

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

דוגמאות לבעיות העומדות בפני עצמן הבנת שפה טבנית:

למשמעותים "נתנו לקובים את התפוזים משם היו רעבים", ו"נתנו לקובים את התפוזים משם היו רוקבים", יש לכואורה מבנה תחבירי זהה, אף כי למעשה אחד מהם המילול "הם" מתייחס לקובים, ואילו באחר לתפוזים. אי אפשר להבין את המשפט כראוי ללא היכרות עם המאפיינים והתכוונות המתיחסים לקובים ולתפוזים.

בשפות רבות קשה לומר את מרכיבי המשפט ולסמן את היחסים ביניהם באמצעות ניתוח פשוט של המבנה התחבירי. למשל, יש שפות שקשה לדעת בהן לאייזה שם עצם מתייחס שם התואר, ויש שפות בהן אין סימן מורפולוגי לחילוק הדבר.

כדי לפתרו את הבעיה האלו, הציעו כמה בלשנים וחוקרי אינטלקטואליות טבנית לעשות שימוש בשפה מלאכותית, שתוכל לבטא את כל הדקדוקיות והעומק של השפות הטבעיות שאנו מכירים, אך עם זאת שתהיה בעלי תחביר וחוקק כתיב עיקריים מבחינה לוגית או מתמטית, כדי להסיר כל עמימות או רב-משמעות הנובעת מבניה המשפט.

כלומר, הבעיה העיקרית בNLP היא זו ממשמעות. (יש כמה ממשמעות ואנו רוצים לסwoג את המשמעות הנכונה של המשפט).

מבחן טירוגן:

מבחן טירוגן הוא כינוי למבחן שהציג אלן טירוגן בשנת 1950, כמזה אפשרי למידה שבה יש למוכנה כלשהי אינטלקטואלית. המבחן נעשה בדרך הבאה: חוקר מקיים דיאלוג בשפה טבנית עם שני גורמים סמויים מעינויו, האחד אדם והשני מכונה. אם החוקר יוכל לקבע בביטחון מי האדםומי המכונה, אז המכונה עברה בהצלחה את המבחן.

```
>>> 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."]
```

ספריה לעיבוד שפות טבניות בPython.

- כאשר אנו רוצים לעבוד על משפט, הפעולה הפטורה ביותר היא Tokenization לשעות חסרים, מודל בסיסי זה אחראי לפצל את המשפט למלילים תוך שימוש במפרדים. הרבה פעמים נרצה לפצל מסמר 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']
```

- תהליך של צמצום מילה לגזע (שורש) המיליה שלה המוצמד לשינוי. Stemming חשוב בהבנת השפה הטבעית (UTN) ובעיבוד השפה הטבנית (NLP). פעמים רבות מתייחס למיללים שונים בכתיבתן צהות (מיללים שונים עם אותו הבסיס. למשל נרצה להציג walks ולולוג walking). ניקח מילה ונשלוף ממנה את הבסיס וכן נגיע לאותה מילה הדומה לה בשורש. תחילה, נצטרך להפריד את המילה ע"י שימוש בtokenization אחר והstemmers מעבד כל מילה בנפרד.

בNLP 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 חלקו של דבר הוא תהליך פופולרי לעיבוד שפה טבנית המקטלג מילים בטקסט בהתאם עם חלק מסויים בדיבור, בהתאם להגדרת המילה והקשר שלה. זה מודל הנומן לכל מילה במשפט אונטזיה (פירוש) ותחבירית לחילוק הדיבור הנכונים במשפט. כגון: פועל, שם עצם, תואר ועוד. באמצעות מודל זה נקבל המון מידע על המשפט ונוכל למצוא מילים שמשמעותם אונטנו ולהסתמך על החוקים שסבירין.

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, dummo, gr8, univeristy

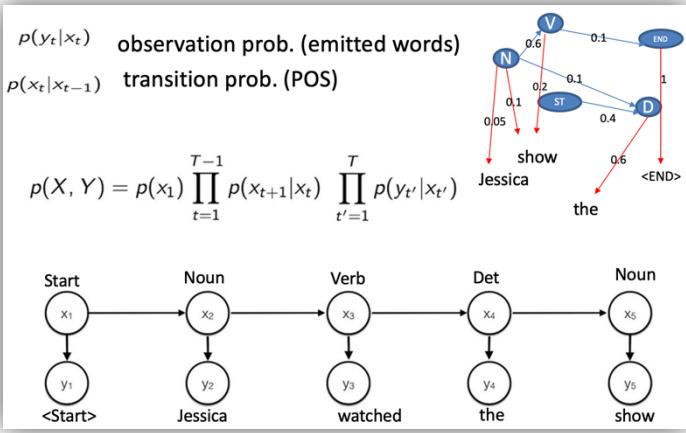
POS באמצעות (HMM) Hidden Markov Models

בහינתן רצף (מלילים, אותיות, משפטים וכו'), HMM מחשב את ההסתברות על פni רצף של תוויות ומונבא את רצף התוויות הטוב ביותר.

מודל זה מtabסס על ההנחה שיש קשר בין כל חלק דיבר במשפט לחלק הקודם לו. נרצה למצוא תיוג (קטלוג) של חלקי המשפט כדי שנתקבל את המשפט הנutan. נחשב מה הסיכוי למעבר בין כל חלק במשפט, ואת הסיכוי שהחלק דיבר כלשהו שייר למלילה הנutan. נבחר את הסידור בעל ההסתברות הגבוהה ביותר. כמו, בהינתן משפט נרצה לראות מה סדרת חלקי הדיבר שהיכי סביר שתנתן לנו את משפט זה. הנחת מרכיבים היא שהקשר בין חלקי

תלי אך ורק בחלק הדיבר של המילה הקודמת.

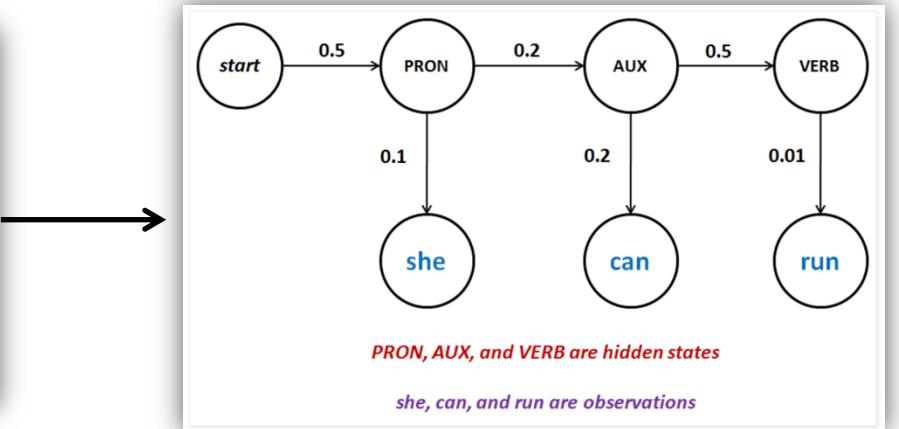
הז נתן לנו והוא מייצג את המילים במשפט הנutan, נרצה למצוא X (רצף חלקי הדיבר המרכיבים את המשפט) בהנחה שהאים לא מושפעים מהז הקודם אלא רק מהז הקודם לו.



למעשה, זה מעין אוטומט. בכל פעם נמצאים במצב וממנו עבריםים למצב הבא, כל מצב הוא חלק דיבר (x_i) וכל מצב מוביל לנו מילה (y_i) בהסתברות מסוימת. נשים לב שהמסלול ממנו הגיעו אל המצב לא מעניין אלא רק המצב הקודם ממנו הגיעו. בכל מעבר נחשב את ההסתברות לעבר במצב הבא בתור. נרצה למצוא מסלול מנצח ההתחלה (start) למצב הסיום (end) על הסתברות הגבוהה ביותר. כדי לקבל את ההסתברות לכל מעבר ממצב למצב, נצטרך לעבור על הdata set (מאגר נתונים הנקרה Penn TreeBank שמכיל המון משפטים ואת חלקי הדיבר של כל אחת מהמילות במשפט), לבדוק לאיזה דיבר נבעור בהסתברות הגבוהה ביותר (לדוגמא נניח שהרשות המחוקים במאגר אחורי דיבר מסווג מופיע דיבר מסווג verb) ולבדק בהינתן חלק דיבר מה הסיכוי לקבל כל מילה במשפט.

נניח שהמשפט הוא "אני אוהב בינה מלאכותית" ועלינו להקצות תג POS לכל מילה. ברור שתגיות ה-POS לכל מילה הן "כינוי" (PRP-Pronoun), פועל (VBP-Verb), שם תואר (Adjective) ושם עצם (NN-Noun) בהתאם. כדי לחשב את הסתברות לתגיות, עלינו לדעת תחילת מה הסיכוי שהמילה "כינוי" יגע "פועל", לאחר מכן שם תואר ולבסוף שם עצם. שנייה, עלינו לדעת מה הסיכוי שהמילה "אני" תהיה כינוי, המילה "אהוב" תהיה פועל, המילה "בינה" תהיה שם תואר, והמילה "מלאכותית" תהיה שם עצם. הסתברויות אלו נקבעות הסתברויות פליטה. ניתן לתאר את הסתברויות אלו באמצעות מטריצות.

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



POS באמצעות Viterbi Algorithm

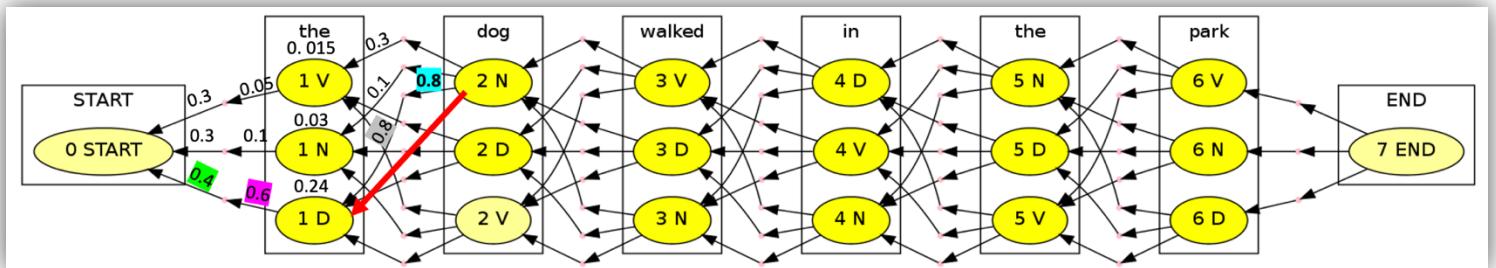
אלגוריתם תכנון דינמי למציאת רצף המצביעים החבויים (חלקי הדיבר) הסביר ביותר, שתוצאותיו היא רצף תצפויות נתון. בין שימושיו הנפוצים - מציאת רצף המצביעים בהינתן מודל מרכיבים חבוי, פיענוח קוד קובולוציה ועוד.

יתרונות הבולטים של האלגוריתם הוא הורדת סיבוכיות המציאה של הרצף הסביר ביותר - מהמיושש הנאי (בעל סיבוכיות מעריכית בגודל הרצף) למימוש היעיל של האלגוריתם (בעל סיבוכיות ליניארית בגודל הרצף).

נתונות טבלאות עם כל הסתברויות. כאשר סכום כל עמודה הוא 1. ישן שתי טבלאות: אחת מכילה את הסתברויות למעבר ממצב אחד (המצב בשורה) לאחר (המצב בעמודה), והשנייה מכילה הסתברויות מותניות, בהינתן חלק דיבר מההסתברות לקבל כל אחת מהמילות במשפט. בכל קודקוד נחשב את הסתברות להגיע לקודקוד זה עם המילה הספציפית מהמצב הקודם, נעשה זאת ע"י הכפלה של הסתברות לעבור מחלק הדיבר הקודם לחולק הדיבר הנוכחי עם הסתברות לקבל מילה זה בחולק הדיבר הנוכחי (לדוגמא: בעבור מ' START ל' The אנו נכפיל את הסתברות לעבור מ' START ל' The בהסתברות לקבל Verb בחלק הדיבר Verb). בכל מעבר ניקח את המעבר עם הסתברות המקסימלית מבין כל המעברים ממצב אחד לאחר (מסומן בחץ אדום). כך ניקח את כל המעברים המקסימליים וויצור מסלול מקודקוד ההתחלה לקודקוד הסיום. זהו אלגוריתם פולינומי ((אורך המשפט) * (חלקי המשפט) $= T^2$) והוא יעיל יותר מרקורסיבי.

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

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
```

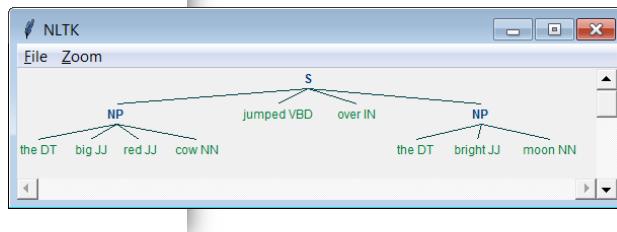
```
from nltk.tokenize import word_tokenize
from nltk import pos_tag
from nltk.stem.wordnet import WordNetLemmatizer
lzs = WordNetLemmatizer()
my_text = "Whoever eats many cookies is regretting
doing so"
lemed = []
for (word, pos) in nltk.pos_tag(word_tokenize(my_text)):
    lemed.append(lzs.lemmatize(word,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

- שיטה לחלוקת משפט (או קבוצה של משפטים), לנתחים שונים (קבוצות). Chunking משתמש בביוטים רגולריים (מבנה) בחלוקת הדיבור.
 - נתח המורכב מdeterminer אופציונלי (0 או 1), אפס או יותר שמות תואר (adjective) ושם עצם (noun). לדוגמה: "The nice big boy".
 - כasher * < ? .NN > זה אחד או יותר סוגים שונים של Nouns עם determiner אופציונלי לפני ופסיק אופציונלי אחריו (?). פירושו שיכל להיות סוגים שונים של Nouns כמו NNS,NNP,NN. בנוסף, <CC> היא מילת קישור or/and, כאשר + אומר שהביטוי שלפני חיב להופיע לפחות פעמי אפס

לדוגמה: "Dogs or cats". "Sara, John, Tom, the girl and the bat" ניתן להגדיר ביטוי מסוים כ"חוק" כאשר כל ביטוי מגדיר את התכנית של המשפט ומאיiza חילוקה בין הפלטים. האלגוריתם נקבע חילוקה של המשפט לחוקים המגדירים את הביטויים השונים. נשתמש בזיה על מנת להבין את מבנה המשפט וכן נדע לענות בהתאם לחוקי המשפט. נניח שהמשפט הוא "הילד דני אכל מהעוגה", באמצעות החילוקה נוכל לענות על השאלה 'מי' רעומד הרא ויתן לראות דוגמאות לחקלאות Chucking NLTCK.

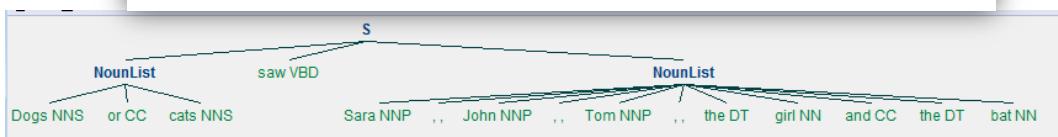
דוגמה 1:

```
>>> 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.?>}"
```

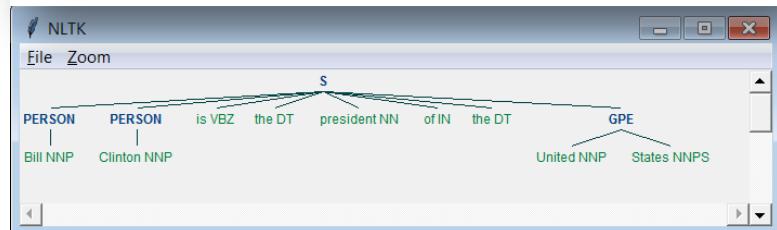


דוגמה 3:

סוג נוסף של chunk הפעול ליזיהו שמות עצם פרטיים וקיטלוגם (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

בעיבוד שפות טבעיות נרצה לחלק את המשפט לצמדים מיילים. כלומר, במקומות להתבונן בכל מילה בנפרד נרצה להתבונן בצמדים מיילים. נפצל את המשפט לזוגות מיילים סמוכות זו לזו. כך נוכל להבין בצורה טוביה יותר את המשפט ויהיה לנו קל יותר למפות לחלק דיבור.

"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)]

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

נפצל את הדטה לשששות של מילים, נקרא את 2 המילים שבתקסס ונססה להבין מה היא המילה השלישית. כלומר, נחפש את השששות המובילות את 2 המילים האלה ומתקן דוגם באקראי אחת מהן והמילה השלישית היא המילה שנשלה. (יכלנו גם לחתה עליה הסתברות הגבואה ביותר. אך במקרה זו קיבל את אותה מילה בכל פעם ולכן נעדיף רנדומליות).

לדוגמא: נניח שהמשפט הוא "היום יומם חמישי הגיע הזמן", ונרצה לדעת מה היא המילה הבאה. נתבונן ב 2 המילים האחרונות, "הגיע הזמן" ונחפש את כל השששות בData set שהמילה הראשונה בהם היא "הגיע" והמילה השנייה היא "זמן". מתוך כל השששות האלה ניקח את כל המילים האחרונות שלן 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))
```


Probabilistic Context Free Grammar (PCFG)

```
>>> 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)
```

דקוד חופשי בהקשר הסתברותי, המטרת היא להשתמש בGrammar כדי להתחיל איזו משמעות היא הסבירה ביותר. לכל גזירה יש הסתברות כך שסכום ההסתברויות מ-non-terminal מסוים צריך להיות 1. העץ תמיד מתחילה מ S וממנו ממשיכים לגזרות שלו והלאה עד שmaguis למלים עצמן.

דוגמה 1: "Jack saw telescope".

קיבלנו את האפשרות הסבירה ביותר לחלוקת המילה לPOS. מה שמעניין אותנו זה שבמקרה יש כמה אפשרויות (כמה משמעותיות) לנתח את המשפט נקבל את האפשרות בעלת ההסתברות הגבוהה ביותר.

דוגמה 2: "They ate spaghetti with meatballs".

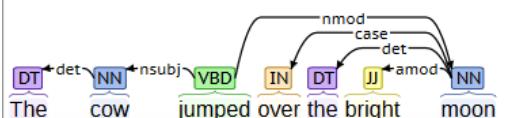
קיבלנו 2 עצי גזירה, כלומר 2 משמעותיות שונות למשפט. ניתן לראות שיש עץ אחד עם ההסתברות הגבוהה יותר (0.0108), וזה אומר שכנראה זו המשמעות הנכונה יותר של המשפט.

פירוש 1: (עם ההסתברות הגבוהה) "הם אכלו ספגטי עם (בעזרת) כדור בשר".

פירוש 2: "הם אכלו ספגטי עם כדור בשר" נשים לב שבדוגמה זו המשמעות הסבירה יותר ע"פ האלגוריתם פחות מדויקת. לכן בדרכ' כדי לקבל דיוק יותר טוב נעבד עם dataset גדול יותר. במקום לכתוב עצמנו את הכללים.

CoreNLP

ספרית קוד פתוח שכתובה בJava, אך ניתן לקרוא אותה בשפות תכנות שונות כולל CoreNLP Python חזק יותר מ-MT. CoreNLP מפעיל את הדאטה על dataset גדול יותר, ובנו את החוקים (Grammar) מדויקת יותר. CoreNLP יש את התכונות הבאות: ניתוח תלות, Grammar גדול (ניתן לטעון Grammar מסוים מקור חיצוני) ותמייה בחסימה. מכילה 2 מרכיבים עיקריים: (1) קוד הבסיס. (2) מודלים של השפה - חלק זה מרכיב את רוב המשקל של הספרייה.



Dependency Parsing (ניתוח תלות) – ספרייה שבאמת עוזרת להבין על מי פועלת כל מילה במשפט כאשר כל מילה מסתמכת על אחת מהמילות שלפניהם. מהיר יותר מוגדר consistency parsing.

Co-reference Resolution – ספרייה שבאמת עוזרת להבין לאיזה מילים מילה מסוימת 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 מסוים הוא חיובי או שלילי. למשל נרצה לדעת האם ביקורת מסוימת היא חיובית או שלילית.

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

יש מספר דרכים לבצע ניתוח סמנטי, ובינהן:

- **words – Bag of words** – השתמש Baseline, Naive Bayes, זה יספר כמה מילים חיוביות/שליליות יש.

לא תמיד עובד כי השיטה לא לוקחת בחשבון את הסדר של המילים אלא רק את המילים עצמן.

ולכן ניתן בצורה זהה את "it wasn't bad, it was actually good" ואת "it wasn't good, it was actually bad".

למרות של שני המשפטים משמעויות הופכה (האחת חיובית והשנייה שלילית).

נתבונן על מילים שנמצאות אחרי מילת שלילה (כמו not) במשמעותה וזו מילה חיובית אז כנראה המשפט הוא שלילי ולהפוך.

גם כאן משפטים כמו "good as not" יהיו בעיתויים לפחות.

ע"י שימוש ב-Grams N.

שימוש בשיטות נוספות של למידה عمוקה.