NLTK: перспективный синтаксический парсер или …

Автор: Антон Бузанов

Оглавление

NLTK: перспективный синтаксический парсер или … 1

1. Natural language Toolkit 1
2. Построение деревьев с помощью Python-а 1
3. Рациональное использование NLTK 2
4. Что делать дальше? 4
5. Вывод 4

1. Natural language Toolkit

NLTK – (очень) популярная платформа для процессинга естественного языка.

Включает в себя:

- интегрированные корпуса (WordNet и др.)
- pos-таггер
- sent-токенизатор
- word-токенизатор
- списки стоп-слов
- etc

Плюсы:

- для английского NLP-функционал в целом работает неплохо
- достаточно много документации

Минусы:

- точность

2. Построение деревьев с помощью Python-а

В первую очередь, чтобы размечать предложения и строить их правильные синтаксические деревья, нужно выделять предложения. Но этот вопрос в докладе затронут не будет. Предположим, что программа получает на вход законченные грамматичные предложения.

И что же дальше? Чтобы это узнать, проведём испытание NLTK на примере одного предложения:

(**`input_str`**) *I want to kill my brother*

В этом предложении имеем именную группу *my brother* и глагольную(ые) группу(ы) *want to kill my brother*.

Предложение выбрано. Следующий этап: POS-tagging. Olga Davydova, Data Monsters предлагают использовать TextBlob, от которого мы отказались ещё на этапе выбора таггера. Тем не менее, моя идея была протестировать NLTK ровно так, как это предлагают исследователи.

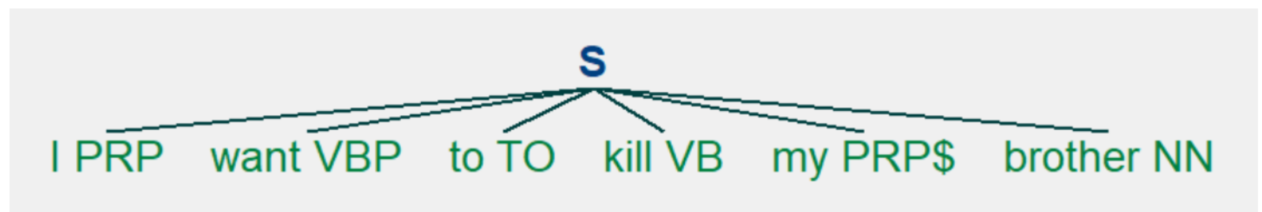
```
from textblob import TextBlob
result = TextBlob(input_str)
```

Что тогда получается на выходе? Переменная result – особая переменная особого типа *TextBlob*, но для наших целей нам нужно получить лишь список слов с частеречными тэгами:

```
result.tags
>>> [('I', 'PRP'), ('want', 'VBP'), ('to', 'TO'), ('kill', 'VB'), ('my', 'PRP$'), ('brother', 'NN')]
```

Анализируем, что получилось: вроде всё неплохо, но тэги простоваты (либо со знаком $).

Кажется, что осталось немного: положить эту штуку в NLTK и получить на выход дерево. Пробуем:



Недерево

Что-то пошло не так, как надо: никакой глубинной структуры, никакого автоматического анализа.

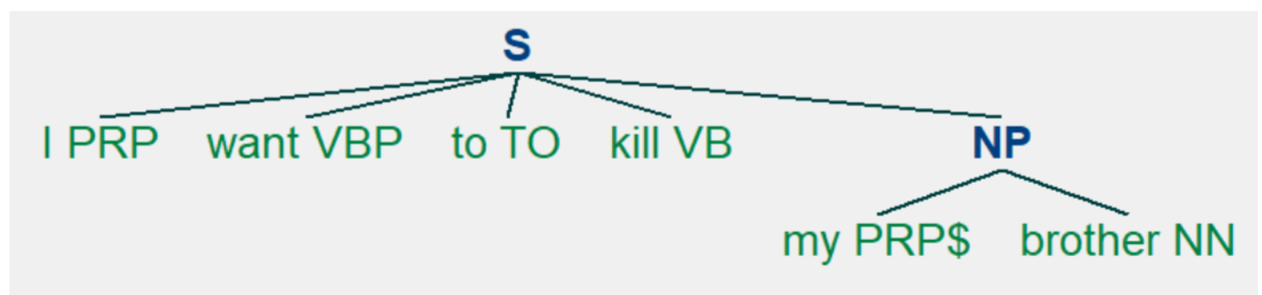
3. Рациональное использование NLTK

Оказывается, чтобы NLTK правильно работал, нужно **вручную** написать регулярные выражения, для схлопывания тех или иных последовательностей в группы.

Выглядит это всё так:

```
reg_exp = r'''
    NP: {<DT>? <PRP\$>? <JJ>* <NN.?>}
    '''
```

Здесь сказано, что именная группа – это последовательность типа детерминат (opt.), притяжательное местоимение (opt.), любое количество прилагательных, а в конце существительное в какой-то форме. Получаем:

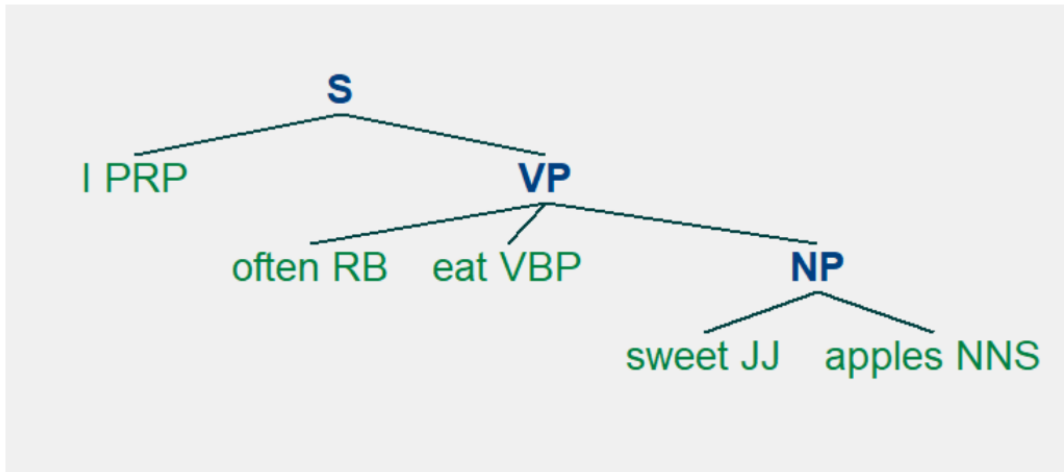


Дерево с NP

Стоит признать, что это уже выглядит лучше. Но дописав регулярку для глагольной группы:

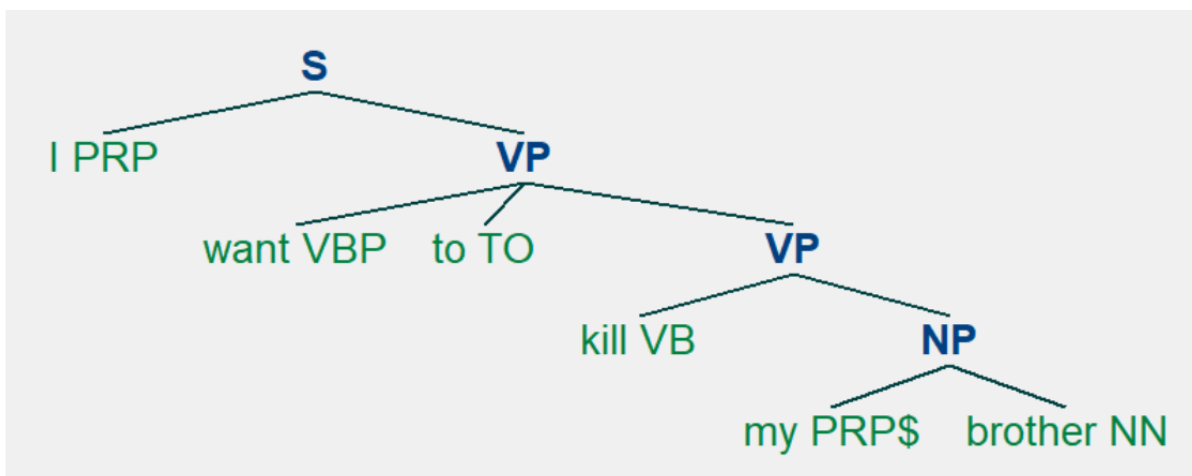
```
reg_exp = r'''
    VP: {<RB>? <VB.?> <NP>?}
    '''
```

Откуда RB? Очень просто, если у нас будет предложение *I often eat sweet apples*, то наречие хочется считать зависимым глагола, и включать их в одну составляющую.



Правильное дерево

Такой результат меня устраивает. Но возникают некоторые теоретические вопросы: стоит ли постулировать отдельную инфинитивную группу, как в первом примере, или делать вложенную VP, как здесь:



Спорное дерево

Стремимся ли мы к бинарности? Какие группы логично выделять? И тд и тп.

4. Что делать дальше?

На этом этапе становится ясно, что на вход можно подавать и предложения размеченные другим таггером, ведь правила мы всё равно пишем сами. Я пока не пробовал перенести описания в парадигму TreeTagger-а, так как не ясно, станем ли мы вообще использовать NLTK для таких целей.

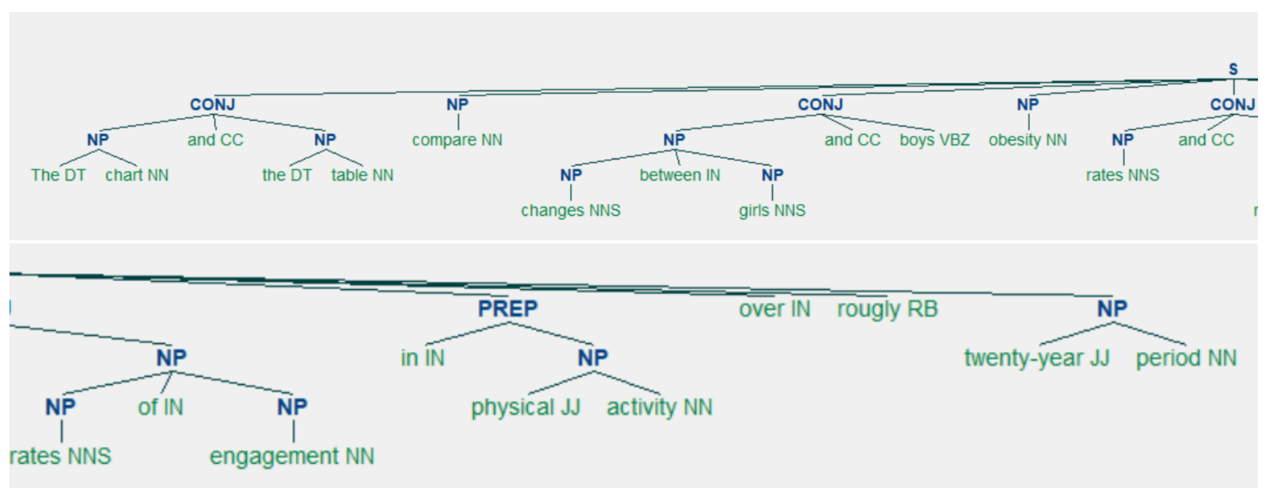
Можно переписать существующее регулярное выражение для именной группы для его работы в NLTK. Также, если рационально задать правила, то можно описать большинство контекстов, а не затронутые не повлияют на статистику существенным образом.

5. Вывод

- Инструмент неплох.
- Но сложен.
- Нужно очень много регулярных выражений, этих, например, не хватает:

```
reg_exp = r'''
    NP: {<DT>? <PRP\$>? <JJ>* <NN.?>}
    NP: {<NN> <NNS>}
    NP: {<NP> <IN> <NP>}
    NP: {<NN> <PREP>}
    NP: {<CONJ> <NP>}
    VP: {<RB>? <VB> <NP>?}
    VP: {<RB>? <VBP> <TO>? <VP>?}
    PREP: {<IN> <CONJ>}
    PREP: {<IN> <NP>}
    CONJ: {<.*> <CC> <.*>}
    '''
```

Итоговые регулярные выражения



Дерево реального предложения из эссе

- зато можно подключать свою pos-разметку
- можно удобно задавать несколько паттернов для одного типа
- вообще удобная архитектура
- возможность рекурсии