# הקדמה לעיבוד שפות טבעיות

## הגדרה

עיבוד שפה טבעית (Natural Language Processing - NLP) הוא תחום החוקר את הבעיות הקשורות לעיבוד ומניפולציה של שפה טבעית, והבנה של שפה טבעית על מנת לגרום למחשבים "להבין" ולנתח דברים שנאמרים או נכתבים בשפות אנושיות. זהו תת-תחום של בינה מלאכותית ובלשנות וחלק מהשיטות בו משלבות למידת מכונה ולמידה עמוקה.

## קשיים בעיבוד שפות טבעיות

**הפרדת משפט למילים -** בשפה מדוברת אין בדרך כלל הפרדה ברורה בין מילים; רוב הבלשנים מניחים כי הדוברים מזהים מילים ורכיבים תחביריים אחרים במשפט על-פי רמזים פרוזודיים (בעיקר הטעמה) ואינטונציה. גם בשפה הכתובה אין בהכרח הפרדה בין המילים. בשפות כמו סינית אין רווחים בין המלים, ובווייטנאמית מסמנים רווח בין הברות ולא בין מילים. גם בשפות אירופיות כללי ההפרדה בין המילים משתנים משפה לשפה (למשל, באנגלית משתמשים ברווח יותר מאשר בגרמנית).

**רב-משמעות לקסיקלית -** למלים רבות יש יותר ממשמעות אחת; מציאת המשמעות הנכונה מחייבת התחשבות בהקשר וב"ידע העולם". רב-משמעות לקסיקלית עשויה להביא ליצירת צירופים שנראים אבסורדיים, אך בעצם הם מובנים לדוברי השפה, למשל: "סבון נטול סבון" (סבון במשמעות "תכשיר רחצה" לעומת סבון במשמעות "סוג של חומר").פתרון רב משמעות לקסיקלית חשוב מאד לזיהוי אוטומטי של ישויות (named entity recognition), למשל האבחנה בין "Nice System is an Israeli company" ל-"I have nice systems at home".

**עמימות תחבירית** - התחביר של השפות הטבעיות הוא עמום, כלומר אין יחס של אחד לאחד בין מבנים תחביריים לבין משמעות. מבנים תחביריים שונים עשויים לשאת משמעות זהה, ולמבנה תחבירי אחד עשויות להיות מספר משמעויות. בחירת המובן הנכון מצריכה מידע סמנטי, קונטקסטואלי וידע חוץ-לשוני על העולם. לדוגמה: למשפטים "נתנו לקופים את התפוזים משום שהם היו רעבים", ו"נתנו לקופים את התפוזים משום שהם היו רקובים", 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.

מעניין לציין שכדי 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. דוגמה לשפה שכזו היא לוז'באן (Lojban, לוז' מ-logji - לוגיקה. באן מ-bangu - שפה).

**קלט לא ברור -** מבטאים מקומיים או זרים, הפרעות בדיבור, טעויות הדפסה או לשון; טעויות בזיהוי תווים אופטי (OCR). כמחצית מהדוברים עברית ישראלית אינם דוברים ילידיים, ולפיכך נושאים מבטא זר כלשהו, דבר המקשה על פענוח ממוחשב של הדיבור. בערבית קיימים סגנונות רבים של כתבי יד, דבר המקשה מאוד על פיתוח זיהוי תווים אופטי לכתב יד ערבי.

**פרגמטיקה של השפה** - פירוש המשפט אינו נובע בהכרח ממשמעות המילים והמבנים התחביריים. למשל: המשפט: "האם אתה יכול להעביר את המלח?", אף שהוא בנוי כשאלת "כן או לא", אינו שאלה למעשה, אלא בקשה. התגובה המצופה למשפט כזה היא העברת המלח למבקש. התשובה "לא" עדיין נשמעת סבירה, אבל התשובה "כן" לשאלה כזאת נשמעת מוזרה. מבחינה לוגית פשוטה, קשה להסביר מדוע התשובה "כן" אינה קבילה עבור משפט שמנוסח כשאלת "כן או לא". התשובה לשאלה: "כמה תלמידים נכשלו בבחינה?" צריכה להיות "כולם" אם כל התלמידים שניגשו לבחינה נכשלו. התשובה "שלושה" (במקרה שבו רק 3 תלמידים ניגשו לבחינה) נתפסת כבלתי-קבילה, אף שמבחינה לוגית אין בה פגם. פול גרייס ניסח שורה של כללים המכונים "עקרון שיתוף הפעולה" שמסבירים את המקרים האלה, אולם אלה הם כללים פסיכולוגיים או סוציולוגיים, ואינם כללים לוגיים. לפיכך, קשה מאוד לכתוב תוכנית מחשב שתכלול אותם.

## עיבוד שפה טבעית סטטיסטי

עיבוד שפה טבעית סטטיסטי משתמש בשיטות של סיכויים והסתברויות על מנת לפתור כמה מהבעיות שצוינו למעלה, במיוחד אלה שעולים מכך שמשפטים ארוכים הם לרוב לא ברורים כשמעבדים אותם בשיטות לשוניות, והדבר מביא לאלפי או מיליוני אפשרויות ניתוח. שיטות להבהרת המצב משתמשות בדרך כלל באוצר של טקסט מוכן ובמודלי מרקוב. הטכנולוגיה לעיבוד סטטיסטי מגיעה בעיקר מתחומי למידת מכונה וכריית מידע

# NLTK

## הגדרה

זוהי ספרייה של פייתון שמאפשרת לנו להפעיל שיטות שונות כדי לעבד ולנתח שפות טבעיות. מיועדת בעיקר לשפה האנגלית, אך יש לה תמיכה מוגבלת בשפות מרכזיות אחרות כמו: צרפתית, ספרדית וגרמנית. nltk אינה כתובה בצורה יעילה וגם לא כל כך משתמשים בה בתעשייה. אנו נשתמש בה בעיקר עדי ללמוד וגם תשמש בהמשך אותנו ללמידה עמוקה. כדי להשתמש בספריה צריך לייבא אותה לפני השימוש.

Import nltk

## Tokenization

שיטה שבה מחלקים מחרוזות למילים או משפטים כדי שנוכל לעבד כל מילה או משפט בנפרד. יש לייבא מ-nltk את הפונקציה word\_tokenize כדי לפצל למילים, ואת sent\_tokenize כדי לפצל למשפטים. כל אחת מהם מקבלת מחרוזת ומחזירה רשימה מסודרת של מילים או משפטים בהתאם לסידורם במחרוזת.

>>> import nltk

>>> from nltk.tokenize import word\_tokenize, sent\_tokenize

>>> my\_text = "Where is St. Paul located? I don't seem to find it. It isn't in my map."

>>> 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."]

הסיבה שלא משתמשים בשיטה split לפי רווח עבור מילים היא כי בשיטה זו נקבל מילים שלמות עם סימנים שונים כמו '.', '?', '!', וכו' בעוד אנו רוצים מילים נפרדות. וכן אנו מעוניינים שמילים מחוברות כמו it’s תפוצל לשני מילים ולא מילה אחת. כמו כן, לא נפצל משפטים באמצעות split על נקודה כי יש דרכים נוספות לסיים משפט כמו '?' ו-'!'. וכן כי נקודה גם משמשת לקיצור מילים, כמו Mrs., ולאו דווקא לסיום משפט. הפונקציות שתיארנו ב-nltk יודעות להתמודד עם כל בעיות אלו בצורה מדויקת.

## POS - Part of Speech

זוהי שיטה המחלקת מילים לקטגוריות על בסיס השימוש התחבירי שלהם. כאשר יודעים ה-part of speech (בקיצור pos) של מילים במשפט אנו לרוב יכולים להכריע מה הכוונה האמיתי במשפט ולא לקבל שגיאת רב משמעות (ambiguous). לדוגמה, עבור המשפטים “I saw the show”, “Show me where to go”, המילה show בראשון מתייחסת לשם עצם (הופעה) ובשני לפועל (להראות). ידיעת ה-pos של המילה יעזור לנו לפרש את המשמעות שלה במשפט. חלקי הדיבור הנפוצים הם:

Noun (שם עצם) - boy, John, birthday.

Verb (פועל) - went, ate, is.

Pronoun (כינוי) - it, she, ours.

Adjective (תואר השם) - big, smart, five.

Adverb תואר הפועל) - well, quickly.

Determiner (קובעת את סוג ההפניה לשם עצם יחיד או קבוצה) - the, an, this, few, every, a.

Preposition (מילת יחס) - over, to, at.

Conjunction (מילת חיבור/קישור) - and, or, but.

Interjection (הערה פתאומית) - Hurray!.

Adposition (תיאור מיקום) - in, under, towards, before.

ב-nltk משתמשים ב-Penn TreeBank שהוא מסד נתונים המכיל אוצר גדול של משפטים שהגדירו בהם ידנית מה ה-pos של כל מילה בכל משפט. מכיל גם עצי גזירה עליהם נלמד בהמשך. ה-Penn TreeBank זהו מסד נתונים שעבדו עליו הרבה ואף השתמשו בו רבות בעבר. כיום פחות בשימוש. הוא מכיל יותר תגיות של pos ממה שתיארנו לעיל שכן כל חלק ניתן לפרק עוד יותר. באמצעות פקודת nltk.help.upenn\_tagset() ניתן לקבל את כל התגיות המוגדרות במסד נתונים זה.

CC - Coordinating conjunction

CD - Cardinal number

DT - Determiner

EX - Existential there

FW - Foreign word

IN - Preposition or subordinating conjunction

JJ - Adjective

JJR - Adjective, comparative

JJS - Adjective, superlative

LS - List item marker

MD - Modal

NN - Noun, singular or mass

NNS - Noun, plural

NNP - Proper noun, singular

NNPS - Proper noun, plural

PDT - Predeterminer

POS - Possessive ending

PRP - Personal pronoun

PRP$ - Possessive pronoun (prolog version PRP-S)

RB - Adverb

RBR - Adverb, comparative

RBS - Adverb, superlative

RP - Particle

SYM - Symbol

TO - to

UH - Interjection

VB - Verb, base form

VBD - Verb, past tense

VBG - Verb, gerund or present participle

VBN - Verb, past participle

VBP - Verb, non-3rd person singular present

VBZ - Verb, 3rd person singular present

WDT - Wh-determiner

WP - Wh-pronoun

WP$ - Possessive wh-pronoun (prolog version WP-S)

WRB - Wh-adverb

כדי להגדיר pos לכל מילה, יש להפעיל פונקציית nltk.pos\_tag() על רשימה של מילים מופרדות באמצעות tokenization. פונקציה זו מחזירה מילון שבו כל המפתחות הן המילים והערכים הן ה-pos של כל מילה.

>>> from nltk.tokenize import word\_tokenize

>>> from nltk import pos\_tag

>>> my\_text = "Whoever eats many cookies is regretting doing so"

>>> nltk.pos\_tag(word\_tokenize(my\_text))

[('Whoever', 'NNP'), ('eats', 'VBZ'), ('many', 'JJ'), ('cookies', 'NNS'), ('is',' VBZ'), ('regretting', 'VBG'), ('doing', 'VBG'), ('so', 'RB')]

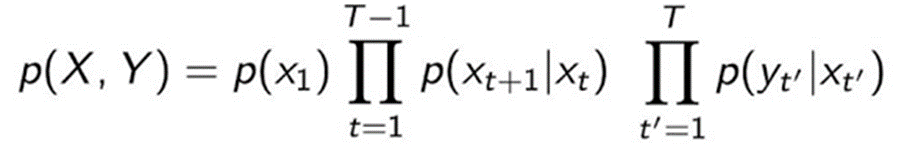
## HMM - Hidden Markov Model עם אלגוריתם Viterbi

בסעיף זה נלמד על אלגוריתם HMM שמשולב עם אלגוריתם Viterbi שמטרתו היא לתייג מילים אל ה-pos המתאים להם.

במודל HMM צריך להחזיק מסד נתונים שבו שתי טבלאות:

1. טבלת Transition שבה העמודות והשורות הן תגיות pos. כל תא (i, j) מראה מה ההסתברות לעבור מתג j אל תג i. לטבלה זו יש להוסיף עמודה של start, המציינת תג של תחילת משפט, ועוד שורה של end המייצגת תג של סוף משפט. גודל הטבלה הוא מספר התגיות בריבוע.
2. טבלת Emission שבה העמודות הן תגיות pos והשורות הן מילים באנגלית. כל תא (i, j) מראה מה ההסתברות שאכן המילה i שייכת ל-pos j.

בכל טבלה נוסיף את התגיות start ו-end ובטבלה השנייה את המילים <start> ו-<end> שתמיד יהיו בתחילת וסוף משפט. שתי טבלאות אלו צריכות להיווצר ממסד נתונים גדול כמו Penn TreeBank. בנוסף, כדי לא לקבל ערכי 0 שיאפסו את המשוואה, כדאי להוסיף הסתברות מאוד קטנה בכל תא, לפי עקרון Laplace's Smoothing.

בשיטת HMM בבואנו למצוא תיוג X למשפט Y, כך שכל מילה מקבלת תיוג, יש לחשב כל מסלול תיוג אפשרי ולבחור את התיוג X שיש לו את ההסתברות הגדולה ביותר. נוסחת החישוב ב-HMM היא:

כאשר זוהי ההסתברות לעבור מתג אל תג (לפי טבלת Transition), ו- זהו ההסתברות שמילה מתויגת לפי (באמצעות טבלת Emission). כיוון שישנם של מסלולים אפשריים, נשתמש באלגוריתם Viterbi שמוצא מסלול הרבה יותר מהר.

### אלגוריתם Viterbi

זהו אלגוריתם תכנון דינמי המשמש בין היתר כדי לחשב בסיבוכיות ליניארית את המסלול תיוג X עם ההסתברות הגדולה ביותר במודל HMM למשפט Y. נניח כי מספר המילים במשפט הוא n ומספר התגיות של pos הוא m. האלגוריתם עובר על כל המילים במשפט ועבור כל מילה מתחזק את m המסלולים הטובים ביותר עד אותה מילה הנגמרים כל אחד בתג pos אחר. לכל מסלול כזה הוא שומר את כל התגיות pos בו ואת ההסתברות שאכן זהו המסלול הנכון לתייג את כל המילים. לבסוף, כאשר מסיים לעבור על כל המילים, לוקח את המסלול עם ההסתברות הגבוהה ביותר מ-m המסלולים.

נגדיר את האלגוריתם לפי שלבי pseudo code: נגדיר לכל להיות ההסתברות שהמסלול הכי טוב לאורך המשפט נגמר בתג i. בנוסף נגדיר להיות קבוצת התגים שנגמרת ב-i. בתחילת האלגוריתם קבוצה זו ריקה ובסוף תהיה בגודל n.

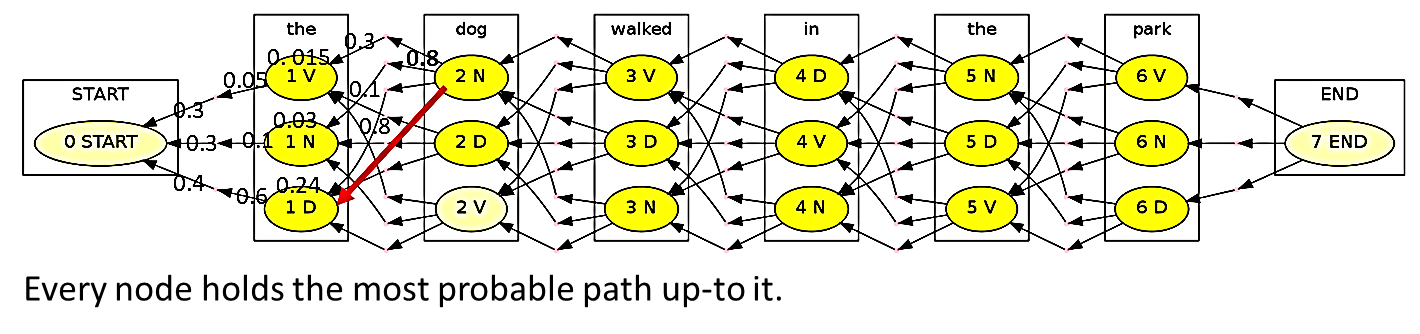
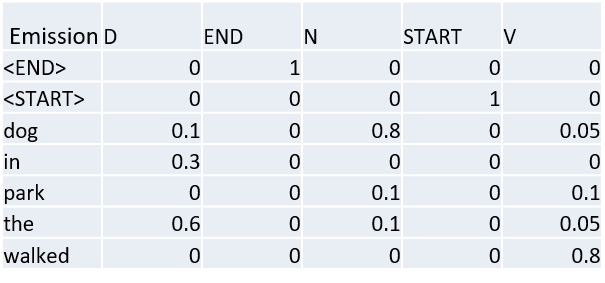
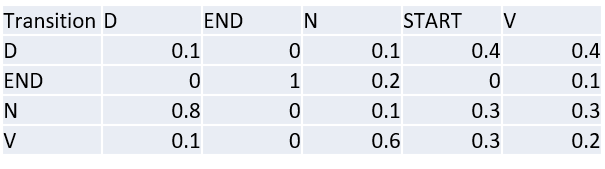
* בלולאה מ- עד n (עוברת על כל המילים במשפט Y):
* בלולאה מ- עד m (עוברת על כל התיוגים האפשריים):
* בלולאה זאת נחשב לכל תג i את המסלול הטוב ביותר שנגמר בו עד מילה j. לשם כך נצטרך לחשב עבור כל המסלולים , כאשר , את ההסתברות כאשר מוסיפים להם את i בסוף ולהגדיר את להיות המסלול הטוב ביותר שמצאנו. נצטרך לחשב:

כאשר זה ההסתברות לעבור מהמסלול שנגמר ב-k לתג i. ו-זה ההסתברות שמילה שייכת לתג i.

* בחר את המסלול עם ההסתברות הגדולה ביותר והגדר לכל מילה את הרכיב ה-j מ-.

כמו שניתן לראות משלבי האלגוריתם, הסיבוכיות היא , כלומר אורך המשפט כפול מספר תגיות pos בריבוע.

דוגמה: עבור המשפט “the dog walked in the park”, טבלאות ה-Transition וה-Emission עם שלושה תגים של pos (Noun, Verb, Determiner), וכן התהליך שיבצע אלגוריתם Viterbi ייראו כך:



https://pydecode.readthedocs.io/en/latest/notebooks/hmm.html

## Stemming

זהו תהליך לצמצום מילים נטויות או נגזרות אל הצורה הבסיסית והשורשית שלהן. תמיד נשתמש בשיטה זו לאחר שהפרדנו את המחרוזת למילים באמצעות tokenization. המניע הוא בדרך כלל כדי להקל על החיפוש בתוך המחרוזת.

יש מספר פונקציות ב-nltk שעושות זאת לפי אלגוריתמים שונים: PorterStemmer, LancasterStemmer, SnowballStemmer. כולן משתמשות בהיוריסטיקות שונות כדי לחתוך את התוספות למילים, לכן לא בהכרח מתקבלת מילה תקינה. לפני שמשתמשים בכל אחד מהם יש לייבא אותם מ-nltk.stem. לאחר מכן יש ליצור אובייקט stemming מתאים ועליו להפעיל שיטת stem המקבלת מילה ומחזירה את המילה השורשית שלה.

>>> 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’]

## Lemmatization

זוהי שיטה שמבצעת את אותה הפעולה של stemming - המרת מילים נטויות לצורה הבסיסית והשורשית שלהן. אלא שבניגוד ל-stemming, שרק חותכת תווים באמצעות כל מיני היוריסטיקות, lemmatization מנהלת מילון מילים וגם מתחשבת ב-pos של כל מילה לפני שממירה אותה. כתוצאה מכך lemmatization מחזירה תוצאות הרבה יותר טובות אך יותר איטית. כיוון שמנהלת מילון היא מכירה את המילים ויודעת להמיר גם מילים שונות לשורש המשותף להן. לדוגמה, going ו-went תמיר ל-go, is ל-be, cookies ל-cooky, וכן הלאה.

כדי להשתמש ב-lemmatization יש לייבא את WordNetLemmatizer מ-nltk.stem.wordnet, ואז ליצור אובייקט WordNetLemmatizer. על אובייקט זה יש להפעיל את שיטת lemmatize המקבלת מילה וה-pos שלה ומחזירה את המילה השורשית שלה. חשוב לשים לב שה-Lemmatizer מכיר פחות תגיות pos ממה שמוגדר ב-Penn TreeBank, לכן צריך על כל תג pos להפעיל פונקציית עזר שתקבל תג pos ותחזיר את התג בצורה יותר בסיסית שלו שגם ה-Lemmatizer מכיר.

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

**lzr = 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(**lzr.lemmatize(word,penn2wn(pos))**)

lemed

['Whoever', 'eat', 'many', 'cooky', 'be', 'regret', 'do', 'so']

## Chunking

זוהי שיטה לחתוך משפט או קבוצת משפטים לקבוצות לפי תבניות משפטים שהגדרנו. המטרה היא לחפש תבניות בתוך הטקסט.

נגדיר תבניות באמצעות ביטויים רגולריים על ה-pos. קודם יש לתת שם לתבנית, לאחר מכן נקודותיים, ואז בתוך סוגריים מסולסלות לרשום את הביטוי הרגולרי באמצעות האופרטורים הבאים:

? – מורה ש-pos זה הוא אופציונלי לתבנית ולא בהכרח חייב להופיע.

\* - 0 או יותר, כלומר יכול להופיע כמה פעמים ברצף או בכלל לא.

?. – כל אות יכולה להחליף סימון זה. ככה ניתן לחפש סוגים דומים של pos ולא אחד ספציפי.

+ - כמו \* רק חייב להופיע לפחות פעם אחת.

אחרי שהגדרנו תבנית ניתן להשתמש בה בתבניות אחרות. כך אנו יוצרים עץ עמוק יותר כמו שנראה בהמשך. דוגמאות:

* NP: {<DT>?<JJ>\*<NN>} - כל המשפטים שיכולים להתחיל ב-determiner, 0 או יותר שמות תואר (adjectives) ואז שם עצם. דוגמה למשפט מתאים: "The nice big boy".
* Noun List: {(<DT>?<NN.?><,>?)+<CC><DT>?<NN.?>} - לפחות ביטוי אחד המורכב מ-determiner, כל סוג של שם עצם ופסיק (כולם אופציונליים), לאחר מכן מילת חיבור, ושוב determiner ושם עצם אופציונליים. דוגמה למשפט מתאים: "Sara, John, Tom, the girl and the bat".

כדי להשתמש ב-Chunking ב-nltk יש ליצור מחרוזת המייצגת את התבנית שאנו מחפשים וליצור אובייקט מסוג nltk.RegexpParser המקבל מחרוזת של תבנית כמו שיצרנו. על אובייקט זה נפעיל שיטת parse המקבלת מילון של כל המילים בטקסט, כך שהמילים הן המפתחות במילון והערך הוא ה-pos שלהן. שיטה זו תחזיר אובייקט המכיל את כל המשפטים מתבנית זו. אובייקט זה אפשר להדפיס למסך ואף לצייר באמצעות שיטת draw().

>>> from nltk.tokenize import word\_tokenize

>>> from nltk import pos\_tag

>>> my\_text = "Dogs or small cats saw Sara, John, Tom, the *pretty* girl and the *big* bat"

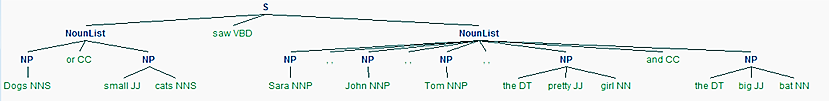
>>> tagged = nltk.pos\_tag(nltk.tokenize.word\_tokenize(my\_text))

>>> grammar = "NP: {<DT>?<JJ>\*<NN.?>}

NounList: {(<NP><,>?)+<CC><NP>}"

>>> cp = nltk.RegexpParser(grammar)

>>> result = cp.parse(tagged)

>>> result.draw()

## Named Entity Recognition (NER)

זהו גם סוג של chinking אלא שמחלק רק שמות עצם למקרים פרטיים. הוא יודע לזהות סוגים של שמות עצם כמו: שמות אנשים, מדינות, אירועים, זמנים, ארגונים וכו'. כמו כן הוא יכול לחבר מילים סמוכות תחת קבוצה אחת אם הם מייצגות ביחד שם עצם פרטי כלשהו. לדוגמה United States הוא יסווג תחת קבוצה אחת למדינה. זהו כלי מאוד שימושי כאשר מחפשים שמות עצם מסוימים.

geo = Geographical Entity

org = Organization

per = Person

gpe = Geopolitical Entity

tim = Time indicator

art = Artifact

eve = Event

nat = Natural Phenomenon

כדי להשתמש יש להכניס לשיטה nltk.ne\_chunk מילון של כל המילים בטקסט, כך שהמילים הן המפתחות במילון והערך הוא ה-pos שלהן. שיטה זו תחזיר אובייקט המכיל את כל הטקסט עם סימון מתאים לשמות עצם המיוחדים. אובייקט זה אפשר להדפיס למסך ואף לצייר באמצעות שיטת draw().

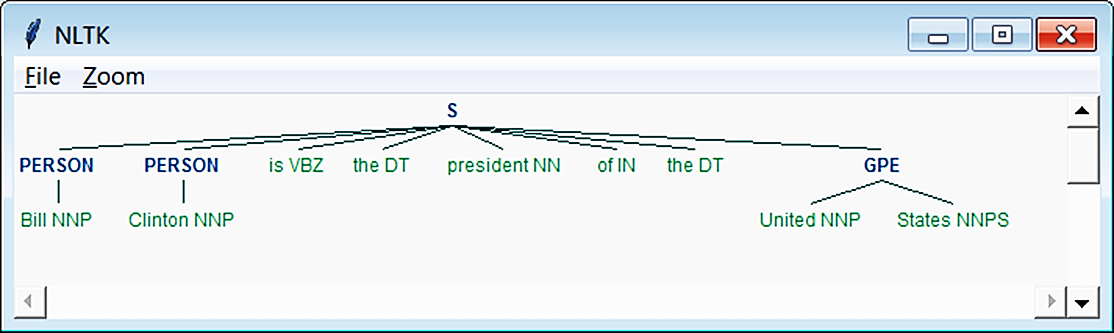
>>> from nltk.tokenize import word\_tokenize

>>> from nltk import pos\_tag

>>> my\_text = "Bill Clinton is the president of the United States"

>>> tagged = nltk.pos\_tag(nltk.tokenize.word\_tokenize(my\_text))

>>> result = nltk.ne\_chunk(tagged)

>>> result.draw()

## N-Grams

בעיבוד שפות טבעיות אנו מתעניינים לרוב בקבוצות של מילים סמוכות ולא רק במילים בודדות. באמצעות ניתוח של קבוצות מילים סמוכות נוכל:

* לשפר את מנגנון סיווג Part Of Speech שתיארנו.
* ללמוד תכונות על הטקסט וכך לעבדו טוב יותר.
* ליצור טקסט חדש. יכול לשמש אותנו בתיקון טקסט (כמו ווטסאפ), השלמת טקסט (כמו גוגל) ואף יצירת סיפור חדש.

חלוקה זו לקבוצת מילים סמוכות בגודל N נקראת באופן כללי N-Grams. כאשר N=2 החלוקה נקראת Bi-Grams וכאשר N=3 החלוקה נקראת Tri-Grams.

הפונקציה ב-nltk המייצרת N-Grams היא nltk.ngrams המקבלת רשימה של כל המילים מופרדות (באמצעות tokenization) ומספר n שיגדיר כמה מילים בכל קבוצה. הפונקציה תחזיר רשימה של tuples בגודל n המכילה את כל ה-n-Grams.

>>> text = "It is a simple text this, this is a simple text, is it simple?"

>>> list(nltk.ngrams(nltk.word\_tokenize(text),3))

[('It', 'is', 'a'), ('is', 'a', 'simple'), ('a', 'simple', 'text'), ('simple', 'text', 'this'), ('text', 'this', ','), ('this', ',', 'this'), (',', 'this', 'is'), ('this', 'is', 'a'), ('is', 'a', 'simple'), ('a', 'simple', 'text'), ('simple', 'text', ','), ('text', ',', 'is'), (',', 'is', 'it'), ('is', 'it', 'simple'), ('it', 'simple', '?')]

## Text Generation

בהינתן טקסט כלשהו נרצה ליצור טקסט חדש שדומה לטקסט הנתון. הסיבות לכך הן:

* השלמת טקסט - בהינתן התחלה של טקסט נרצה אלגוריתם שיסיים אותו. דוגמה מוכרת היא מנוע החיפוש של גוגל.
* ייצור טקסט - יכול להיות שימושי גם כאשר לטקסט החדש אין באמת משמעות. דוגמאות שימושיות הן: במשחקים כאשר פותחים ספר אקראי שיהיה כתוב שם משהו בלי שנצטרך באמת לכתוב אותו, בעיצוב פליירים התצוגה תהיה עם טקסט שהמחשב יצר ולא משהו קבוע.

נתאר אלגוריתם לייצור טקסט:

* פצל את כל המידע ל-Tri-Grams.
* נרשום שני מילים כלשהם שיהוו התחלה של הטקסט החדש.
* בלולאה עד שנקבל טקסט בגודל הרצוי (כל איטרציה מוסיפה מילה):
* לחזות את המילה הבאה ניקח את שני המילים האחרונות ונחפש את כל השלישיות המתחילות בשני מילים אלו.
* מכל שלישייה ניקח רק את המילה האחרונה ונכניס לרשימה. יכולים להיות שלישיות עם מילה אחרונה זהה כך שמילה זו תופיע ברשימה פעמים רבות.
* נגריל מתוך הרשימה מילה באופן אחיד שהיא תהיה המילה הבאה בטקסט.
* לעשות detokenization כדי לאחד את כל המילים שנוצרו אל מחרוזת אחת.

הסיבה שלא הולכים על המילה שמופיעה הכי הרבה פעמים היא כדי למנוע מקרים שבהם חוזרים על אותו טקסט שוב ושוב. אפשר לייעל אלגוריתם זה אם מתייחסים גם ל-pos. כיום, באמצעות טכניקות של למידה עמוקה הצליחו ליצור מחולל טקסט ברמה מאוד גבוהה.

import nltk

import urllib # the source of the text

from random import randint

paragraph\_len = 100

all\_text = urllib.request.urlopen("https://s3.amazonaws.com/text-datasets/nietzsche.txt").read().decode("utf-8")

my\_grams = list(nltk.ngrams(nltk.word\_tokenize(all\_text), 3))

sentence = ["It", "is"]

for i in range(paragraph\_len):

options = []

for trig in my\_grams:

if trig[0].lower() == sentence[-2].lower() and trig[1].lower() == sentence[-1].lower():

options.append(trig[2])

if len(options) > 0:

sentence.append(random.choice(option))

print(" ".join(sentence))

## Context Free Grammar (CFG)

זוהי שיטה המאפשרת לנו להגדיר דקדוק חסר הקשר (Context Free-Grammar, כמו שלמדנו בקורס אוטומטים 2). דקדוק חסר הקשר במובן של עיבוד שפות טבעיות מורכב מארבעה רכיבים:

V – קבוצת כל ה-pos שמהם יורכב המשפט.

T – קבוצת כל המילים שיכולים להופיע במשפט.

S – משתנה התחלתי ממנו נתחיל לגזור את המשפט, .

P – קבוצה של כללי גזירה מהצורה , כאשר תג pos כלשהו, ו-b הוא רצף של סמלים מ-, כלומר מכיל מילים ותגי pos.

כדי להגדיר דקדוק חסר הקשר מספיק להגדיר את הקבוצה P שכן כבר מכילה את כל שאר הקבוצות.

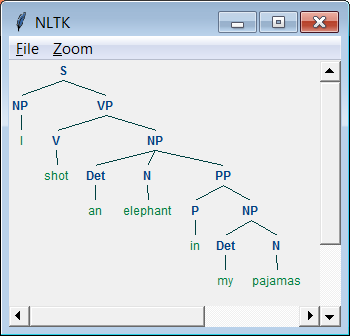
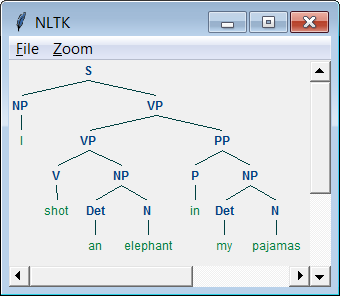
הסיבות שנרצה להגדיר דקדוק חסר הקשר הם:

1. לבדוק האם משפט נתון שייך לשפה של הדקדוק. זוהי עוד דרך לחפש תבניות בתוך הטקסט (Chunking).
2. באמצעות סדרת גזירה ממנה נוצר המשפט נוכל לסווג לכל מילה במשפט תג pos מתאים.
3. מסדרת גזירה היוצרת משפט נוכל לבנות עץ גזירה. באמצעות חלוקת הרמות בעץ הגזירה נוכל להכריע מה הכוונה מאחורי משפטים רבי משמעות (ambiguous). זה השימוש העיקרי ב-CFG.

עבור משפטים להם יש כמה סדרות גזירה, נלמד בהמשך כיצד לתת לכל סדרת גזירה כזו הסתברות, ומתוך כך להחליט מה המשמעות הכי סבירה של המשפט וכן מהי החלוקה הכי סבירה של המילים ל-pos.

יצירת דקדוק חסר הקשר נעשית באמצעות פונקציית nltk.CFG.fromstring, המקבלת מחרוזת של כל כללי הגזירה בדקדוק. פונקציה זו מחזירה אובייקט המייצג את הדקדוק. לאחר מכן יש להכניס את האובייקט שקיבלנו אל הפונקציה nltk.ChartParser שמחזירה אובייקט CFG parser. אל ה-parser יש להכניס רשימה של מילים. ה-parser יחזיר רשימה של סדרות גזירה אפשריות, אם קיימות, למשפט שקיבל. סדרת גזירה ניתן להדפיס באמצעות print ואפשר גם לצייר את עץ הגזירה באמצעות draw.

>>> grammar = nltk.CFG.fromstring("""

... S -> NP VP

... PP -> P NP

... NP -> Det N | Det N PP | 'I'

... VP -> V NP | VP PP

... Det -> 'an' | 'my'

... N -> 'elephant' | 'pajamas'

... V -> 'shot'

... P -> 'in'

... """)

>>> parser = nltk.ChartParser(grammar)

>>> for tree in parser.parse(“I shot an elephant in my pajamas”.split() ):

... tree.draw()

ניתן לראות שלפי עץ הגזירה הימני המשמעות היא "צילמתי פיל שהיה לבוש בפיג'מה שלי" ואילו לפי העץ השמאלי המשמעות היא "צילמתי פיל כשהייתי לבוש בפיג'מה".

### אלגוריתם CYK

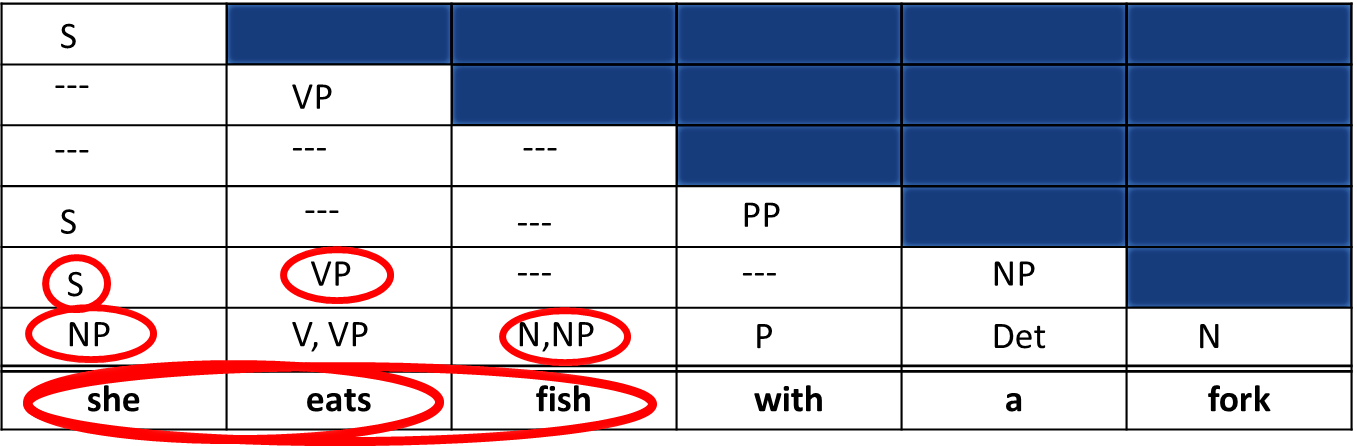
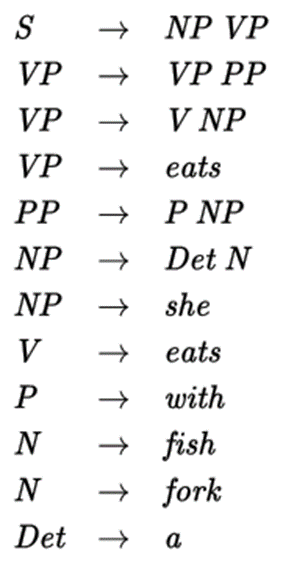
זהו האלגוריתם תכנון דינמי באמצעותו מוצאים את סדרת הגזירה המתאימה למשפט. האלגוריתם בונה את עץ הגזירה מהדרגה התחתונה, שבה המילים, לשורש S.

לפני שמריצים את האלגוריתם יש להמיר את הדקדוק לצורה הנורמלית של חומסקי (Chomsky Normal Form - CNF), שבו כל כללי הגזירה הם מהצורה או , כלומר מ-pos אל שני תגיות pos או מ-pos אל מילה. המרה זו אינה משנה את השפה של הדקדוק. נוכל לעשות אותה באמצעות הכללים הבאים:

* שנה את כל ההופעות של S אל והוסף את כלל הגזירה .
* בכל כלל גזירה שבו יש מילה w בצד ימין שאינה בודדת נחליף אותה בסימון חדש של תגית W ונוסיף כלל חדש. .
* בכל כלל גזירה שבו יש יותר משני תגיות pos בצד שמאל, נפצל אותם על ידי הוספת תגיות בדרך הבאה: .

באלגוריתם CYK עוברים בלולאה כגודל מספר המילים במשפט. בכל איטרציה i עוברים על כל i מילים סמוכות במשפט ומחפשים את ה-pos ממנו ניתן לגזור את כל i המילים, כאשר מתחשבים ב-pos המוביל אל חלק מ-i המילים שמצאנו באיטרציות קודמות. את התוצאות רושמים בתוך טבלה (כמו שמתוארת בדוגמה למטה), ועבור כל pos שמיקמנו בטבלה צריך לזכור את שני ה-pos בשורות התחתונות שהוא מצביע אליהם ודרכם הוא גוזר את כל i המילים. באמצעות מצביעים אלו מגדירים את עץ הגזירה (parse tree).

לדוגמה, עבור הדקדוק למטה נרצה להגדיר האם המשפט "she eats fish with a fork", שייך לשפת הדקדוק. נמלא את הטבלה המצורפת מהשורה התחתונה לעליונה, כאשר באיטרציה הראשונה בוחנים כל מילה בנפרד. באיטרציה השנייה עוברים על כל שתי מילים סמוכות, וכן הלאה. המטרה בשכבות הגבוהות היא למצוא שתי pos שדרכיהם אפשר לגזור את כל המילים במשפט. ניתן דוגמה לחישוב תא אחד בטבלה. בשורה החמישית מלמטה במילה eats, נרצה למצוא תיוג דרכו אפשר לגזור את חמשת המילים מ-eats עד fork. ניתן לראות שאכן עבור התג VP, אפשר לגזור אותו ל-PP ול-VP כך שמ-PP ניתן לגזור את שלושת המילים “with a fork” ומ-VP ניתן לגזור את שני המילים הראשונות   
eats fish".



הסיבוכיות של אלגוריתם CYK היא כאשר n הוא מספר המילים במשפט ו-|G| הוא מספר כללי הגזירה ב-G. הסיבה לכך היא שיש למלא טבלה בגודל כך שבכל תא יש לסרוק n אפשרויות כדי לכסות את כל המילים, ועבור כל אפשרות יש לחפש בכל כללי הגזירה בדקדוק כדי לבדוק האם אכן קיים כלל כזה.

## Probabilistic Context Free Grammar (PCFG)

כמו CFG רגיל שלמדנו אלא שלכל כלל גזירה יהיה הסתברות לשימוש בו. כעת עבור כל משפט שיש לו מספר סדרות גזירה אפשריות, נוכל לתת לכל סדרת גזירה כזו הסתברות, ומתוך כך להחליט מה המשמעות הכי סבירה של המשפט וכן מהי החלוקה הכי סבירה של המילים ל-pos.

שני ההבדלים היחידיים בין CFG ל-PCFG הוא שלאחר כל כלל גזירה בפירוט הדקדוק נרשום בסוגריים מרובעות את ההסתברות של הכלל. חשוב לשים לב שלכל תג pos סך הסתברות של כל כללי הגזירה שיוצאים ממנו חייב להיות 1. הבדל נוסף שה-parser שנשתמש בו הוא nltk.ViterbiParser.

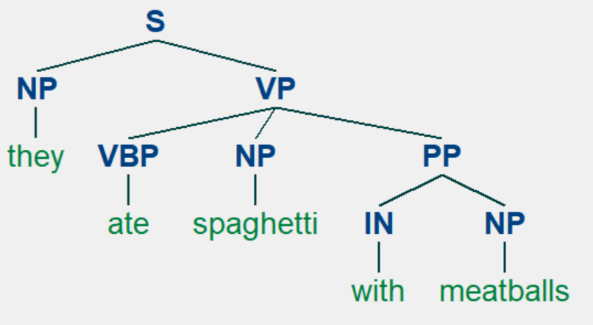
>>> grammar = nltk.PCFG.fromstring("""

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] """)

>>> viterbi\_parser = nltk.ViterbiParser(grammar)

>>> for tree in viterbi\_parser.parse( “they ate spaghetti with meatballs”.split()):

... tree.draw()

הקוד לעיל יחזיר את התמונה הזו בהסתברות הגבוהה ביותר של 0.006.

דקדוק איכותי הוא דקדוק שההסתברויות על כללי הגזירה שלו נקבעו לפי מסד נתונים מאוד גדול וגם יכול להגיע להמון משפטים. ב-NLTK אין דקדוק מובנה כזה, אמנם כן יש מספר דקדוקים קטנים אותם ניתן לטעון מ-nltk\_data.

## Sentiment Analysis

בהינתן משפט נרצה לדעת האם מייצג רגש חיובי, שלילי או ניטרלי. דוגמה שימושית היא לדרג מוצר בהתאם לביקורות שנכתבו עליו. נוכל גם לפרק כל ביקורת לתכונות ספציפיות של המוצר אליהן הביקורת מתייחסת וכך לדרג גם תכונות מסוימות ולא רק את המוצר כולו.

האלגוריתמים הנפוצים לביצוע Sentiment Analysis הוא שימוש ב-N-Grams (בדרך כלל Tri-Grams) כדי לבדוק כמה מילים חיוביות למול מילים שליליות וניתוח מעמיק בלמידה עמוקה.

כדי לעשות זאת ב-NLTK יש לייבא אובייקט SentimentIntensityAnalyzer ולהפעיל עליו שיטת polarity\_scores המקבלת משפט. שיטה זו מחזירה מילון המכיל תוצאות בארבע קטגוריות: שלילי, חיובי, ניטרלי ומורכב. ניתן להחליט האם המשפט מבטא רגש חיובי שלילי על ידי השוואה איזו קטגוריה גדולה יותר חיובי או שלילי. דרך נוספת היא בקטגוריה “compound”, המכילה ערך בין 1 המייצג רגש חיובי אל -1 המייצג רגש שלילי.

>>>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}

# CoreNLP

## הגדרה

זוהי ספרייה של אוניברסיטת Stanford לעיבוד שפות טבעיות. ספרייה זו היא חזקה יותר מ-NLTK ויותר משתמשים בה בתעשייה. ניתן לקרוא עוד על הספרייה ועל היכולות שלה [כאן](https://stanfordnlp.github.io/CoreNLP/). ספרייה זו נכתבה ב-Java אך ניתן להשתמש בה גם דרך פייתון.

בניגוד ל-NLTK ספריית CoreNLP מאפשרת לטעון דקדוק איכותי עם הסתברויות שהסיקו אותו ממסד נתונים גדול מאוד. נתמקד כאן במספר תכונות מרכזיות של ספרייה זו.

## Dependency Parsing

בניגוד ל- Constituent Parsingשנעשה באמצעות דקדוק חסר הקשר ומתאר איך כל מילה נגזרת ממילה אחרת, Dependency Parsing מראה את התלות והיחסים בין המילים השונות במשפט. לדוגמה, הוא מראה לנו: מיהו הנושא (subject) במשפט שמבצע את הפועל, לאיזה שם עצם determiner מפנה, מיהם ה-modifiers (תואר או פועל עליהם מוסיפים מידע) במשפט, וכו'. בדרך כלל הפועל במשפט יהיה השורש. Dependency Parsing הרבה יותר מהיר מ-Constituent Parsing.

דוגמה לשימוש בספריית CoreNLP דרך NLTK.

from nltk.parse.stanford import StanfordDependencyParser

path\_to\_jar = 'path\_to/stanford-parser-full-2014-08-27/stanford-parser.jar'

path\_to\_models\_jar = 'path\_to/stanford-parser-full-2014-08-27/stanford-parser- 3.4.1-models.jar'

dependency\_parser = StanfordDependencyParser(path\_to\_jar=path\_to\_jar, path\_to\_models\_jar=path\_to\_models\_jar)

result = dependency\_parser.raw\_parse('I shot an elephant in my sleep')

dep = result.next()

list(dep.triples())

## Co-reference Resolution

טכניקה שבה גם כן מזהים תלויות בין מילים כמו ב-Dependency Parser, אלא שמתעניינת רק בלזהות לאיזה שם עצם מילה כלשהי מתייחסת. נביא מספר דוגמאות:

"”The bus was full. It drove very fast – כאן המילה it מתייחסת לאוטובוס.

"Emma was happy. She didn't say a word" – המילה she מתייחס ל-Emma.

"The cow ate a lot of grass. It was hungry" – המילה it מתייחסת לפרה.

כל הבחנות אלו הן מאוד קלות לבני אדם אך למחשב הן יכולות להיות די מסובכות, במיוחד אם המילה מופיעה הרבה אחרי השם עצם אליה היא מתייחסת. האלגוריתמים באמצעות מבצעים Co-reference Resolution משתמשים בלמידת מכונה והם כל הזמן משתפרים כיוון שמספרים להם יותר מידע. ניתן לקרוא עוד [כאן](http://stanfordnlp.github.io/CoreNLP/coref.html).