thema im Korpus ausmacht. Die Fläche dieser Themenkreise ist proportional zur Anzahl der Wörter, die zu jedem Thema im Wörterbuch gehören. während das rechte Diagramm die Gesamt- und geschätzten Begriffshäufigkeiten der 30 wichtigsten Begriffe zeigt. Salient ist eine Metrik, mit der die aussagekräftigsten Wörter zur Identifizierung von Themen im Korpus ermittelt werden. Höhere Ausprägungswerte weisen darauf hin, dass ein Wort für die Identifizierung eines bestimmten Themas bedeutsamer oder informativer ist. Die ausgegebene „Geschätzte Begriffshäufigkeit innerhalb des ausgewählten Themas“ wird aus Themen-Begriffs-Wahrscheinlichkeiten abgeleitet, die von unserem Themenmodell berechnet wurden.

```
In [77]: # analyzing LDA model output under using of visualization package, pyLDAvis
#!pip install pyLDAvis
import pyLDAvis.gensim_models
import pickle
import pyLDAvis

# enable automatic D3 display of prepared model data in the IPython notebook.
pyLDAvis.enable_notebook()
LDAvis_data_filepath = os.path.join('./ldavis_prepared')
## this is a bit time consuming - make the if statement True
## if you want to execute visualization prep yourself

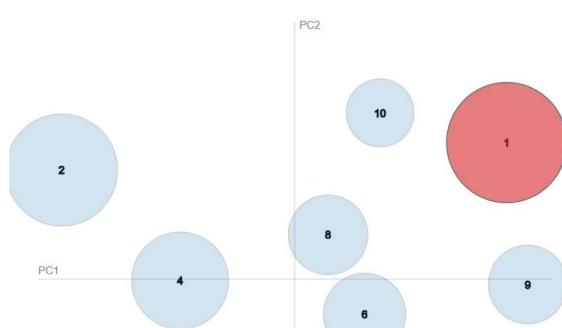
if 1 == 1:
    # prepare: transforms the topic model distributions and related corpus data into the data structures needed for the visualization.
    # corpus is the term frequency, word_dict is the vocab parameter,
    # R is the number of terms to display in the barcharts of the visualization. Default is 30
    LDAvis_prepared = pyLDAvis.gensim_models.prepare(lda_mo, corpus, word_dict#, R= 20)
    with open(LDAvis_data_filepath, 'wb') as f:
        pickle.dump(LDAvis_prepared, f)
    # Load the pre-prepared pyLDAvis data from disk
    with open(LDAvis_data_filepath, 'rb') as f:
        LDAvis_prepared = pickle.load(f)
    #write the visualization to a standalone html file, so that we have later only to read and display
    # the file (we don't need to prepare the data for visualization again)
    pyLDAvis.save_html(LDAvis_prepared, './ldavis_prepared' + '.html')
LDAvis_prepared
# the outputted 'Estimated term frequency within the selected topic' is derived from topic-term probabilities(given by lda_mo).
# the marginal topic distribution is the "importance" of each topic for the whole corpus:the percentage that the topic makes up in the corpus
```



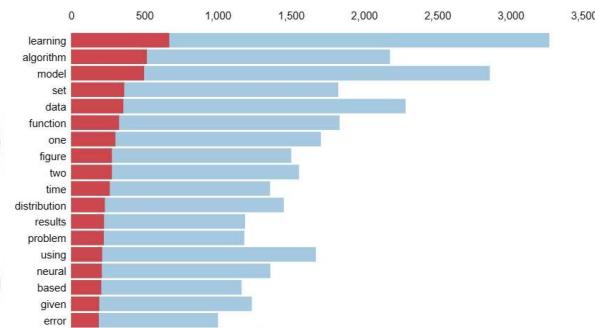
In[7]: Selected Topic: 1 Previous Topic Next Topic Clear Topic

Slide to adjust relevance metric:(2)  $\lambda = 1$  0.0 0.2 0.4 0.6 0.8 1

Intertopic Distance Map (via multidimensional scaling)



Top-30 Most Relevant Terms for Topic 1 (17.2% of tokens)



```
1 {
2     "cells": [
3         {
4             "cell_type": "code",
5             "execution_count": null,
6             "id": "ddee2ccd",
7             "metadata": {},
8             "outputs": [],
9             "source": [
10                 "# we'll use the dataset of papers published in NeurIPS (NIPS) conference\n",
11                 "import pandas as pd\n",
12                 "import os\n",
13                 "\n",
14                 "# importing paper data \n",
15                 "papers = pd.read_csv('papers.csv')\n",
16                 "# head \n",
17                 "print(len(papers))\n",
18                 "#papers.head()\n",
19                 "# removing some metadata columns such as id, event type and pdf name, sampling 200 rows or 200 papers\n",
20                 "paper_cleaned = papers.drop(columns = ['id', 'pdf_name', 'event_type'], axis =1).sample(200)\n",
21                 "# remove punctuation and upper case\n",
22                 "import re\n",
23                 "def clean_txt(txt):\n",
24                     #text = re.sub(r'[m+ -- \\\*. \\\??* \\d+ \\^ \\s]', '', txt)\n",
25                     text = re.sub('--', '', txt)\n",
26                     text = re.sub('[\\\[].*?[\\]]', '', text)\n",
27                     text = re.sub('(\\\b\\\s+\\-?|^-?)\\d+|\\d*\\.\\d+ )\\b', '', text)\n",
28                     return text\n",
29                 "paper_cleaned['processed_text'] = paper_cleaned['paper_text'].map(lambda x: clean_txt(x))\n",
30                 "paper_cleaned['processed_text'] = paper_cleaned['processed_text'].map(lambda x: x.lower())\n",
31                 "\n",
32                 "# print out the first 10 rows of the processed text\n",
33                 "paper_cleaned['processed_text'].head(10)\n"
34             ],
35         },
36     {
37         "cell_type": "code",
38         "execution_count": null,
39         "id": "464f15c2",
40         "metadata": {},
41         "outputs": [],
42         "source": [
43             "# import wordcloud library\n",
44             "#!pip install WordCloud\n",
45             "from wordcloud import WordCloud\n",
46             "# combining the text in all papers, so that we apply text modelling\n",
47             "All_papers = ','.join(list(paper_cleaned['processed_text'].values))\n",
48             "# Create a WordCloud object\n",
49             "wc = WordCloud(background_color=\"white\", max_words=5000, contour_width=3, contour_color='steelblue')\n",
50             "# Generate a word cloud\n",
51             "wc.generate(All_papers)\n",
52             "# Visualize the word cloud\n",
53             "wc.to_image()"
54         ],
55     },
56     {
57         "cell_type": "code",
58         "execution_count": null,
59         "id": "88d4685b",
60         "metadata": {},
61         "outputs": [],
62         "source": [
63             "# tokenization of the text before training LDA model \n",
64             "# using gensim simple_preprocess function we lowercase, tokenize, de-accent the text\n",
65             "# tokens are unicode strings of a set of characters those have a meaning together.\n",
66             "import gensim\n",
67             "import nltk\n",
68             "from gensim.utils import simple_preprocess\n",
69             "nltk.download('stopwords')\n",
70             "from nltk.corpus import stopwords\n",
71             "\n",
72             "# stop words of the English dictionary\n",
73             "stop_w = stopwords.words('English')\n",
74             "#stop_words.extend(['from', 'subject', 're', 'edu', 'use','of', 'as', 'by', 'uc'])\n",
75             "# we use yield generator and not return when we want to iterate over a sequence, but don't want to store the entire seq\n76             "\n",
77             "def doc_to_tokens(text):\n",
78                 for doc in text:\n",
79                     # deacc=True removes accent\n",
80                     yield(gensim.utils.simple_preprocess(str(doc), deacc=True))# tokenize each document\n",
81             "
82         ]
83     }
84 }
```

```

81         \n",
82     "def rm_stopwords(texts):\n",
83     "    wrds = [[word for word in doc if word not in stop_w] for doc in texts]\n# for each doc in the text, remove stop words
84     "    return wrds\n",
85     "\n",
86     "# joining the paper content in a list of sentences\n",
87     "txt = paper_cleaned.processed_text.values.tolist() # txt is a collection of sentences\n",
88     "tokens = list(doc_to_tokens(txt)) \n",
89     "#tokens[0][:10]\n",
90     "docs = rm_stopwords(tokens)\n",
91     "\n"
92   ]
93 },
94 {
95   "cell_type": "code",
96   "execution_count": null,
97   "id": "49358134",
98   "metadata": {},
99   "outputs": [],
100  "source": [
101    "print(words[:1][0][:30])"
102  ],
103 },
104 {
105   "cell_type": "code",
106   "execution_count": null,
107   "id": "3d5a7c9d",
108   "metadata": {},
109   "outputs": [],
110   "source": [
111     "# create the dictionary of the text to be modelled; a mapping between words and their integer ids\n",
112     "import gensim.corpora as corporo\n",
113     "\n",
114     "# build dictionary of the text\n",
115     "word_dict = corporo.Dictionary(docs)\n",
116     "\n",
117     "# doc2bow : Convert document (a list of words) into the bag-of-words format = list of (token_id, token_count) 2-tuples.
118     "# term frequency\n",
119     "corpus = [word_dict.doc2bow(doc) for doc in docs]\n",
120     "\n",
121     "# View\n",
122     "len(corpus)\n",
123     "print(corpus[:1][0][:10])# print out the tupels"
124   ],
125 },
126 {
127   "cell_type": "code",
128   "execution_count": null,
129   "id": "c0a72789",
130   "metadata": {},
131   "outputs": [],
132   "source": [
133     "len(words)"
134   ],
135 },
136 {
137   "cell_type": "code",
138   "execution_count": null,
139   "id": "907c65b3",
140   "metadata": {},
141   "outputs": [],
142   "source": [
143     "# training the model\n",
144     "# we will build a model with 10 topics,\n",
145     "# where each topic is a combination of keywords, each keyword holds a specific contribution/weight to the topic.\n",
146     "#from pprint import pprint\n",
147     "# building LDA text modelling using all CPU cores to parallelize and speed up model training.\n",
148     "lda_mo = gensim.models.LdaMulticore(corpus=corpus,\n",
149     "                                     id2word=word_dict,\n",
150     "                                     num_topics=10)\n",
151     "# Printing the Keywords in the 10 topics\n",
152     "# Get the most significant topics\n",
153     "# num_words: The number of keywords to be included per topics (ordered by significance).\n",
154     "pprint(lda_mo.print_topics( num_words=20))\n"
155   ],
156 },
157 {
158   "cell_type": "code",
159   "execution_count": null,
160   "id": "44d71190",
161   "metadata": {},
162   "outputs": [],
163   "source": [
164     "# Get the most significant topics\n",
165     "# num_words: The number of keywords to be included per topics (ordered by significance).\n",
166     "pprint(lda_mo.print_topics(num_words=20))\n"
167   ],
168 },
169 {
170   "cell_type": "code",
171   "execution_count": null,
172   "id": "078ea976",
173   "metadata": {},
174   "outputs": [],
175   "source": [
176     "# analyzing LDA model output under using of visualization package, pyLDAvis\n",
177     "\n",
178     "#!pip install pyLDAvis\n",
179     "import pyLDAvis.gensim_models\n",
180     "import pickle\n",
181     "import pyLDAvis\n",
182     "\n",
183     "# enable automatic D3 display of prepared model data in the IPython notebook.\n",
184     "pyLDAvis.enable_notebook()\n",

```

```

185 "LDAvis_data_filepath = os.path.join('./ldavis_prepared')\n",
186 "# # this is a bit time consuming - make the if statement True\n",
187 "# # if you want to execute visualization prep yourself\n",
188 "\n",
189 "if 1 == 1:\n",
190     "# prepare: transforms the topic model distributions and related corpus data into the data structures needed for the
191     "# corpus is the term frequency, word_dict is the vocab parameter,\n",
192     "# R is the number of terms to display in the barcharts of the visualization. Default is 30\n",
193     "LDAvis_prepared = pyLDAvis.gensim_models.prepare(lda_mo, corpus, word_dict)#, R= 20)\n",
194     "with open(LDAvis_data_filepath, 'wb') as f:\n",
195         pickle.dump(LDAvis_prepared, f)\n",
196     "# load the pre-prepared pyLDAvis data from disk\n",
197     "with open(LDAvis_data_filepath, 'rb') as f:\n",
198         LDAvis_prepared = pickle.load(f)\n",
199     "#write the visualization to a standalone html file, so that we have later only to read and display\n",
200     "# the file(we don't need to prepare the data for visualization again)\n",
201     "pyLDAvis.save_html(LDAvis_prepared, './ldavis_prepared' +'.html')\n",
202     "LDAvis_prepared\n",
203     "# the outputted 'Estimated term frequency within the selected topic' is derived from topic-term probabilities(given by :
204     "# the marginal topic distribution is the ?importance? of each topic for the whole corpus:the percentage that the topic i
205     "
206     ]
207     ],
208     "metadata": {
209         "kernelspec": {
210             "display_name": "Python 3 (ipykernel)",
211             "language": "python",
212             "name": "python3"
213         },
214         "language_info": {
215             "codemirror_mode": {
216                 "name": "ipython",
217                 "version": 3
218             },
219             "file_extension": ".py",
220             "mimetype": "text/x-python",
221             "name": "python",
222             "nbconvert_exporter": "python",
223             "pygments_lexer": "ipython3",
224             "version": "3.9.13"
225         }
226     },
227     "nbformat": 4,
228     "nbformat_minor": 5
229 }

```

[papers.csv \(210.38 MB\)](#)

## Quiz

Was wird bei der HMM-basierten Spracherkennung verwendet?

- Akustisches Modell.
- Aussprachewörterbuch.
- Sprachmodell.
- Alles das oben Genannte.

**Richtig!**

[Auswerten](#)

In einem typischen HMM wird das Sprachsignal in \_\_\_\_ aufgeteilt?

- 10-Millisekunden-Fragmente.
- 5-Millisekunden-Fragmente.
- 20-Millisekunden-Fragmente.
- Keines der oben genannten.

**Richtig!**

[Auswerten](#)

Sprachaktivitätsdetektoren (VADs) werden verwendet, um:

- Die ausgegebenen Vektoren in ein oder mehrere Phoneme einzuordnen.
- Die Wortfolge zu finden, die am wahrscheinlichsten die Phoneme erzeugt haben.
- Ein Audiosignal nur auf die Teile zu reduzieren, die wahrscheinlich Sprache enthalten.

**Richtig!**

Auswerten

Zur Textgenerierung wird das nächste Wort in einem Text basierend auf \_\_\_\_\_ vorhergesagt:

- dem aktuell eingegebenen Wort.
- allen vorherigen Wörtern.
- dem nächsten Wort.

**Richtig!**

Auswerten

das Markov-Ketten Modell wird verwendet, um den Wörtern in einem Korpus mit unterschiedlicher Wahrscheinlichkeit zugeordnet zu werden

- Richtig
- Falsch

**Richtig!**

Auswerten

Welcher Wert (e) gibt ein LDA-Modell zurück?

- Die Wortverteilung für jedes Thema.
- die Themenverteilung für jedes Dokument.
- Alles das oben Genannte.

**Richtig!**

Auswerten

Die Themenmodellierung ist eine unbeaufsichtigte Lernmethode, die darauf abzielt:

- Die Themen zu klassifizieren.
- Die in einer Sammlung von Dokumenten vorkommenden Themen zu extrahieren.
- Schlüsselwörter in jedem Thema zu extrahieren.
- Alles das oben Genannte.

**Richtig!**

Auswerten

## Aufgabe

In dieser Aufgabe verwenden wir ein Korpus zur Themenmodellierung und Textgenerierung:

1. Tokenisieren Sie den Text und entfernen Sie Stopwörter.
2. Wenden Sie bei Bedarf Stemming- und Lemmatisierungsschritte an.
3. Erstellen Sie ein Korpuswörterbuch und ein LDA-Modell mit 3 Themen.
4. Drucken Sie die 5 häufigsten Wörter pro Thema aus.
5. Überprüfen Sie die Themenverteilung im Korpus. Drucken Sie das Thema jeder Phrase gemäß dem Modell aus.
6. Wir werden dasselbe Korpus verwenden, um einen Textgenerator zu erstellen. Beginnen Sie mit der Erstellung eines Sprachobjekts aus dem Text.
7. Bilden Sie aus dem Text 4 kurze Sätze à 100 Zeichen.

Laden Sie bitte Ihre Lösung als .ipynb Datei [hier](#) hoch.

```
1 {
2   "cells": [
3     {
4       "cell_type": "code",
5       "execution_count": null,
6       "id": "41781628",
7       "metadata": {},
8       "outputs": [],
9       "source": [
10         "corpus= [\"Presumed human remains? were recovered from the debris field of the doomed Titan submersible.\",\n",
11         \"    \"\n",
12         \"        \\"The company that owns the remotely operated vehicles that brought Titan's wreckage to the surface has success\n",
13         \"        \\" Putin said he ?did not doubt? the support of Russian citizens during the Wagner rebellion\",\\n\",\n",
14         \"        \\"Russia's Federal Security Service learned of the rebellion two days before it was due to take place\",\\n\",\n",
15         \"        \\"the Wagner rebellion, according to a Kremlin readout on Wednesday\",\\n\",
```

```

16      "More than a third of the US population is under air quality alerts due to smoke from Canadian wildfires.\",\n"
17      "\"Some of the worst air quality, which is classified as ?very unhealthy,? is centered over the Chicago area\",\\n"
18      "\"rain and storms will help cleanse the air\",\\n",
19      "\"Canada is seeing its worst fire season on record as almost five hundreds fires rage across the country\"]"
20  ]
21 },
22 {
23   "cell_type": "code",
24   "execution_count": null,
25   "id": "175df13d",
26   "metadata": {},
27   "outputs": [],
28   "source": [
29     "#tokenization of the text before training LDA model \\n",
30     "# using gensim simple_preprocess function we lowercase, tokenize, de-accent the text\\n",
31     "import gensim\\n",
32     "import nltk\\n",
33     "from gensim.utils import simple_preprocess\\n",
34     "nltk.download('stopwords')\\n",
35     "from nltk.corpus import stopwords\\n",
36     "from nltk.stem import WordNetLemmatizer\\n",
37     "from nltk.stem import PorterStemmer \\n",
38     "\\n",
39     "# stop words of the English dictionary\\n",
40     "stop_w = stopwords.words('English')\\n",
41     "\\n",
42     "# tokenization function\\n",
43     "def doc_to_tokens(text):\\n",
44     "    for doc in text:\\n",
45     "        # deacc=True removes accent\\n",
46     "        #use gensim simple_preprocess function we lowercase, tokenize, de-accent the text: see documentation: https://t
47     "        yield(???????)\\n",
48     "tokens = list(doc_to_tokens(corpus)) \\n",
49     "\\n",
50     "# stop words removal\\n",
51     "def rm_stopwords(texts):\\n",
52     "    #remove stop words for each doc in the text\\n",
53     "    wrds = ??????\\n",
54     "    return wrds\\n",
55     "tokens = rm_stopwords(tokens)\\n",
56     "\\n",
57     "# lemmatization, stemming\\n",
58     "docs =[]\\n",
59     "lemmatizer = WordNetLemmatizer()\\n",
60     "stemmer = PorterStemmer()\\n",
61     "for doc in tokens:\\n",
62     "    word =[]\\n",
63     "    for j in range(len(doc)):\\n",
64     "        # hint: use lemmatize(): see https://www.nltk.org/_modules/nltk/stem/wordnet.html\\n",
65     "        lemm= ???????\\n",
66     "        # hint: use stem, see https://www.nltk.org/howto/stem.html\\n",
67     "        stem= ???????\\n",
68     "        word.append(stem)\\n",
69     "    docs.append(word)\\n",
70     "\\n",
71     "print(docs) \\n",
72     "print(tokens)"
73   ],
74 },
75 {
76   "cell_type": "code",
77   "execution_count": null,
78   "id": "b0972678",
79   "metadata": {},
80   "outputs": [],
81   "source": [
82     "# create the dictionary of the text to be modelled; a mapping between words and their integer ids\\n",
83     "import gensim.corpora as corpora\\n",
84     "\\n",
85     "# build dictionary of the tokens, use corpora.Dictionary, see docuemtation: https://radimrehurek.com/gensim/corpora/dic
86     "word_dict = ???????\\n",
87     "\\n",
88     "# using the created dictionary in the last step ,Convert the token into the bag-of-words format = list of (token_id, to
89     "# use corpora.doc2bow function, see documentaion: https://tedboy.github.io/nlps/generated/generated/gensim.corpora.Dict
90     "# term frequency\\n",
91     "bow = [??????]\\n",
92     "\\n",
93     "# print out the tuples\\n",
94     "print(corpus[:1][0][:10])"
95   ],
96 },
97 {
98   "cell_type": "code",
99   "execution_count": null,
100  "id": "fc4a73e2",
101  "metadata": {},
102  "outputs": [],
103  "source": [
104    "# we will build an LDA model with 3topics,\\n",
105    "# where each topic is a combination of keywords, each keyword holds a specific contribution/weight to the topic.\\n",
106    "from pprint import pprint\\n",
107    "# building LDA text modelling on the corpus using all CPU cores to parallelize and speed up model training.\\n",
108    "# use ldaMulticore: https://tedboy.github.io/nlps/generated/generated/gensim.models.LdaMulticore.html\\n",
109    "lda_mo = ???????\\n",
110    "# Printing the Keywords in the 3 topics\\n",
111    "# hint: use pprint with print_topics , see documentation: https://tedboy.github.io/nlps/generated/generated/gensim.mode
112    "pprint(???????)"
113  ],
114 },
115 {
116   "cell_type": "code",
117   "execution_count": null,
118   "id": "d35a9b85",
119   "metadata": {}

```

```

120     "outputs": [],
121     "source": [
122         "# Return topic distribution for the given documents/phrases in bow, as a list of (topic_id, topic_probability) 2-tuples
123         "# hint : use get_document_topics(): see https://tedboy.github.io/nlp/generated/generated/gensim.models.LdaMulticore.hti
124         "doc_topics= ??????\n",
125         "# check phrase topics\n",
126         "for i in range(len(bow)):\n",
127             print("\\"\\n doc: \\\n", corpus[i], doc_topics[i])"
128     ],
129 },
130 {
131     "cell_type": "code",
132     "execution_count": null,
133     "id": "547ae86d",
134     "metadata": {},
135     "outputs": [],
136     "source": [
137         "# generate text using markov chains\n",
138         "import markovify\n",
139         "import spacy\n",
140         "import re\n",
141         "# we have to bring the corpus into a string of lines, that can be recognized by language object...\n",
142         "corpus = \"Presumed human remains were recovered from the debris field of the doomed Titan submersible.\n Titan was ma
143         \"\n",
144         "# build language object on the corpus\n",
145         "# spacy.load() is used as a wrapper to read the pipeline by means of language:'en' to construct language object\n",
146         "eng_obj = spacy.load('en_core_web_sm')\n",
147         "# create language object on the corpus: see https://spacy.io/models\n",
148         "lang_obj = ??????\n",
149         "\n",
150         "# lets see the language object sentences\n",
151         "for s in lang_obj.sents:\n",
152             print(s.text)"
153     ],
154 },
155 {
156     "cell_type": "code",
157     "execution_count": null,
158     "id": "298d9954",
159     "metadata": {},
160     "outputs": [],
161     "source": [
162         "#combining the sentences in the documents, language object consists of 'word' elements.\n",
163         "sents = ' '.join([sent.text for sent in lang_obj.sents if len(sent.text) > 1])\n",
164         "#build markov chains model on the sentences.\n",
165         "#hint: use markovify.Text, see the syntax here: https://github.com/jsvine/markovify\n",
166         "gen = ??????\n",
167         "\n",
168         "#generating short sentences using make_short_sentence(), up to 100 characters\n",
169         "#hint: use make_short_sentence: see https://github.com/jsvine/markovify\n",
170         "print('short sentences: \\\n')\n",
171         "for i in range(4):\n",
172             print(???????""
173     ],
174 },
175 ],
176 "metadata": {
177     "kernelspec": {
178         "display_name": "Python 3 (ipykernel)",
179         "language": "python",
180         "name": "python3"
181     },
182     "language_info": {
183         "codemirror_mode": {
184             "name": "ipython",
185             "version": 3
186         },
187         "file_extension": ".py",
188         "mimetype": "text/x-python",
189         "name": "python",
190         "nbconvert_exporter": "python",
191         "pygments_lexer": "ipython3",
192         "version": "3.9.13"
193     }
194 },
195 "nbformat": 4,
196 "nbformat_minor": 5
197 }

```

```

1 {
2     "cells": [
3     {
4         "cell_type": "code",
5         "execution_count": 5,
6         "id": "a656caa2",
7         "metadata": {},
8         "outputs": [],
9         "source": [
10             "corpus= \"Presumed human remains? were recovered from the debris field of the doomed Titan submersible.\",\n",
11             "        \"Titan was made of carbon fiber and titanium\",\\n",
12             "        \"The company that owns the remotely operated vehicles that brought Titan?s wreckage to the surface has success",
13             "        \" Putin said he ?did not doubt? the support of Russian citizens during the Wagner rebellion\",\\n",
14             "        \"Russia?s Federal Security Service learned of the rebellion two days before it was due to take place\",\\n",
15             "        \"the Wagner rebellion, according to a Kremlin readout on Wednesday\",\\n",
16             "        \"More than a third of the US population is under air quality alerts due to smoke from Canadian wildfires.\",\\n",
17             "        \"Some of the worst air quality, which is classified as ?very unhealthy?, is centered over the Chicago area\",\\n",
18             "        \"rain and storms will help cleanse the air\",\\n",
19             \"Canada is seeing its worst fire season on record as almost five hundreds fires rage across the country\"]"
20     ],
21 }

```

```

22  {
23      "cell_type": "code",
24      "execution_count": 6,
25      "id": "dfb74ec6",
26      "metadata": {},
27      "outputs": [
28          {
29              "name": "stdout",
30              "output_type": "stream",
31              "text": [
32                  "[['presum', 'human', 'remain', 'recov', 'debris', 'field', 'doom', 'titan', 'submers'], ['titan', 'made', 'carbon', 'f'],
33                  "[['presumed', 'human', 'remains', 'recovered', 'debris', 'field', 'doomed', 'titan', 'submersible'], ['titan', 'made',
34              ]
35          },
36          {
37              "name": "stderr",
38              "output_type": "stream",
39              "text": [
40                  "[nltk_data] Downloading package stopwords to\n",
41                  "[nltk_data]     C:\\\\Users\\\\la2022\\\\AppData\\\\Roaming\\\\nltk_data...\\n",
42                  "[nltk_data]     Package stopwords is already up-to-date!\\n"
43              ]
44          }
45      ],
46      "source": [
47          "#tokenization of the text before training LDA model \\n",
48          "# using gensim simple_preprocess function we lowercase, tokenize, de-accent the text\\n",
49          "import gensim\\n",
50          "import nltk\\n",
51          "from gensim.utils import simple_preprocess\\n",
52          "nltk.download('stopwords')\\n",
53          "from nltk.corpus import stopwords\\n",
54          "from nltk.stem import WordNetLemmatizer\\n",
55          "from nltk.stem import PorterStemmer \\n",
56          "\\n",
57          "# stop words of the English dictionary\\n",
58          "stop_w = stopwords.words('English')\\n",
59          "\\n",
60          "\\n",
61          "def doc_to_tokens(text):\n",
62              for doc in text:\n",
63                  # deacc=True removes accent\\n",
64                  yield(gensim.utils.simple_preprocess(str(doc), deacc=True))\\n",
65          "tokens = list(doc_to_tokens(corpus)) \\n",
66          "\\n",
67          "def rm_stopwords(texts):\n",
68              wrds = [word for word in doc if word not in stop_w] for doc in texts]# for each doc in the text, remove stop words
69              return wrds\\n",
70          "tokens = rm_stopwords(tokens)\\n",
71          "# lemmatization, stemming\\n",
72          "docs =[]\\n",
73          "lemmatizer = WordNetLemmatizer()\\n",
74          "stemmer = PorterStemmer()\\n",
75          "for doc in tokens:\\n",
76              word =[]\\n",
77              for j in range(len(doc)):\\n",
78                  \\n",
79                  lemm= lemmatizer.lemmatize(doc[j])\\n",
80                  stem= stemmer.stem(lemm)\\n",
81                  word.append(stem)\\n",
82              docs.append(word)\\n",
83          "\\n",
84          "print(docs) \\n",
85          "print(tokens)"
86      ]
87  },
88  {
89      "cell_type": "code",
90      "execution_count": 3,
91      "id": "b0a5e2fc",
92      "metadata": {},
93      "outputs": [
94          {
95              "name": "stdout",
96              "output_type": "stream",
97              "text": [
98                  "[0, 1), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1), (7, 1), (8, 1)]\\n"
99              ]
100         }
101     ],
102     "source": [
103         "# create the dictionary of the text to be modelled; a mapping between words and their integer ids\\n",
104         "import gensim.corpora as corporo\\n",
105         "\\n",
106         "# build dictionary of the text\\n",
107         "word_dict = corporo.Dictionary(docs)\\n",
108         "\\n",
109         "# doc2bow : Convert document (a list of words) into the bag-of-words format = list of (token_id, token_count) 2-tuples.
110         "# term frequency\\n",
111         "bow = [word_dict.doc2bow(twt) for twt in docs]\\n",
112         "\\n",
113         "# print out the tuples\\n",
114         "print(bow[:1][0][:10])"
115     ]
116  },
117  {
118      "cell_type": "code",
119      "execution_count": 4,
120      "id": "24569dbf",
121      "metadata": {},
122      "outputs": [
123          {
124              "name": "stdout",
125              "output_type": "stream",

```

```

126     "text": [
127         "[0,\n",
128         " 0.031*\\"fire\\\" + 0.022*\\"readout\\\" + 0.022*\\"rebellion\\\" + 0.022*\\"titan\\\" + '\n",
129         " 0.022*\\"wednesday\\\"),\n",
130         "(1,\n",
131         " 0.036*\\"rebellion\\\" + 0.025*\\"due\\\" + 0.022*\\"titan\\\" + 0.021*\\"success\\\" + '\n",
132         " 0.021*\\"offshor\\\"),\n",
133         "(2,\n",
134         " 0.048*\\"air\\\" + 0.031*\\"worst\\\" + 0.029*\\"titan\\\" + 0.028*\\"qualiti\\\" + '\n",
135         " 0.028*\\"chicago\\\")\n"
136     ]
137   }
138 ],
139 "source": [
140   "\n",
141   "# we will build a model with 3 topics,\n",
142   "# where each topic is a combination of keywords, each keyword holds a specific contribution/weight to the topic.\n",
143   "from pprint import pprint\n",
144   "# building LDA text modelling using all CPU cores to parallelize and speed up model training.\n",
145   "lda_mo = gensim.models.LdaMulticore(corpus=bow,\n",
146   "                                    id2word=word_dict,\n",
147   "                                    num_topics=3)\n",
148   "# Printing the Keywords in the 3 topics\n",
149   "# Get the most significant topics\n",
150   "# num_words: The number of keywords to be included per topics (ordered by significance).\n",
151   "pprint(lda_mo.print_topics( num_words=5))\n"
152 ]
153 },
154 {
155   "cell_type": "code",
156   "execution_count": null,
157   "id": "3155d5cf",
158   "metadata": {},
159   "outputs": [],
160   "source": [
161     "doc_topics= lda_mo.get_document_topics(bow, minimum_probability=None)\n",
162     "# minimum_probability=None, to NOT ignore topics with very low probability.\n",
163     "# check phrase topics\n",
164     "for i in range(len(bow)):\n",
165       print("\\\\n doc: \\\\n", corpus[i], doc_topics[i])\n"
166   ],
167 },
168 {
169   "cell_type": "code",
170   "execution_count": 210,
171   "id": "648835db",
172   "metadata": {},
173   "outputs": [
174     {
175       "name": "stdout",
176       "output_type": "stream",
177       "text": [
178         "Presumed human remains were recovered from the debris field of the doomed Titan submersible.\n",
179         "\n",
180         "Titan was made of carbon fiber and titanium.\n",
181         "\n",
182         "The company that owns the remotely operated vehicles that brought Titan's wreckage to the surface has successfully cor",
183         "\n",
184         "Putin said he ?did not doubt? the support of Russian citizens during the Wagner rebellion.\n",
185         "\n",
186         "Russia's Federal Security Service learned of the rebellion two days before it was due to take place.\n",
187         "\n",
188         "the Wagner rebellion, according to a Kremlin readout on Wednesday.\n",
189         "\n",
190         "More than a third of the US population is under air quality alerts due to smoke from Canadian wildfires.\n",
191         "\n",
192         "Some of the worst air quality, which is classified as ?very unhealthy?, is centered over the Chicago area,rain and stc",
193         "\n",
194         "Canada is seeing its worst fire season on record as almost five hundreds fires rage across the country.\n"
195     ]
196   ],
197 },
198 ],
199 "source": [
200   "import markovify\n",
201   "import spacy\n",
202   "import re\n",
203   "# we have to bring the corpus into a string of lines, that can be recognized by language object...\n",
204   "corpus = \"Presumed human remains were recovered from the debris field of the doomed Titan submersible.\\n Titan was ma",
205   "\n",
206   "# build language object on the corpus\n",
207   "# spacy.load() is used as a wrapper to read the pipeline by means of language:'en' to construct language object\n",
208   "eng_obj = spacy.load('en_core_web_sm')\n",
209   "lang_obj = eng_obj(corpus)\n",
210   "\n",
211   "# lets see the language object sentences\n",
212   "for s in lang_obj.sents:\n",
213     "print(s.text)"
214   ],
215 },
216 ],
217 "cell_type": "code",
218 "execution_count": null,
219 "id": "882acd4f",
220 "metadata": {},
221 "outputs": [],
222 "source": [
223   "#combining the sentences in the documents, language object consists of 'word' elements, that's why hamlet_obj[:100] is :
224   "sents = ' .join([sent.text for sent in lang_obj.sents if len(sent.text) > 1])\n",
225   "gen = markovify.Text(sents, state_size=1)\n",
226   "\n"
227 ]
228 ],
229 
```

```
230     "#generating short sentences using make_short_sentence(), up to 50 characters\n",
231     "print('short sentences: \\\n')\n",
232     "for i in range(4):\n",
233         print(gen.make_short_sentence(100))\n"
234     ]
235 }
236 ],
237 "metadata": {
238     "kernelspec": {
239         "display_name": "Python 3 (ipykernel)",
240         "language": "python",
241         "name": "python3"
242     },
243     "language_info": {
244         "codemirror_mode": {
245             "name": "ipython",
246             "version": 3
247         },
248         "file_extension": ".py",
249         "mimetype": "text/x-python",
250         "name": "python",
251         "nbconvert_exporter": "python",
252         "pygments_lexer": "ipython3",
253         "version": "3.9.13"
254     }
255 },
256 "nbformat": 4,
257 "nbformat_minor": 5
258 }
```