

עיבוד שפה טבעית (Natural Language Processing) הוא תת-תחום של בינה מלאכותית. הוא עוסק בבעיות הקשורות לעיבוד ומניפולציה של שפה טבעית והבנה של שפה טבעית על מנת לגרום למחשבים "להבין" דברים שנאמרים או נכתבים בשפות אנושיות.

דוגמאות לבעיות העומדות בפני מערכות הבנת שפה טבעית:

למשפטים "נתנו לקופים את התפוזים משום שהם היו רעבים", ו"נתנו לקופים את התפוזים משום שהם היו רעבים", יש לכאורה מבנה תחבירי זהה, אף כי למעשה באחד מהם המילה "הם" מתייחסת לקופים, ואילו באחר לתפוזים. אי אפשר להבין את המשפט כראוי ללא היכרות עם המאפיינים והתכונות המתייחסים לקופים ולתפוזים.

בשפות רבות קשה למיין את מרכיבי המשפט ולסמן את היחסים ביניהם באמצעות ניתוח פשוט של המבנה התחבירי. למשל, יש שפות שקשה לדעת בהן לאיזה שם עצם מתייחס שם התואר, ויש שפות בהן אין סימון מורפולוגי לחלקי הדיבר.

כדי לפתור את הבעיות האלו, הציעו כמה בלשנים וחוקרי אינטליגנציה טבעית לעשות שימוש בשפה מלאכותית, שתוכל לבטא את כל הדקויות והעומק של השפות הטבעיות שאנו מכירים, אך עם זאת שתהיה בעלת תחביר וחוקי כתיב עקביים מבחינה לוגית או מתמטית, כדי להסיר כל עמימות או רב-משמעות הנובעת ממבנה המשפט.

כלומר, הבעיה העיקרית ב-NLP היא דו משמעויות. (יש כמה משמעויות ואנו רוצים לסווג את המשמעות הנכונה של המשפט).

מבחן טיורינג:

מבחן טיורינג הוא כינוי למבחן שהציע אלן טיורינג בשנת 1950, כמדד אפשרי למידה שבה יש למכונה כלשהי אינטליגנציה. המבחן נעשה בדרך הבאה: חוקר מקיים דיאלוג בשפה טבעית עם שני גורמים סמויים מעיניו, האחד אדם והשני מכונה. אם החוקר אינו מסוגל לקבוע בביטחון מי האדם ומי המכונה, אזי המכונה עברה בהצלחה את המבחן.

NLTK:

ספרייה לעיבוד שפות טבעיות בPython.

```
>>> my_text = "Where is St. Paul located? I don't seem to find it. It isn't in my map."
>>> my_text.split(" ") # or my_text.split()
['Where', 'is', 'St.', 'Paul', 'located?', 'I', 'don't', 'seem', 'to', 'find', 'it.', 'It', 'isn't', 'in', 'my', 'map.']
>>> import nltk
>>> from nltk.tokenize import word_tokenize, sent_tokenize
>>> word_tokenize(my_text)
['Where', 'is', 'St.', 'Paul', 'located?', '?', 'I', 'do', 'n't', 'seem', 'to', 'find', 'it', '.', 'It', 'is', 'n't', 'in', 'my', 'map', '.']
>>> sent_tokenize(my_text)
['Where is St. Paul located?', 'I don't seem to find it.', 'It isn't in my map.']
```

Tokenization - כאשר אנו רוצים לעבוד על משפט, הפעולה הפשוטה ביותר היא לעשות tokenization, מודל בסיסי זה אחראי לפצל את המשפט למילים תוך שימוש במפרדים. הרבה פעמים נרצה לפצל מסמך למשפטים המרכיבים אותו. בתמונה משמאל ניתן לראות דוגמא לביצוע tokenization באמצעות NLTK.

```
>>> from nltk.stem import PorterStemmer
>>> from nltk.tokenize import word_tokenize
>>> ps = PorterStemmer()
>>> my_text = "Whoever eats many cookies is regretting doing so"
>>> stemmed_sentence = []
>>> for word in word_tokenize(my_text):
...     stemmed_sentence.append(ps.stem(word))
>>> stemmed_sentence
['Whoever', 'eat', 'mani', 'cooki', 'is', 'regret', 'do', 'so']
```

Stemming - תהליך של צמצום מילה לגזע (שורש) המילה שלה המוצמד לסיומת. Stemming חשוב בהבנת השפה הטבעית (NLU) ובעיבוד השפה הטבעית (NLP). פעמים רבות נרצה להתייחס למילים שונות בכתיבתן כזהות (מילים שונות עם אותו הבסיס. למשל נרצה להתייחס לwalks ולwalking כwalk). ניקח מילה ונשלוף ממנה את הבסיס וכך נגיע לאותה מילה הדומה לה בשורש. תחילה, נצטרך להפריד את המילה ע"י שימוש בtokenization מאחר והstemmers מעבד כל מילה בנפרד. NLTK קיימים מספר אלגוריתמים לביצוע Stemming:

- from nltk.stem.porter import PorterStemmer
- from nltk.stem.lancaster import LancasterStemmer
- from nltk.stem import SnowballStemmer

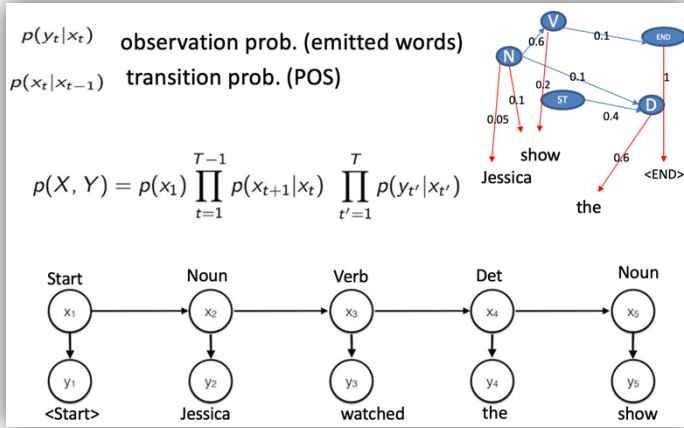
```
>>> my_tokenized_text =
word_tokenize(my_text)
>>> nltk.pos_tag(my_tokenized_text)
[('Whoever', 'NNP'), ('eats', 'VBZ'), ('many', 'JJ'), ('cookies', 'NNS'), ('is', 'VBZ'), ('regretting', 'VBG'), ('doing', 'VBG'), ('so', 'RB')]
>>> nltk.help.upenn_tagset()
```

POS (Part Of Speech) - תיוג חלקי דיבור הוא תהליך פופולרי לעיבוד שפה טבעית המקטלג מילים בטקסט בהתכתבות עם חלק מסוים בדיבור, בהתאם להגדרת המילה והקשר שלה. זהו מודל הנותן לכל מילה במשפט אנוטציה (פירוש) תחבירית לחלקי הדיבור הנכונים במשפט. כגון: פועל, שם עצם, תואר ועוד. באמצעות מודל זה נקבל המון מידע על המשפט ונוכל למצוא מילים שמעניינות אותנו ולהסתכל על החוקים שסביבן. בתמונה משמאל ניתן לראות דוגמא לביצוע POS באמצעות NLTK (כאן הקלט הוא משפט ולא מילה).

Tag	Meaning	English Examples
ADJ	adjective	new, good, high, special, big, local
ADP	adposition	on, of, at, with, by, into, under
ADV	adverb	really, already, still, early, now
CONJ	conjunction	and, or, but, if, while, although
DET	determiner, article	the, a, some, most, every, no, which
NOUN	noun	year, home, costs, time, Africa
NUM	numeral	twenty-four, fourth, 1991, 14:24
PRT	particle	at, on, out, over per, that, up, with
PRON	pronoun	he, their, her, its, my, I, us
VERB	verb	is, say, told, given, playing, would
.	punctuation marks	., : !
x	other	ersatz, esprit, dunno, gr8, univeristy

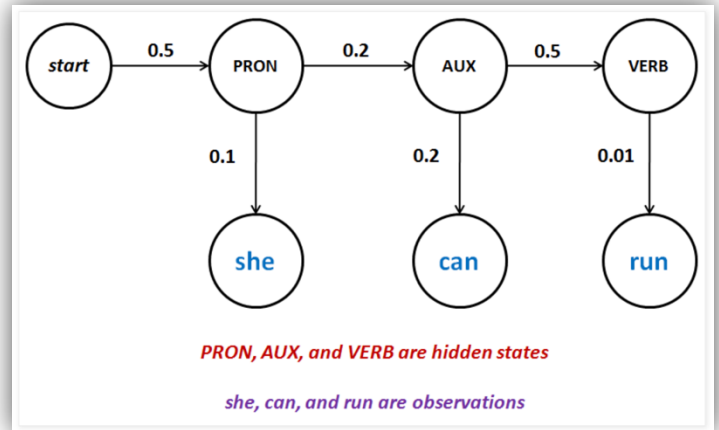
POS באמצעות Hidden Markov Models (HMM):

בהינתן רצף (מילים, אותיות, משפטים וכו'), HMM מחשב את ההסתברות על פני רצף של תוויות ומנבא את רצף התוויות הטוב ביותר. מודל זה מתבסס על ההנחה שיש קשר בין כל חלק דיבר במשפט לחלק הקודם לו. נרצה למצוא תיוג (קטלוג) של חלקי המשפט כדי שנקבל את המשפט הנתון. נחשב מה הסיכוי למעבר בין כל חלק במשפט, ואת הסיכוי שחלק דיבר כלשהו שייך למילה הנתונה. נבחר את הסידור בעל ההסתברות הגבוהה ביותר. כלומר, בהינתן משפט נרצה לראות מה סדרת חלקי הדיבר שהכי סביר שתתן לנו את משפט זה. הנחת מרקוב היא שהקשר בין חלקי תלוי אך ורק בחלק הדיבר של המילה הקודמת. Y נתון לנו והוא מייצג את המילים במשפט הנתון, נרצה למצוא X (רצף חלקי הדיבר המרכיבים את המשפט) בהנחה שהאים לא מושפעים מהע הקודם אלא רק מהא הקודם לו.



למעשה, זה מעין אוטומט. בכל פעם נמצאים במצב וממנו עוברים למצב הבא, כל מצב הוא חלק דיבר (x_i) וכל מצב מוליד לנו מילה (y_i) בהסתברות מסוימת. נשים לב ש המסלול ממנו הגענו אל המצב לא מעניין אותנו אלא רק המצב הקודם ממנו הגענו. בכל מעבר נחשב את ההסתברות לעבור ממצב למצב הבא בתור. נרצה למצוא מסלול ממצב ההתחלה (start) למצב הסיום (end) בעל ההסתברות הגבוהה ביותר. כדי לקבל את ההסתברות לכל מעבר ממצב למצב, נצטרך לעבור על data set (מאגר נתונים הנקרא Penn TreeBank שמכיל המון משפטים ואת חלקי הדיבר של כל אחת מהמילים במשפט), לבדוק לאיזה דיבר נעבור בהסתברות הגבוהה ביותר (לדוגמה נניח שברוב המשפטים במאגר אחרי דיבר מסוג noun מופיע דיבר מסוג verb) ולבדוק בהינתן חלק דיבר מה הסיכוי לקבל כל מילה במשפט. נניח שהמשפט הוא "אני אוהב בינה מלאכותית" ועלינו להקצות תגי POS לכל מילה. ברור שתגיות ה-POS לכל מילה הן "כינוי (PRP-Pronoun)", פועל (VBP-Verb), שם תואר (JJ-Adjective) ושם עצם (NN-Noun) בהתאמה. כדי לחשב את ההסתברויות לתגיות, עלינו לדעת תחילה מה הסיכוי שאחרי "כינוי" יגיע "פועל", לאחר מכן "שם תואר" ולבסוף "שם עצם". הסתברויות אלו נקראות הסתברויות מעברים. שנית, עלינו לדעת מה הסיכוי שהמילה "אני" תהיה כינוי, המילה "אוהב" תהיה פועל, המילה "בינה" תהיה שם תואר, והמילה "מלאכותית" תהיה שם עצם. הסתברויות אלו נקראות הסתברויות פליטה. ניתן לתאר את הסתברויות אלו באמצעות מטריצות.

Transition probabilities	Emission probabilities
$P(\text{NOUN} \text{PRON})=0.001$	$P(\text{she} \text{PRON})=0.1$
$P(\text{PRON} \text{START})=0.5$	$P(\text{run} \text{VERB})=0.01$
$P(\text{VERB} \text{AUX})=0.5$	$P(\text{can} \text{AUX})=0.2$
$P(\text{AUX} \text{PRON})=0.2$	$P(\text{can} \text{NOUN})=0.001$
$P(\text{NOUN} \text{AUX})=0.001$	$P(\text{run} \text{NOUN})=0.001$
$P(\text{VERB} \text{NOUN})=0.2$	
$P(\text{NOUN} \text{NOUN})=0.1$	



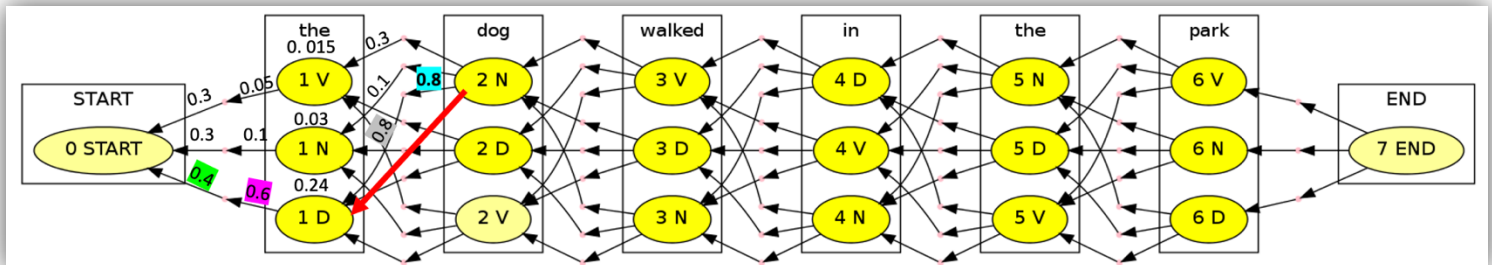
POS באמצעות Viterbi Algorithm:

אלגוריתם תכנון דינמי למציאת רצף המצבים החבויים (חלקי הדיבר) הסביר ביותר, שתוצאתו היא רצף תצפיות נתון. בין שימושי הנפוצים - מציאת רצף המצבים בהנחת מודל מרקוב חבוי, פיענוח קודי קונבולוציה ועוד. יתרונו הבולט של האלגוריתם הוא הורדת סיבוכיות המציאה של הרצף הסביר ביותר - מהמימוש הנאיבי (בעל סיבוכיות מעריכית בגודל הרצף) למימוש היעיל של האלגוריתם (בעל סיבוכיות ליניארית בגודל הרצף).

נתונות טבלאות עם כל ההסתברויות. כאשר סכום כל עמודה הוא 1. ישנן שתי טבלאות: אחת מכילה את ההסתברויות למעבר ממצב אחד (המצב בשורה) לאחר (המצב בעמודה), והשנייה מכילה הסתברויות מותנות, בהינתן חלק דיבר מה ההסתברות לקבל כל אחת מהמילים במשפט. בכל קודקוד נחשב את ההסתברות להגיע לקודקוד זה עם המילה הספציפית מהמצב הקודם, נעשה זאת ע"י הכפלה של ההסתברות לעבור מחלק הדיבר הקודם לחלק הדיבר הנוכחי עם ההסתברות לקבל מילה זה בחלק הדיבר הנוכחי (לדוגמה: במעבר מ- $START$ ל- V עם המילה 'אני' נכפיל את ההסתברות לעבור מ- $START$ ל- V בהסתברות לקבל 'אני' בחלק הדיבר (Verb)). בכל מעבר ניקח את המעבר עם ההסתברות המקסימלית מבין כל המעברים ממצב אחד לאחר (מסומן בחץ אדום). כך ניקח את כל המעברים המקסימליים וניצור מסלול מקודקוד ההתחלה לקודקוד הסיום. זהו אלגוריתם פולינומי ((אורך המשפט) * (חלקי הדיבר בריבוע) $= k^2 T$) והוא יעיל יותר ממרקוב.

Transition	D	END	N	START	V
D	0.1	0	0.1	0.4	0.4
END	0	1	0.2	0	0.1
N	0.8	0	0.1	0.3	0.3
V	0.1	0	0.6	0.3	0.2

Emission	D	END	N	START	V
<END>	0	1	0	0	0
<START>	0	0	0	1	0
dog	0.1	0	0.8	0	0.05
in	0.3	0	0	0	0
park	0	0	0.1	0	0.1
the	0.6	0	0.1	0	0.05
walked	0	0	0	0	0.8



```
from nltk.corpus import wordnet as wn

def is_noun(tag):
    return tag in ['NN', 'NNS', 'NNP', 'NNPS']

def is_verb(tag):
    return tag in ['VB', 'VBD', 'VBG', 'VBN', 'VBP', 'VBZ']

def is_adverb(tag):
    return tag in ['RB', 'RBR', 'RBS']

def is_adjective(tag):
    return tag in ['JJ', 'JJR', 'JJS']

def penn2wn(tag):
    if is_adjective(tag):
        return wn.ADJ
    elif is_noun(tag):
        return wn.NOUN
    elif is_adverb(tag):
        return wn.ADV
    elif is_verb(tag):
        return wn.VERB
    return wn.NOUN
```

:Lemmatization

דומה מאוד ל Stemming אך מורכב יותר (כבד ואיטי אך מביא תוצאות טובות יותר).
Lemmatization מוצא את גרעין המילה עבור כל מילה במשפט בצורה חכמה יותר כיוון שהוא "מכיר" את המילים, מילים כמו going ו-went מתורגמות ל-go. Lemmatizer לוקח בחשבון את חלק הדיבר של המילה (ולכן יותר איטי אך מביא תוצאות טובות יותר).
ה-WordNetLemmatizer ב-NLTK משתמש ברשימה קצרה יותר של תגיות חלקי דיבור מאשר Penn TreeBank.

```
from nltk.tokenize import word_tokenize
from nltk import pos_tag
from nltk.stem.wordnet import WordNetLemmatizer
lzm = WordNetLemmatizer()
my_text = "Whoever eats many cookies is regretting doing so"
lemmed = []
for (word,pos) in nltk.pos_tag(word_tokenize(my_text)):
    lemmed.append(lzm.lemmatize(word,penn2wn(pos)))
>>> lemmed
['Whoever', 'eat', 'many', 'cooky', 'be', 'regret', 'do', 'so']
```

:Chunking

שיטה לחלוקת משפט (או קבוצה של משפטים), לנתחים שונים (קבוצות). Chunking משתמש בביטויים רגולריים (תבניות) בחלקי הדיבור.

- NP: {< DT >? < JJ >? < NN >}
- נתח ה מורכב determiner אופציונלי (0 או 1), אפס או יותר שמות תואר (adjective) ושם עצם (noun). לדוגמה: "The nice big boy".
- NounList: {(< DT >? < NN >? < , >?) + < CC > < DT >? < NN >? }
- כאשר < NN >? זה אחד או יותר סוגים שונים של Nouns עם determiner אופציונלי לפני ופסיק אופציונלי אחרי (?). פירושו שיכול להיות סוגים שונים של Nouns כמו NN, NNP, NNS. בנוסף, < CC > היא מילת קישור and/or, כאשר + אומר שהביטוי שלפני חייב להופיע לפחות פעם אחת.
- לדוגמה: "Sara, John, Tom, the girl and the bat". דוגמה נוספת: "Dogs or cats".

ניתן להגדיר ביטוי מסוים כ"חוק" כאשר כל ביטוי מגדיר את התבנית של המשפט ומאיזה חלקי דיבור הוא צריך להיות מורכב. ברגע שנפעיל את האלגוריתם נקבל חלוקה של המשפט לחלקים המגדירים את הביטוי שלנו.

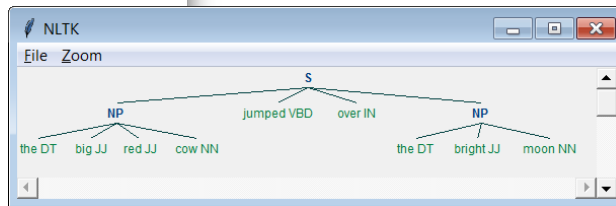
נשתמש בזה על מנת להבין את מבנה המשפט וכך נדע לענות בהתאם לחלקי המשפט.

נניח שהמשפט הוא "הילד דני אכל מהעוגה", באמצעות החלוקה נוכל לענות על השאלה 'מי אכל?' "הילד דני".

בעמוד הבא ניתן לראות דוגמאות ל-Chunking ב-NLTK.

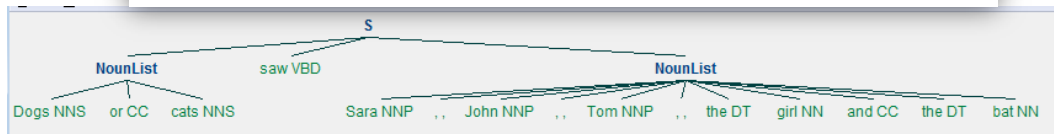
דוגמה 1:

```
>>> my_text = "the big red cow jumped over the bright moon"
>>> tagged = nltk.pos_tag(nltk.tokenize.word_tokenize(my_text))
>>> grammar = "NP: {<DT>?<JJ>*<NN>}"
>>> cp = nltk.RegexpParser(grammar)
>>> result = cp.parse(tagged)
>>> print(result)
(S
 (NP the/DT big/JJ red/JJ cow/NN)
 jumped/VBD
 over/IN
 (NP the/DT bright/JJ moon/NN))
>>> result.draw()
```



דוגמה 2:

```
my_text = "Dogs or cats saw Sara, John, Tom, the girl and the bat"
grammar = "NounList: {( <DT>?<NN.?><,?>)+<CC><DT>?<NN.?>}"
```

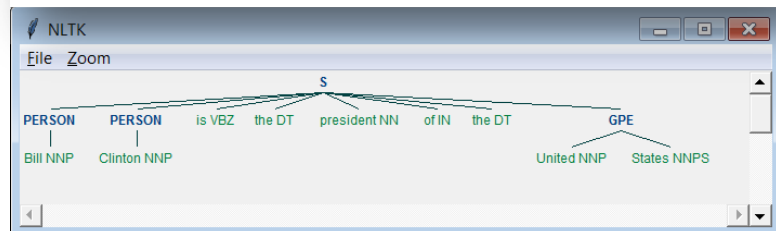


דוגמה 3:

Named Entity Recognition (NER) - סוג נוסף של chunk הפועל לזיהוי שמות עצם פרטיים וקטלוגם (למשל שמות פרטיים, ערים וכו').

geo = Geographical Entity
org = Organization
per = Person
gpe = Geopolitical Entity
tim = Time indicator
art = Artifact
eve = Event
nat = Natural Phenomenon

```
>>> result = nltk.ne_chunk(nltk.pos_tag(nltk.word_tokenize("Bill Clinton is the president of the United States")))
>>> result.draw()
```



Bi Grams

"I did it you did it you did it"
 We get the following bi-grams count:
 [(I, did), 1], [(did, it), 3], [(it, you), 2], [(you, did), 2]

בעיבוד שפות טבעיות נרצה לחלק את המשפט לצמדי מילים. במקום להתבונן בכל מילה בנפרד נרצה להתבונן בצמדי מילים. נפצל את המשפט לזוגות מילים הסמוכות זו לזו. כך נוכל להבין בצורה טובה יותר את המשפט ויהיה לנו קל יותר למפות לחלקי דיבור.

N Grams

Bi Gram זהו מקרה פרטי עבור $n = 2$, כלומר בכל פעם נתבונן על N מילים יחד. שימוש נפוץ בN Grams הוא בהשלמות משפטים.

נפצל את הדאטה לשלוש של מילים, נקרא את 2 המילים שבטקסט ונסה להבין מה היא המילה השלישית. כלומר, נחפש את השלוש המכילות את 2 המילים האלו ומתוכן נדגום באקראי אחת מהן והמילה השלישית היא המילה שנשלים. (יכולנו גם לקחת בעלת ההסתברות הגבוהה ביותר. אך בצורה זו נקבל את אותה מילה בכל פעם ולכן נעדיף רנדומליות).

לדוגמא: נניח שהמשפט הוא "היום יום חמישי הגיע הזמן", ונרצה לדעת מה היא המילה הבאה. נתבונן ב2 המילים האחרונות, "הגיע הזמן" ונחפש את כל השלוש ב Data set שהמילה הראשונה בהם היא "הגיע" והמילה השנייה היא "הזמן". מתוך כל השלוש האלו ניקח את כל המילים האחרונות שלהן ונגריל אחת מהן באקראי.

```
import nltk
import urllib
from random import randint
paragraph_len = 100
all_text = urllib.request.urlopen("https://s3.amazonaws.com/text-datasets/nietzsche.txt").read().decode("utf-8")
tokens = nltk.word_tokenize(all_text)
my_grams = list(nltk.ngrams(tokens,3))
sentence = ["It", "is"]
for i in range(paragraph_len):
    options = []
    for trig in my_grams:
        if trig[0].lower() == sentence[len(sentence)-2].lower() and trig[1].lower() == sentence[len(sentence)-1].lower():
            options.append(trig[2])
    if len(options) > 0:
        sentence.append(options[randint(0, len(options)-1)])
    print(" ".join(sentence))
```

Context-Free Grammar (CFG)

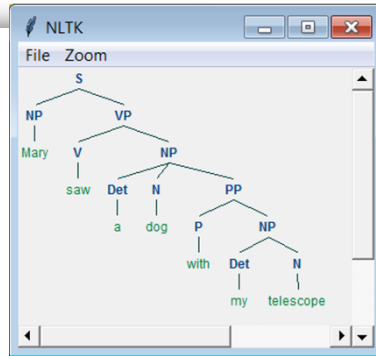
דקדוק חופשי הקשר בנוי מארבעה חלקים:

- T - אוצר מילים סופי, טרמינלי (קבוצת כל המילים המוגדרות בשפה).
- V - אוצר מילים לא סופי, לא טרמינלי.
- P - סט חוקים מהצורה $a \rightarrow b$ (כאשר $a \in V$ ו- b הוא רצף של תווים השייכים ל- $T \cup V$).
- S - סמל ההתחלה (שייך ל- V).

נתחיל מ- S ונעבוד לפי חוקי הגזירה ב- P עד שנקבל מילה סופית (טרמינלית)

נרצה למצוא את העץ הסביר ביותר שפורש את המשפט. כלומר, מהו רצף הגזירות הסביר ביותר שבאמצעותו נגיע למשפט הזה וכך נוכל להבין מה המשמעות של המשפט (כאשר יש לו כמה משמעויות).

```
>>> grammar1 = nltk.CFG.fromstring("""
S -> NP VP
VP -> V NP | V NP PP
PP -> P NP
V -> "saw" | "ate" | "walked"
NP -> "John" | "Mary" | "Bob" | Det N | Det N PP
Det -> "a" | "an" | "the" | "my"
N -> "man" | "dog" | "cat" | "telescope" | "park"
P -> "in" | "on" | "by" | "with" | ""
""")
>>> sentence = "Mary saw Bob".split()
>>> rd_parser = nltk.RecursiveDescentParser(grammar1)
>>> print(list(rd_parser.parse(sentence))[0])
(S (NP Mary) (VP (V saw) (NP Bob)))
>>> print(list(rd_parser.parse("Mary saw a dog with my telescope".split()))[0])
(S (NP Mary) (VP (V saw)
  (NP (Det a) (N dog) (PP (P with) (NP (Det my) (N telescope))))))
>>> (list(rd_parser.parse("Mary saw a dog with my telescope".split()))[0]).draw()
```



מצד שמאל ניתן לראות שימוש ב-CFG Parser ב-NLTK.

המשפט שנרצה למצוא את חלקי הדיבר שלו הוא:

"מרי ראתה כלב עם הטלסקופ שלי".

ניתן לפענח את המשפט ב-2 דרכים שונות.

האם הכוונה היא שמרי ראתה את הכלב באמצעות

הטלסקופ או שמרי ראתה את הכלב שהטלסקופ שלי אצלו.

נשים לב שה-Parser מחזיר רשימה של כל האפשרויות ואנו

מתבוננים באפשרות הראשונה בלבד (זו שנמצאת במקום

ה-0) כיוון שהיא הסבירה ביותר (בעלת ההסתברות הגבוהה

ביותר). לפעמים נרצה להתבונן בכל האפשרויות שחזרו מה-Parser וכך

נוכל לראות את כל המשמעויות שיש למשפט.

CYK Algorithm

אלגוריתם תכנות דינמי שמטרתו היא מציאת העץ היוצר בדקדוק חסר הקשר (עץ החוקים שיצר את המשפט). סיבוכיות - $O(n^3 |G|)$ כאשר n מייצגת את אורך המשפט G מייצג את גודל קבוצת המשתנים הלא טרמינליים.

לפני הרצת האלגוריתם נמיר את כל החוקים לצורת CNF. כל כללי היצירה הם מסמל לא טרמינלי לסמל טרמינלי יחיד, או מסמל לא טרמינלי ל-2

סמלים לא טרמינליים. פורמלית, כל הכללים צריכים להראות באחת מ-2 הצורות הבאות: $A \rightarrow BC$ or $A \rightarrow a$.

לאחר ההמרה נקבל סט חוקים בצורה הנורמלית של חומסקי ואז נמצא את חלקי הדיבר המרכיבים את המשפט הנתון.

דוגמה: נרצה לקבוע האם המשפט "she eats fish with a fork" הוא בדקדוק או לא (הדקדוק מופיע בתמונה מימין).

לשם כך ניצור טבלה בצורה הבאה:

בשורה התחתונה נכתוב את המשפט.

בשורה שמעליה נכתוב עבור כל מילה מאיזה גזירה ניתן לקבל אותה.

בשורה הבאה נתבונן בכל 2 תאים סמוכים בשורה שמתחת ונכתוב את המשתנה שממנו ניתן לקבל את המשתנים שרשומים

בתאים. במידה ולא קיים חוק גזירה שמחזיר לנו את המשתנים שב-2 התאים, נשאיר את התא ריק.

בשורה הבאה נתבונן ברצף של שלושה מילים נמשיך כך עד שנמלא את כל הטבלה.

במידה ומילאנו את השורה העליונה, המשפט בדקדוק.

דוגמה להרצה מלאה של האלגוריתם: <https://www.youtube.com/watch?v=VTH1k-xiswM>

```
S -> NP VP
VP -> VP PP
VP -> V NP
VP -> eats
PP -> P NP
NP -> Det N
NP -> she
V -> eats
P -> with
N -> fish
N -> fork
Det -> a
NP -> fish
```

S					
---	VP				
---	---	---			
S	---	---	PP		
S	VP	---	---	NP	
NP	V, VP	N, NP	P	Det	N
she	eats	fish	with	a	fork

כדי למצוא את עץ היצירה (Parse Tree), כל node חייב להכיל non-terminal שיצביע לתת עץ (כך שהמילים מתחתיו יצביעו לאיך הגענו אליו).

Probabilistic Context Free Grammar (PCFG)

דקדוק חופשי בהקשר הסתברותי, המטרה היא להשתמש בGrammar כדי להחליט איזו משמעות היא הסבירה ביותר.
לכל כלל גזירה יש הסתברות כך שסכום ההסתברויות מ non-terminal מסוים צריך להיות 1.
העץ תמיד מתחיל מ S וממנו ממשיכים לגזירות שלו והלאה עד שמגיעים למילים עצמן.

דוגמא 1: "Jack saw telescope"

קיבלנו את האפשרות הסבירה ביותר לחלוקת המילה POS.

מה שמעניין אותנו זה שבמידה ויש כמה אפשרויות (כמה משמעויות) לנתח את המשפט נקבל את האפשרות בעלת ההסתברות הגבוהה ביותר.

דוגמא 2: "They ate spaghetti with meatballs"

קיבלנו 2 עצי גזירה, כלומר 2 משמעויות שונות למשפט. ניתן לראות שיש עץ אחד עם הסתברות גבוהה יותר (0.0108), וזה אומר שכנראה זו המשמעות הנכונה יותר של המשפט.
פירוש 1: (עם ההסתברות הגבוהה) "הם אכלו ספגטי עם (בעזרת) כדורי בשר."

פירוש 2: "הם אכלו ספגטי עם כדורי בשר"

נשים לב שבדוגמה זו המשמעות הסבירה יותר ע"פ האלגוריתם פחות מדויקת. לכן בדרך"כ כדי לקבל דיוק יותר טוב נעבוד עם dataset גדול יותר. במקום לכתוב בעצמנו את הכללים.

CoreNLP

ספריית קוד פתוח שכתובה בJava, אך ניתן לקרוא אותה בשפות תכנות שונות כולל Python. CoreNLP חזק יותר מ-NLTK.
בCoreNLP אימנו את הדאטה על dataset גדול יותר, ובנו את החוקים (Grammar) מדויקת יותר.
לCoreNLP יש את התכונות הבאות: ניתוח תלות, Grammar גדול (ניתן לטעון Grammar מסויים ממקור חיצוני) ותמיכה בCoreference Resolution.
מכילה 2 מרכיבים עיקריים: (1)קוד הבסיס. (2)מודלים של השפה - חלק זה מרכיב את רוב המשקל של הספרייה.

Dependency Parsing (ניתוח תלות) – ספרייה שבאמצעותה אפשר להבין על מי פועלת כל מילה במשפט כאשר כל מילה מסתמכת על אחת מהמילים שלפניה. מהיר יותר מconsistency parsing.

Co-reference Resolution – ספרייה שבאמצעותה ניתן להבין לאיזה מילים מילה מסויימת מתייחסת.

לדוגמה, עבור המשפט "The bus was full. It drove very fast" נרצה לדעת למי המילה It מתייחסת.

Sentiment Analysis – ניתוח סמנטי, נרצה להבין האם משפט מסויים הוא חיובי או שלילי.

למשל נרצה לדעת האם ביקורת מסויימת היא חיובית או שלילית.

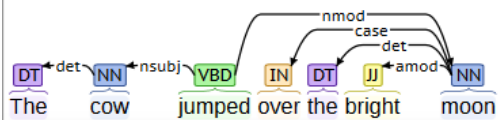
בתמונה משמאל ניתן לראות ביצוע ניתוח סמנטי ע"י NLTK.

יש מספר דרכים לבצע ניתוח סמנטי, וביניהן:

- **Bag of words** – נשתמש בNaive Base, זה יספור כמה מילים חיוביות/שליליות יש. לא תמיד עובד כי השיטה לא לוקחת בחשבון את הסדר של המילים אלא רק את המילים עצמן. ולכן ינתח בצורה זאת את "it wasn't good, it was actually bad" ואת "it wasn't bad, it was actually good". למרות שלשני המשפטים משמעות הפוכה (האחת חיובית והשנייה שלילית).
- נתבונן על מילים שנמצאות אחרי מילת שלילה (כמו not) במידה וזו מילה חיובית אז כנראה המשפט הוא שלילי ולהפך. גם כאן משפטים כמו "not as good" יהיו בעייתיים לפענוח.
- ע"י שימוש בN Grams.
- שימוש בשיטות נוספות של למידה עמוקה.

```
>>> grammar = nltk.PCFG.fromstring("""
S -> NP VP      [1.0]
VP -> TV NP      [0.4]
VP -> IV         [0.3]
VP -> DatV NP NP  [0.3]
TV -> 'saw'      [1.0]
IV -> 'ate'       [1.0]
DatV -> 'gave'    [1.0]
NP -> 'telescopes' [0.8]
NP -> 'Jack'      [0.2] """)
>>> viterbi_parser = nltk.ViterbiParser(grammar)
>>> for tree in viterbi_parser.parse(['Jack', 'saw', 'telescopes']):
...     print(tree)
(S (NP Jack) (VP (TV saw) (NP telescopes))) (p=0.064)
```

```
>>> grammar = nltk.PCFG.fromstring("""
S -> NP VP      [1.0]
VP -> VBP NP      [0.50]
VP -> VBP NP PP   [0.50]
TV -> 'saw'       [1.0]
VBP -> 'ate'       [1.0]
NP -> NP PP        [0.3]
PP -> IN NP        [1.0]
NP -> 'spaghetti'  [0.2]
NP -> 'they'       [0.3]
NP -> 'meatballs'  [0.2]
IN -> 'with'       [1.0]
""")
(S (NP they) (VP (VBP ate) (NP spaghetti) (PP (IN with) (NP meatballs)))) (p=0.0108)
(S (NP they) (VP (VBP ate) (NP (NP spaghetti) (PP (IN with) (NP meatballs)))))) (p=0.00288)
""")
```



```
>>>from nltk.sentiment.vader import
SentimentIntensityAnalyzer
>>>sna = SentimentIntensityAnalyzer()
>>>sna.polarity_scores("The movie was great!")
{'neu': 0.406, 'neg': 0.0, 'compound': 0.6588, 'pos': 0.594}
>>>sna.polarity_scores("I liked the book, especially the
ending.")
{'neu': 0.641, 'neg': 0.0, 'compound': 0.4215, 'pos': 0.359}
>>>sna.polarity_scores("The staff were nice, but the food was
terrible.")
{'neu': 0.536, 'neg': 0.318, 'compound': -0.5023, 'pos': 0.146}
```

Compound is a normalized value between -1 (negative) to +1 (positive).