ЦИФРОВАЯ КУЛЬТУРА

УНИВЕРСИТЕТ ИТМО

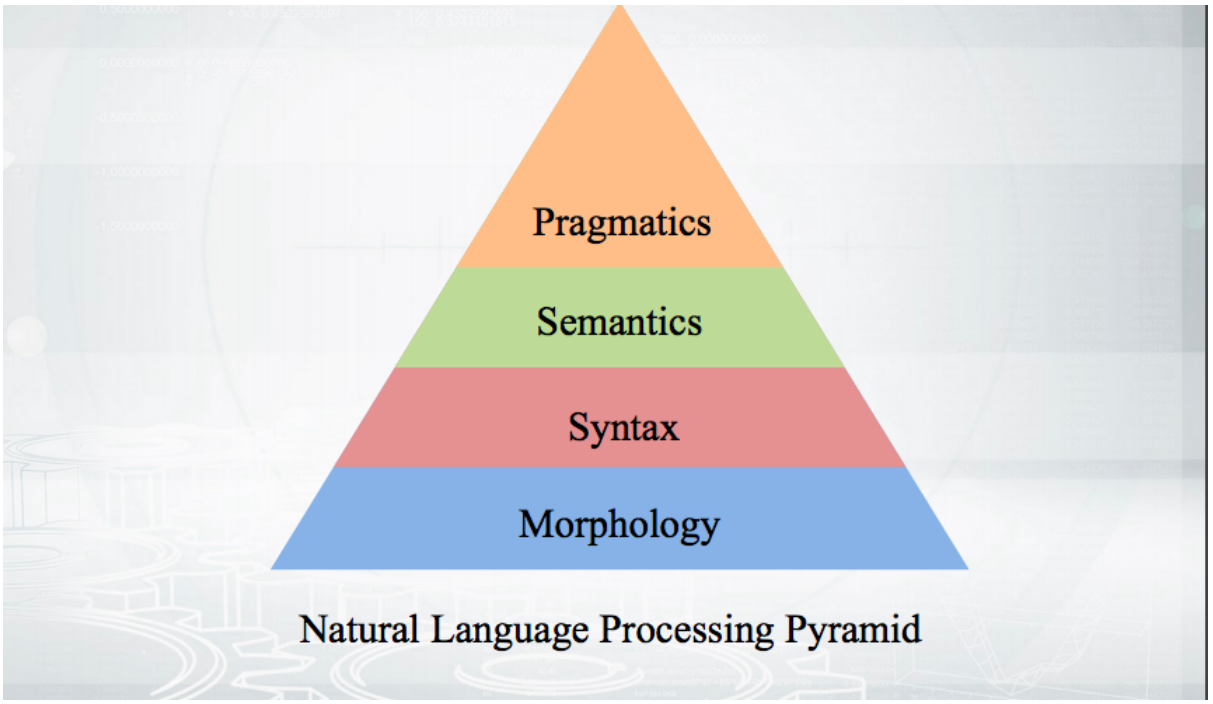
Синтаксис

Высшая школа цифровой культуры
Университет ИТМО
dc@itmo.ru

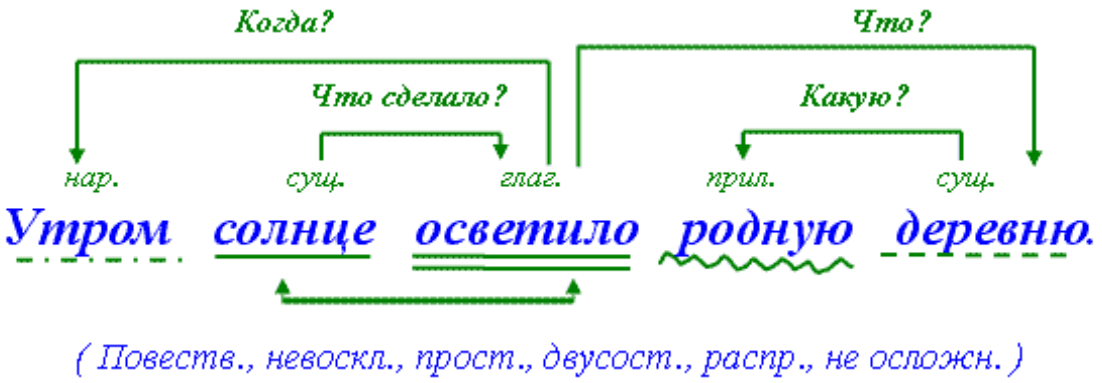
Оглавление

Что такое синтаксический анализ?

Давайте поговорим о том, что такое **автоматический синтаксический парсинг**. Но сначала обсудим, что такое синтаксис и зачем он нужен. Автоматический разбор текста состоит из нескольких этапов.



Вначале делают токенизацию, разбивая предложения на подстроки, или токены, к которым относятся как слова, так и знаки пунктуации. С токенизацией мы уже сталкивались на первых неделях курса. Затем следует этап морфологического анализа, в рамках которого анализируются отдельные слова: каждому слову приписываются определенные грамматические категории: падеж, число и т.д. После этого начинает свою работу синтаксический анализ, а именно анализ на уровне взаимодействия между словами. Сфера рассмотрения синтаксиса - это словосочетания, от минимального сочетания двух слов вплоть до целого предложения.



В школе мы все учились делать синтаксический анализ, выделяя в предложении подлежащее, сказуемое, определения, дополнения и т.д. Компьютеру нужно сделать то же самое. Вот пример: «Утром солнце осветило родную деревню». Солнце – подлежащее, осветило – сказуемое, и т.д. При этом слова зависят друг от друга. Например, сказуемое зависит от подлежащего, другие слова зависят от сказуемого, и т.д. Например, определение «родную» зависит от дополнения «деревню».

Есть алгоритмы, которым с некоторой точностью под силу делать всё это автоматически, а синтаксический анализ помогает в ряде других важных задач. Например, если рассматривать предложение просто как набор слов, как делает модель Bag of Words -- мешок слов, никак не учитывая зависимости между словами и их порядок, то предложения «Банкомат съел карту» и «Карта съела банкомат» машина воспримет как абсолютно идентичные.

Меж тем, как только мы начинаем учитывать порядок слов, а значит, и синтаксические связи между словами, эти примеры становятся разными. Есть множество задач обработки естественного языка, в которых синтаксис действительно важен. Например, для системы порождения речи (Natural Language Generation) фразы должны быть сгенерированы правильно не только с точки зрения смысла, но и с точки зрения расстановки слов. Кроме того, сейчас достаточно большую популярность набирают вопросно-ответные системы, устроенные так, что на заданный текстом вопрос должен быть найден ответ. Часто к запросу применяется метод извлечения троек: субъекта, предиката и объекта, или, иными словами, подлежащего, сказуемого и прямого дополнения. После извлечения этой тройки из текстового запроса пользователя, она отправляется в некоторую базу знаний в виде структурированного, то есть разобранного и преобразованного в удобный для понимания машиной вид,

запроса. Если же запрос не удалось структурировать, то и найти правильный ответ, скорее всего, не удастся.

Синтаксический анализ нужен и в машинном переводе. Если мы правильно поняли смысл фразы на исходном языке, но при этом в порождённом нами переводе сделали грубые ошибки с точки зрения синтаксиса, то едва ли человек вообще сможет понять, что имелось в виду. Фраза для пользователя будет потеряна, и кто знает, насколько она была важной для восприятия человеком всего переведённого текста.

Кроме того, есть различные методы обработки естественного языка, в которых синтаксическая роль токена используется для, можно сказать, уточнения важности его сигнала. Что это значит? Допустим, мы рассматриваем задачу классификации. Как правило, когда мы составляем по фразе некоторый вектор (например, как мы это уже делали с помощью мешка слов) и пытаемся сопоставить этот вектор какому-то классу в классификаторе, то вклады каждого из токенов оцениваются как равнозначные. Мы уже знаем, что перевзвешивать токены можно, например, с помощью tf-idf, но это не единственный способ. Так, если у нас есть синтаксический парсер, то можно изменить вес каждого слова и с учетом его синтаксической роли. Например, вес у подлежащего может быть больше, чем у определения. Таким образом можно иногда улучшить результат работы классификатора, если включить в один из этапов подготовки данных синтаксический анализ.

Модели структурного синтаксиса

Теперь поговорим о том, какие существуют теоретические модели для построения синтаксической структуры предложения. В лингвистике существует два основных подхода: это грамматика непосредственно составляющих и грамматика зависимостей. Грамматика непосредственно составляющих на английском называется constituency grammar, а грамматика зависимостей называется dependency grammar.

Грамматика зависимостей

В грамматике зависимостей (ГЗ) предполагается, что каждое слово в предложении зависит от какого-то другого слова, и при этом только от одного. Зависимость - это отношение между двумя словами, при котором одно является главным (head), а второе -- зависимым (dependent). Так, в школе нас учили, что главное

слово -- подлежащее, от него зависит сказуемое, и т.д. В грамматике зависимостей главным словом считается глагол-сказуемое. Рассмотрим предложение: «Голодный волк вышел на опушку леса.». В рамках грамматики зависимостей от слова “вышел” зависит “волк”, от “волка” зависит “голодный”, от “вышел” зависит “на”, от “на” зависит “опушку”, и от “опушки” зависит “леса”.

Грамматика непосредственно составляющих

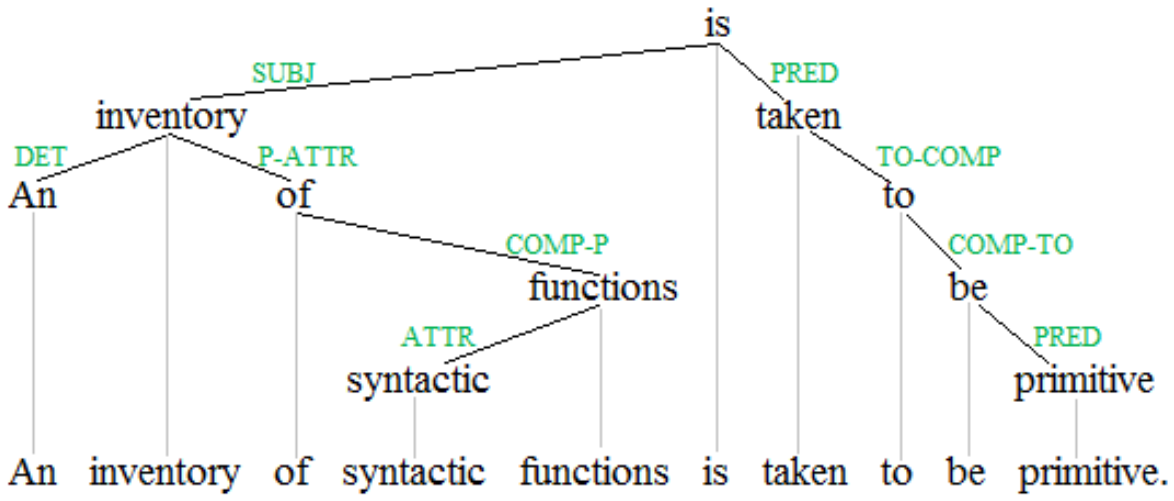
Грамматика непосредственно составляющих (ГНС) исходит из предположения, что некоторые слова в предложении связаны друг с другом теснее, чем с прочими. В ней предложение изначально разбивается на так называемые группы: именную и глагольную, каждая из которых может рекурсивно разбиваться и дальше, пока мы не дойдем до уровня слова. Так, во фразе “Голодный волк вышел на опушку леса.”, есть отдельная группа вышел, есть группа “Голодный волк” и есть группа “на опушку леса”. Мы оперируем понятием “группа”, когда говорим про более близко контактирующие друг с другом слова, или синтаксически связанные. Обратите внимание, что в англоязычной литературе группа называется phrase.

В некоторых языках порядок слов в предложениях почти не вариативен, жёстко закреплён, то есть существует не так много вариантов, как именно можно расставить слова в предложении. Так, например, в немецком языке одна из частей сказуемого практически обязательно стоит на втором месте. А в русском языке порядок слов часто называют свободным. “Голодный волк вышел на опушку леса.” -- “На леса опушку волк вышел голодный “ -- и так далее. Если слова, составляющие одну группу, разнесены в разные концы предложения, как это, например, часто можно увидеть в стихотворениях, то описать синтаксическую структуру такого предложения в рамках грамматики непосредственно составляющих становится очень сложно. Поэтому для русского, как и для других языков с так называемым свободным порядком слов, построение парсера -- то есть автоматического синтаксического анализатора -- на основе грамматики непосредственно составляющих -- не оправдано, и на практике таких парсеров вы не встретите.

В английском языке порядок слов значительно жестче, чем в русском, поэтому для него существует огромное семейство парсеров, работающих в рамках грамматики непосредственно составляющих, phrase-based parsers. У многих *других* языков, подобных в этом смысле английскому, такие парсеры также есть.

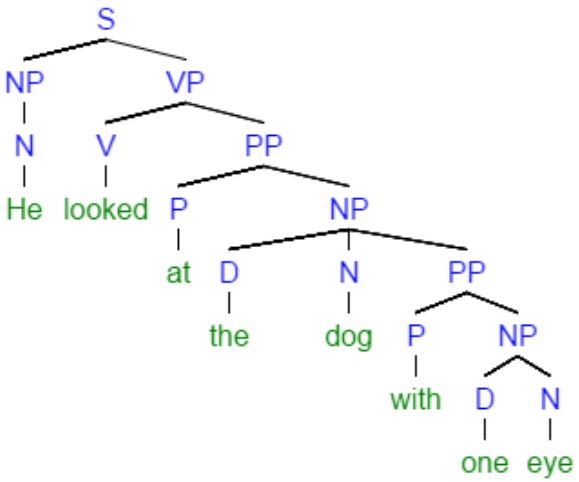
Сравнение и проблемы

Давайте посмотрим, как выглядят разборы, чуть более пристально. Сейчас перед вами на слайде -- пример разбора в рамках грамматики зависимостей. Рассмотрим предложение «an inventory of syntactic functions is taken to be primitive». В нем есть главное слово, глагол, в данном случае is (в случае нескольких глаголов мы бы выбрали один главный), дальше от is есть две группы основных зависимостей: это подлежащее, inventory, и вторая часть сказуемого, taken. Далее это дерево строится в разные стороны, главное, что каждое слово зависит от какого-то другого и при этом связи между словами имеют типы; эти типы связей написаны на экране зеленым цветом. Тип связи -- это, например, подлежащее, сказуемое, прямое дополнение и т.д. Так, можно сказать, что inventory зависит от is “как подлежащее”.



Если мы теперь посмотрим на пример разбора при помощи грамматики непосредственно составляющих, то увидим немного другую картину. Разберем фразу “He looks at the door with one eye”. В грамматике непосредственно составляющих у нас всегда разбор начинается с самой высокоуровневой единицы -- sentence, в верхней части экрана она обозначена как S. В подавляющем большинстве случаев S разделяется на две группы, именную и глагольную. Другими словами, группу подлежащего и группу сказуемого. В данном случае именная группа состоит только из местоимения he. Глагольная группа значительно богаче, в нее входят все остальные слова этого предложения. Она, в свою очередь, разбивается далее на ряд других групп. Список групп в грамматике непосредственно составляющих является конечным, хотя и в нем можно встретить некоторые особенности, в зависимости от языка. Каркас этого списка

-- это глагольная группа verb phrase, именная группа noun phrase, предложная группа adpositional phrase и группа прилагательного adjectival phrase.



Если говорить о том, как ГЗ и ГНС соотносятся друг с другом, то нужно иметь в виду, что:

во-первых, они обе давно и достаточно тщательно разработаны в лингвистике,

во-вторых, отчасти они формально могут быть взаимозаменяемыми, поскольку обе основаны на взаимодействии частей речи.

Мы рассмотрели примеры, которые хорошо разбираются в рамках обеих грамматик. Но, конечно, у каждого подхода есть свои недостатки.

Самая знаменитая проблема грамматики зависимостей -- это представление однородных членов предложения. Если в предложении есть слова, связанные отношением сочинения, т.е. связанные союзом «и» и «или», например: «Вася и Петя пошли в магазин», то трудно сказать, что из этого здесь подлежащее, и кто от кого зависит: Вася от Пети или Петя от Васи. Или оба зависят от чего-то еще и требуется придумать, как тогда они связаны друг с другом.

Грамматика непосредственно составляющих же испытывает, как мы уже упоминали, трудности при “свободном” порядке слов в предложениях.

Современные инструменты автоматического парсинга умеют описывать синтаксическую структуру фразы в рамках обоих формализмов, это лишний раз подчеркивает тезис взаимозаменяемости описываемых грамматик. Тем не менее, хочется еще раз обратить ваше внимание, что не для всех языков описание структуры предложения в рамках обоих формализмов происходит одинаково успешно.

Так, часто можно видеть, что некоторые алгоритмы в автоматической обработке естественного языка показывают высокий результат, но при этом их авторы приводят данные только английского языка, или, например, английского и китайского, который тоже славится жестким порядком слов. И оказывается, что те алгоритмы, которые хорошо улавливают связь слов в одном языке, совершенно не годятся при попытке описать синтаксическую структуру предложения на другом языке.

Давайте рассмотрим проблемы, которые делают автоматический парсинг более сложным.

В первую очередь, это **эллипсис**. Эллипсис -- это пропуски слов. Например: «Вася сегодня ел за обедом борщ, а Петя солянку». Имелось в виду, что “Петя ел за обедом сегодня солянку”, то есть несколько слов были пропущены. И если человек понимает сказанную фразу легко, то у машины могут возникнуть проблемы.

Следующая проблема - **синтаксическая омонимия**. Рассмотрим, что это такое на примере фразы «Он увидел их семью своими глазами». Если допустить отсутствие знаний о реальном мире, то можно легко разглядеть в этой фразе неоднозначность:

либо он сам увидел кого? “их семью”,

либо он увидел кого? “их” при помощи чего? “своих глаз”.

Хрестоматийный пример, который обычно приводят в учебниках, -- это “Эти типы стали есть в цехе”. В этом примере непонятно: “эти типы стали” есть, т.е. металл существует в цехе, либо “эти типы”, какие-то, возможно, неприятные люди, стали питаться в цехе.

Допустим, у нас есть морфологический модуль, на выходе которого работает синтаксический анализ. Тогда понятно, что мы разговариваем сейчас о тех примерах, у которых есть разные леммы, “семью” – это “семь” либо “семья”, а “стали” - это либо “сталь”, либо“стать”. Но есть и более неприятный в этом отношении пример.

Представим, что вы вернулись из командировки, заполняете отчетность и видите фразу: “требуется подпись руководителя группы или командированного лица”. В том случае, если в командировку ездила группа, совершенно понятно, что требуется подпись руководителя группы, а вот в том случае, если командированное лицо было одно, то непонятно, должно ли оно расписываться само или идти к руководителю за подписью. В этом случае уже никакой морфологический модуль спасти нас не сможет. Даже человеку не под силу разобрать этот случай синтаксической омонимии.

Последняя проблема -- это **непроективность**. Что такое “непроективность”? Давайте дадим формальное определение. Представим, что у нас есть некоторое

предложение, и синтаксические связи между словами указаны стрелками прямо над ними. Предложение называется проективным, если ни одна из стрелок не пересекает другую стрелку, и никакая стрелка не накрывает корневую. Корневая стрелка направлена от сказуемого к подлежащему.

Рассмотрим на конкретном примере: «Все мечтают выиграть кубок». На слайде перед вами -- расположение стрелок, это предложение является проективным. Рассмотрим другой порядок слов: «Кубок все выиграть мечтают». Видим теперь, что стрелка от “выиграть” к “кубок” и стрелка от “мечтают” ко “все” пересекаются, накладываются друг на друга. Таким образом, предложение является непроективным, в нем нарушен так называемый принцип пересечения.

Другая перефразировка предложения: «Кубок все мечтают выиграть». Здесь нет никакого пересечения, но корневая стрелка от “мечтают” ко “все” находится строго под стрелкой от “кубок” к “выиграть”. Таким образом, это предложение является непроективным: нарушен принцип обрамления. Иногда те предложения, в которых нарушен только принцип обрамления, называются слабо проективными.

Непроективность представляет из себя отдельную проблему для парсеров, мы поговорим об этом чуть позднее, в следующих разделах. По сути, непроективность представляет из себя формализованное понятие “нехорошего” порядка слов.

Обратите внимание, что непроективность -- это не свойство абстрактного предложения, это свойство конкретного порядка слов, что и видно на примерах выше.

Непроективность -- далеко не редкое явление в языках мира, вот пример из слайдов известного исследователя синтаксического парсинга, профессора Йоакима Нивре. Он свел статистику из разных работ по долям непроективных предложений в языках мира. Тогда оказывается, что в русском языке непроективно всего 10% предложений, что, на первый взгляд, довольно мало, а с другой стороны, довольно

много. Потому что парсеру в каждом десятом предложении придется сталкиваться с непроективным порядком слов. При этом русский язык далеко не рекордсмен; из тех языков, которые анализировал Нивре, рекордсменом является баскский, в котором практически четверть предложений является непроективными. Так что непроективности довольно мало в условном английском, но довольно много в других языках, в том числе и не самых экзотических индоевропейских.

- Non-projectivity in natural languages

Language	Trees	Arcs
Arabic [Hajič et al. 2004]	11.2%	0.4%
Basque [Aduriz et al. 2003]	26.2%	2.9%
Czech [Hajič et al. 2001]	23.2%	1.9%
Danish [Kromann 2003]	15.6%	1.0%
Greek [Prokopidis et al. 2005]	20.3%	1.1%
Russian [Boguslavsky et al. 2000]	10.6%	0.9%
Slovene [Džeroski et al. 2006]	22.2%	1.9%
Turkish [Oflazer et al. 2003]	11.6%	1.5%

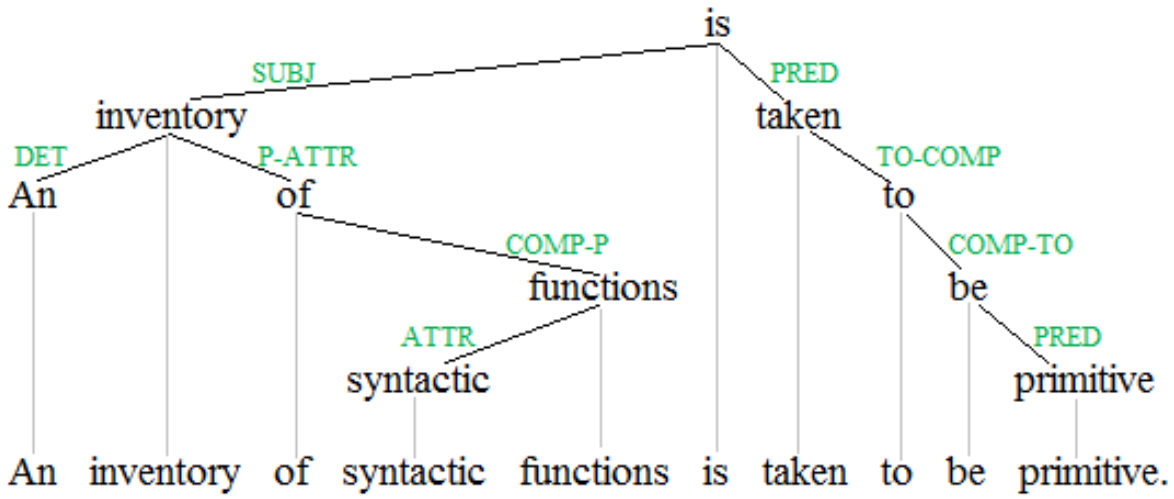
Table from invited talks by Prof. Joakim Niver: Beyond MaltParser - Advances in Transition-Based Dependency Parsing

Дерево зависимостей

Сейчас мы с вами поговорим о синтаксическом разборе -- или парсинге (parsing) в рамках грамматики зависимостей. Для этого сначала вспомним о том, что представляет из себя дерево зависимостей. Если говорить формально, дерево зависимостей - это ориентированный (или направленный) граф, удовлетворяющий следующим условиям:

- Есть специальный корневой узел, в который не входит ни одно ребро
- За исключением этого узла у каждого из прочих есть ровно по одному входящему ребру
- в графе всего одна компонента связности; это, в частности, значит и то,что из корневого узла можно дойти до любого другого единственным путем.

Посмотрим еще раз на пример разбора предложения «an inventory of syntactic function is taken to be primitive». Корневой узел is -- это вершина всего разобранного предложения. Обычно -- это глагол, от которого зависят другие слова; Обратите внимание, что, как мы уже говорили, в этом дереве каждое слово зависит только от одного другого. Иными словами, каждый узел имеет входящее ребро и при этом только одно, и из вершины is можно добраться, переходя по рёбрам в правильном направлении, до любого узла и при этом есть лишь один такой путь. Т.е. граф зависимостей по набору рёбер строится единственным образом.



Мы поговорили о том, как задаётся структура предложения, пора узнать и как нам её находить. Давайте поговорим о том, каким образом может происходить автоматический парсинг в рамках грамматики зависимостей.

Существует два основных подхода: graph-based (в основе которого непосредственное построение графа зависимостей) и transition-based (в основе которой некоторый автомат, последовательно добавляющий рёбра дерева разбора), оба они являются методами машинного обучения с учителем. В данном случае это значит, что потребуются уже вручную аннотированные экспертами-лингвистами предложения -- в рамках грамматики зависимостей -- которые можно было бы использовать для обучения предсказательной модели. Затем при помощи этой модели предсказывают разбор для предложений, которые модель не видела на этапе обучения.

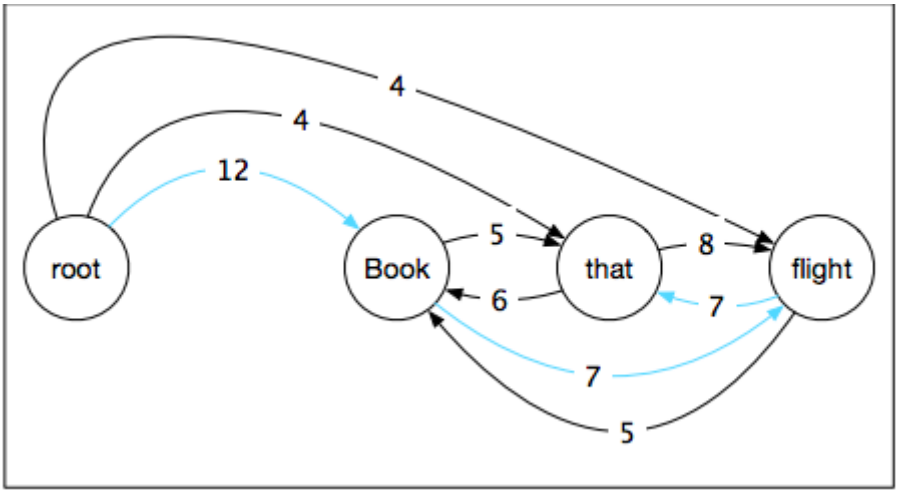
Помимо этих двух основных подходов есть и более маргинальные способы, но мы их затрагивать не будем. Надо сказать, что в отличие от, например, машинного перевода, в котором появление специальных архитектур нейронных сетей в последние годы кардинально улучшило качество самого машинного перевода, в синтаксическом

анализе сами принципы устройства алгоритмов разбора остались теми же. Можно не без оговорок сказать, что роль нейронных сетей пока сводится к более эффективному подбору параметров алгоритмов, что также, конечно, значительно повышает итоговое качество работы этих алгоритмов.

Graph-based dependency parsing

Первый из этих подходов называется «graph-based method». В чем он состоит? Мы берем предложение и строим все возможные типы связей между всеми словами. Рассмотрим пример «book that flight».

Итак, у нас есть три слова; при автоматическом разборе в рамках грамматики зависимостей для главного слова вводится специальный указатель вершины root, чтобы у каждого слова в предложении был узел, от которого оно зависит, -- итого получается 4 узла. При этом root не может иметь входящих ребер. Таким образом, есть root, из которого есть только исходящие ребра; для всех остальных пар узлов (слов) есть ребра в каждую сторону. Мы хотим обучить алгоритм, который бы присвоил этим связям некоторый вес -- аппроксимацию оценки, что именно эта связь является правильной. Таким образом, на этом этапе решается задача регрессии: связи между каждой парой слов нужно присвоить определенный вес. Чем он больше, тем вероятнее, что эта связь есть в разборе.



На примере видно, что мы обучили оценивающую рёбра модель, допустим, она оценила вес связи между root и book как 12, между root и that = 4, между root и flight = 4. То есть, скорее всего, вершиной нашего предложения, его сказуемым, является book. Оценка того, что that зависит от book = 5, а вес обратной зависимости -- 6. Это, конечно, упрощение, потому что на слайде представлено только одно ребро, идущее от

каждого слова в другому слову. На самом деле при обучении таких ребер ровно столько, сколько у нас есть с вами типов связей. Поэтому в ходе обучения мы пытаемся предсказывать: верно ли, что that зависит от book как подлежащее, верно ли, что that зависит от book как определение, и т.д. Для всех этих типов связей соответственно имеется свой вес. После того, как алгоритм отработал, т.е. оценил факт наличия связей между всеми имеющимися словами, мы отбираем дерево, которое имеет наибольший суммарный вес. Это делается при помощи алгоритма нахождения минимального остовного дерева.

Заметим, что в рамках графового подхода проще справляться с непроективностью, так как нам все равно, какой у нас был исходный порядок слов. Мы просто отберем то дерево, которое выиграло с точки зрения нашего алгоритма. Мы не думаем о том, чтобы это дерево было проективным и никакие ограничения при построении дерева нас к этому не будут подталкивать.

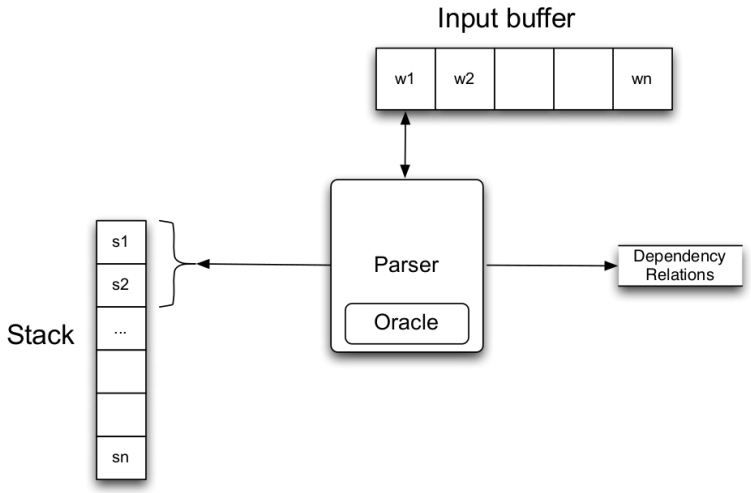
Transition-based dependency parsing

Перейдем к другому подходу, Transition-Based Dependency Parsing. Надо сказать, что на русский язык это словосочетание хорошо не переводится. Что такое Transition-Based? Своё название подход получил из-за того, что, грубо говоря, алгоритм заключается в обучении и использовании некоторой абстрактной машины, переходящей из состояния в состояния в зависимости от текущей ситуации. Звучит не слишком понятно, давайте рассмотрим пример.

Пусть у нас есть фраза «Book me the morning flight», которую мы разбираем постепенно, то есть не все слова нам видны с самого начала. Нам как будто их открывают в том же порядке, в котором они появляются в предложении. Когда у нас с вами в начале есть только слово Book, нам с вами очень трудно представить полный разбор предложения, мы даже не знаем, что такое Book: “заказать” или “книга”. Когда мы с вами видим уже два слова, т.е. когда появляется me, мы понимаем, что это местоимение, которое зависит от book. Book me the не вносит нам дополнительной ясности, да и book me the morning тоже не придаёт никакой уверенности, потому что слово morning может зависеть от слова book, в контексте «забронируй мне утро у стоматолога». Конечно, это чересчур разговорная фраза, но никто не застрахован от некорректного с точки зрения письменной речи примера. Но после того как мы видим последнее слово, flight, всё встает на свои места. Мы специально разбирали

предложение пословно, т.к. похожим образом работают стандартные алгоритмы Transition-Based DP.

Пусть есть список токенов, т.е «Book, me, the, morning, flight», он сложен в некий массив, который называется буфером. Также есть некоторый стэк для токенов, о котором мы поговорим немного позже. Кроме того, есть хранилище уже готовых дуг, которые войдут в итоговое дерево. Всё это -- со всеми заполняющими буфер, стек и хранилище значениями, будем называть конфигурацией системы.



Задан и список операций, благодаря которым мы меняем конфигурацию. Есть три дефолтные операции. Во-первых, можно провести зависимость между первыми двумя токенами на вершине стэка из самого верхнего во второй сверху элемент, то есть “стрелочку налево”. Эта операция называется «LeftArc». Во-вторых, можно сделать то же самое, но провести дугу в другом направлении. Это «RightArc». И, наконец, операция «SHIFT» -- перенос токена из буфера в стек.

А также есть некоторый “оракул” -- предсказательный алгоритм, который по текущей конфигурации системы предсказывает, какая операция должна быть выполнена следующей. Например, таким оракулом может быть нейронная сеть. Существует разные парсеры, у которых отличаются лишь архитектуры нейронных сетей - “оракулов” и способы представления текущего состояния для подачи им на вход; иногда не так плохо работают довольно простые. Об их внутреннем устройстве мы говорить не будем.

Давайте рассмотрим это на нашем примере. Сначала у нас есть стек, в нем находится только элемент ROOT, корень. И есть список токенов «Book, me, the, morning, flight» в буфере.

Берём первое слово из буфера, в данном случае book, и переносим в стек. В стеке оказываются два слова root, book. А в буфере -- все слова, кроме book. Остаются me, the, morning, flight.

На этом шаге мы еще не знаем, что book является тем самым сказуемым, тем самым root. Мы с вами видим только book, и не видим ни одного другого слова, значит, мы, как и было при разборе ранее, не знаем, существительное это или глагол. Поэтому парсер в идеале применит в данном случае операцию SHIFT, перенесет следующие слово me из буфера в стек. В этом случае, в стеке -- root, book, me, а в буфере the, morning, flight.

На этом шаге мы с вами знаем, что me зависит от book, точно так же, как мы с вами поняли это, когда пословно шли по предложению. Мы проводим стрелочку от book к me, направо, к последнему токену, который мы видели, и зависимое слово me выкидывается из стека. Соответственно, на следующем шаге мы имеем в стеке “root, book”, а в буфере “the, morning, flight”.

После этого ничего не остается, кроме как последовательно применять операцию SHIFT, перенося токены из буфера в стек, пока мы не останемся с полным стеком и пустым буфером. На этом шаге мы уже сможем провести все связи (так же, как мы все поняли на примере, лишь дойдя до конца предложения).

Итак, имеем верхушку стека, в ней содержатся morning и flight. morning зависит от flight, проводим стрелочку налево и выкидываем зависимое, в нашем случае это morning. У нас остается в стеке root, book, the. Проводим стрелочку от flight к the и выкидываем артикль, остаются root, book, flight. Понимаем, что flight зависит от book, проводим стрелочку от book к flight, и выкидываем flight.

На следующем шаге у нас в стеке остаются root и book. Заметьте, это точно такое же положение, которое у нас было в самом начале, когда мы только перекинули book в стек. Но на этом шаге мы уже видели всё предложение и понимаем, что book это точно наш корень, проводим стрелочку от root к book, и выкидываем слово book.

Step	Stack	Word List	Action	Relation Added
0	[root]	[book, me, the, morning, flight]	SHIFT	
1	[root, book]	[me, the, morning, flight]	SHIFT	
2	[root, book, me]	[the, morning, flight]	RIGHTARC	(book → me)
3	[root, book]	[the, morning, flight]	SHIFT	
4	[root, book, the]	[morning, flight]	SHIFT	
5	[root, book, the, morning]	[flight]	SHIFT	
6	[root, book, the, morning, flight]	[]	LEFTARC	(morning ← flight)
7	[root, book, the, flight]	[]	LEFTARC	(the ← flight)
8	[root, book, flight]	[]	RIGHTARC	(book → flight)
9	[root, book]	[]	RIGHTARC	(root → book)
10	[root]	[]	Done	

Наконец перед нами точная картина идеального разбора. Буфер пуст, все дуги проведены, а в стеке остался один-единственный элемент root. Заметьте, все манипуляции с токенами происходили исключительно на вершине стека, действия либо с буфером, либо с двумя верхними элементами стека порождали новую конфигурацию. Перед нами пример правильного разбора. Root это -- book, и все остальные связи также проведены в соответствии с идеальным (человеческим) разбором.

Увы, не всегда приходится работать с хрестоматийными примерами, которые подобраны под эту архитектуру. Проблемы появятся, как только мы попробуем разобрать непроективное предложение с помощью transition-based парсера. Есть строгое доказательство, что Transition-Based может разобрать любое проективное предложение. Естественно, это теоретический результат, а на деле все Transition-Based парсеры, как и любые другим алгоритмы машинного обучения, всегда где-то да ошибаются.

Рассмотрим простейший пример непроективного предложения, «Я мальчика вижу красивого». У нас нарушен принцип пересечения, стрелка от “вижу” к “я” пересекается со стрелкой от “мальчика” к “красивого”.

Попробуем разобрать это предложение в рамках Transition-Based парсера, вообразить его работу. На первом шаге у нас как всегда есть стек, в котором есть только корень, и буфер, в котором содержаться все токены: мальчика, красивого и т.д. На первом шаге мы добавляем “Я” в стек, и больше ничего не можем сделать. “Я” явно не является сказуемым, стрелку от root к “я” мы не проведём. Далее, в стеке root, “я” и “мальчика”, и мы опять же ничего не можем сделать, ведь “мальчика” никак не зависит от “я”. Добавляем “вижу”; мы и сейчас ничего не сделаем, ведь применение LeftArc нас оставит с “вижу” и “красивого” в стеке. Так что можем добавить слово “красивого” и убедиться, что даже с идеальным парсером мы здесь бессильны. У нас с вами пустой

буфер и полный стек. Мы не провели ни одной связи, а значит, не получили синтаксического дерева.

Шаг	**Стэк**	**Буфер**
0	[root]	[я, мальчика, вижу, красивого]
1	[root, я]	[мальчика, вижу, красивого]
2	[root, я, мальчика]	[вижу, красивого]
3	[root, я, мальчика, вижу]	[красивого]
4	[root, я, мальчика, вижу, красивого]	[]

Однако, как мы уже заметили, на одном из последних шагов будто бы можно провести связь между “вижу” и “мальчика” и тогда, как кажется, ситуация станет лучше. Давайте попробуем так и сделать. Но тогда по правилам Transition-Based parsing, придется выкинуть зависимое слово, т.е. слово “мальчика”. Тогда, как можно видеть на экране, на шаге 4 в разборе останется root, “я” и “вижу” -- в стеке и “красивого” -- в буфере. Еще мы выкинем “я” и на 6 шаге останемся со стеком root, “вижу” и “красивого”.

Чтобы получить хоть какой-то разбор, придётся соединить “вижу” и “красивого”. Но это неправильно. Слово “красивого” на самом деле зависит от слова “мальчика”. Мы сделали единственный ход, который имелся в нашем распоряжении, и получили в результате ошибочный разбор.

Шаг	**Стэк**	**Буфер**	**Операция**
0	[root]	[я, мальчика, вижу, красивого]	SHIFT
1	[root, я]	[мальчика, вижу, красивого]	SHIFT
2	[root, я, мальчика]	[вижу, красивого]	SHIFT
3	[root, я, мальчика, вижу]	[красивого]	LeftArc
4	[root, я, вижу]	[красивого]	LeftArc

5	[root, вижу]	[красивого]	SHIFT
6	**[root, вижу, красивого]**	**[]**	**RightArc**
7	[root, вижу]	[]	RightArc
8	[root]	[]	

Иными словами, Transition-Based модели, если обобщить, не могут работать с непроективными предложениями. Поэтому был предложен ряд модификаций Transition-Based парсеров для того чтобы бороться с непроективностью. На экране -- список методик, на которых мы подробно останавливаться не будем, обсудим лишь способ, который завоевал особенную популярность.

В предлагаемом подходе добавляется 4-я операция. То есть у нас есть не только операции LeftArc, RightArc и SHIFT, но и операция SWAP: вернуть *второй* элемент стека обратно в буфер, тем самым изменив порядок токенов в буфере. Автор идеи и статьи, уже знакомый нам Йоаким Нивре, замечает, что лучше применять эту операцию с некоторым штрафом для алгоритма, то есть по большей части в случаях, когда нет иных вариантов.

Допустим, есть предложение вида A B, тогда если вернуть при помощи операции SWAP A в буфер, а потом обратно в стек, то мы, можно сказать, получим строку BA. Иными словами, так делается reordering, то есть изменение порядка токенов в исходном предложении. Как мы помним, непроективность является свойством конкретного порядка слов, а раз его можно поменять в исходном предложении, предлагаемый подход способен помочь, “превратив” непроективное предложение в проективное.

Рассмотрим, как это работает, на том же примере. Вот уже известный нам разбор, когда мы по максимуму добавляем все в стек. Однако на 4 шаге, когда мы добавили в стек всё, что могли, мы (или наш “оракул”), вооружившись нашим новым инструментом, можем применить операцию SWAP и вернуть второй элемент стека обратно в буфер. В нашем случае это слово “вижу”. И дальше можно провести зависимость между двумя элементами на верхушке стека, т.е. между словами “мальчика” и “красивого”, вернуть из буфера обратно в стек слово “вижу” и провести

все нужные зависимости. От “вижу” к “мальчику”, от “вижу” к “я”, от “root” к “вижу”. Так, в результате перед нами окажется правильное дерево разбора.

Шаг	**Стэк**	**Буфер**	**Операция**
0	[root]	[я, мальчика, вижу, красивого]	SHIFT
1	[root, я]	[мальчика, вижу, красивого]	SHIFT
2	[root, я, мальчика]	[вижу, красивого]	SHIFT
3	[root, я, мальчика, вижу]	[красивого]	SHIFT
4	[root, я, мальчика, вижу, красивого]	[]	**SWAP**
5	**[root, я, мальчика, красивого]**	**[вижу]**	RightArc
6	[root, я, мальчика]	[вижу]	SHIFT
7	[root, я, мальчика, вижу]	[]	LeftArc
8	[root, я, вижу]	[]	LeftArc
9	[root, вижу]	[]	RightArc
10	[root]	[]	

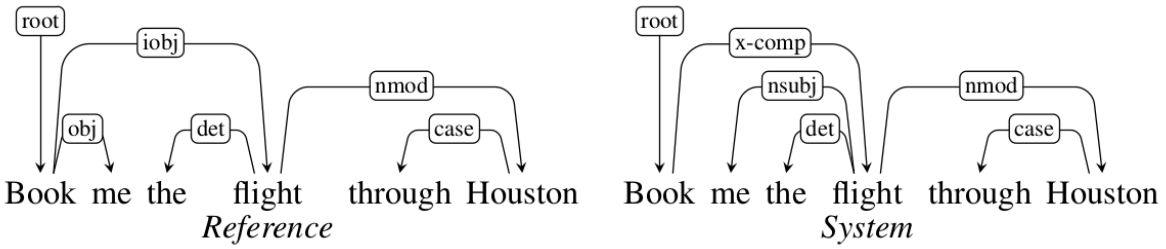
Метрики качества

Поговорим об оценке качества работы парсеров, работающих в рамках грамматики зависимостей. В сущности, все, что нам дает автоматический синтаксический анализ, -- это два типа данных: это, во-первых, сам факт связи, то есть зависимости, между словами, во-вторых, направление этой связи (то есть начало и конец дуги), и, в-третьих, тип этой зависимости. Эти вещи и измеряют метрики оценки качества парсера.

Одна из них называется «Unlabeled Attachment Score», UAS, и оценивает лишь сам факт того, что мы для каждого слова правильно приписали ему начало дуги (то есть от какого слова оно зависит), а вторая -- «Labeled Attachment Score», LAS, оценивает тот факт, что мы для каждого слова правильно приписали как начало дуги,

так и тип метки. Так как можно сказать, что LAS -- это “UAS с дополнительными требованиями”, ясно, что значение LAS никак не может быть больше, чем UAS.

Рассмотрим это на примере «Book me the flight through Houston», и два разбора. Слева разбор, являющийся эталонным, т.е. размеченный живыми экспертами, людьми. Справа разбор, предложенный некоторым алгоритмом, некоторой системой. Если не обращать внимания на ярлыки связей, которые здесь написаны, то есть смотреть от какого слова к какому проведена стрелочка, то легко обнаружим ошибку в вершине слова me: система считает, что это зависимое от слова flight, в то время, как в размеченном примере оно зависит от слова book. У всех остальных слов вершины точно такие же, как и в разборе. Значит, UAS равен 5/6.



Перейдем к «Labeled Attachment Score». Он уже как максимум %, он не может быть больше. Обратившись к типам зависимостей, видим, что они совпадают у всех токенов, чьи начала дуг мы определили верно, кроме связи от book к flight. Нам даже не так важно, какой именно тип зависимости прописан в разметке, и какой предложила система, мы видим, что они разные. У всех остальных токенов типы зависимостей совпадают. Значит, из имевшихся ранее 5/6 мы должны вычесть еще 1/6, и таким образом получается «Labeled Attachment Score» 4/6 .

Кроме того, метрики бывают, можно сказать, «macro-averaged” (то есть они вычисляются для каждого предложения по отдельности и затем усредняются) и “micro-averaged». В этом случае, грубо говоря, мы воспринимаем все имеющиеся наши предложения как единый текст, и вычисляем оценки на нём как на одном предложении. Казалось бы, есть смысл использовать accuracy, что мы с вами и делали, брали долю правильных связей относительно общего количества. Однако пока отложим этот вопрос в сторону и поговорим об одном проекте, связанным с разбором в рамках грамматики зависимостей, который эту проблему решал.

Речь пойдет о «Universal Dependencies», это самый известный проект по парсингу зависимостей и, что, наверное, важнее для нас, он довольно актуален. Организаторы регулярно проводят так называемые дорожки, то есть соревнования по

качеству разбора, и это вовсе не история науки, это очень свежие данные. Родился этот проект как попытка решить две проблемы:

- во-первых, лингвистическую проблему, т.е. несоответствие терминов и правил из грамматик зависимостей разных языков. К примеру, есть правила для английского языка. Но там, скажем, есть артикли, которых нет в русском. Или, например, вряд ли вы когда-то в школе выделяли в английском тексте обстоятельства, дополнения и т.д. Есть смысл обзавестись системой обозначений, которая бы позволила единообразно размечать тексты как на английском, так и на русском (и еще на сотнях других языков), и затем предсказывать подобные синтаксические структуры на основе текстов на разных языках.
- Во-вторых, есть и амбициозная вычислительная задача обучения синтаксического парсера сразу для многих языков. Есть хрупкая надежда получить особый синтаксический алгоритм машинного обучения, который бы брал на вход размеченные данные на некоторым языке, совершенно неважно каком, пусть даже “племени мумба-юмба”, и выдавал нам бы на выходе некоторую разметку на тех предложениях, которых он еще не видел.

Героическим авторам этого проекта удалось собрать более чем сто трибанков (корпусов синтаксических деревьев) для более чем 70 языков, причем не только хорошо известных нам индоевропейских, но и более экзотических языков, например, используемых в Африке. И теги, то есть метки зависимостей, что очень важно, унифицированы, т.е. каким бы ни был язык, мы конечным набором в буквально 20-30 тегов можем разметить все имеющиеся в нем синтаксические связи.

В 2017-2018 годах, а также в 2020 году организаторы проекта «universal dependencies» также проводили дорожки, т.е. соревнования по синтаксическому разбору. Точнее, не только синтаксическому разбору, так как в рамках этих соревнований участникам предлагалось, взяв на вход сырой текст, дойти до разметки в рамках грамматики зависимостей. Поскольку грамматика зависимостей строится поверх морфологии, а та -- поверх токенизации, то на самом деле участники должны были подготовить целый пайплайн обработки данных.

В соревновании в 2017 году использовался 81 трибанк, 49 языков, а в 2018 году алгоритмам требовалось разметить данные по аж 57 языкам. И участникам был дан выбор: либо они могли получать на входе абсолютно сырой текст, т.е. писать все те модули, о которых мы говорили, самостоятельно, а именно: разбиение на предложения,

затем на токены, простановку морфологических меток у токенов, и только затем синтаксический разбор. Либо, второй вариант, можно было взять любой из этих алгоритмов у организаторов, не идеально, но работающий с некоторым гарантированным качеством. Иными словами, можно было сделать всё от сырого текста до синтаксического дерева, а можно было подготовить только синтаксический разбор на всём готовом.

Что не может не радовать -- это то, что один из лучших результатов среди всех команд и всех трибанков был показан на материале русского языка. Ничего удивительного нет в том, что в 2018 году, метрики, полученные на корпусе русского языка, немного упали в связи с тем, что трибанки меняются. Также имело место другое разделение на обучающую и тестовую выборку.

То, что нас интересует особенно: каким образом организаторы оценивали эти пайплайны?

Поскольку кто-то выбрал сырой текст, то есть токенизацию некоторые участники использовали свою, возникает вопрос, как именно оценивать качество. При нестандартной токенизации выходит, что если мы где-то ошиблись, то синтаксический модуль уже априори не мог отработать так же, как это предполагается в размеченных данных, ведь он пытался провести синтаксические связи не между реальными словами, а между чем-то, что породил частный токенизатор, потенциально неправильно.

Это означает, что мы должны оценивать не только *точность*, т.е. количество точных попаданий относительно всех предсказаний, которые мы сделали, но и *полноту*, которая в данном случае является количеством точных попаданий, которые мы сделали относительно количества связей в исходных размеченных данных. Количество связей в исходных данных -- это, в сущности, и есть прямое следствие токенизации.

Поскольку по определению в дереве зависимостей у нас у каждого слова есть вершина, и при том только одна, то количество связей всегда равно числу токенов. Соответственно, если у нас верная токенизация, то, как можно видеть на экране, точность совпадает с полнотой, а F-мера совпадает с accuracy. А вот при неидеальной токенизации точность и полнота не равны друг другу, а значит, и F-мера не равна accuracy.

Значит, мы возвращаемся к тому, о чем я говорил в начале нашего раздела: получается, что брать accuracy не очень хорошо, когда и если мы можем неверно токенизировать. С другой стороны у организаторов соревнования получилось, что мы

будто бы хотим оценивать синтаксическую метрику, но на самом деле, если мы плохо отработали где-то в самом начале, на токенизации, то это скажется на нашей с вами синтаксической метрике.

Получается, что, во-первых, нет метрики, которая хотя бы теоретически выделяет идеальный парсер, во-вторых, у нас нет единой метрики, которая позволяет нам сравнить все пайплайны целиком, потому что метрика вроде бы завязана на синтаксис, но при этом оценивает весь пайплайн и при этом не делает это очень корректно.

Допустим, мы сделали гениальную токенизацию, но плохой синтаксис, и получились достаточно высокие результаты, а кто-то написал очень плохую токенизацию, но гениальный синтаксический парсер, и цифры у них получились близкие к нулю лишь потому, что связи проводились между неправильными юнитами. Тогда тот факт, что эта команда написала гениальный синтаксический парсер, останется человечеству не известен.

И как делать иначе -- не очень понятно! Попробуйте сами изучить те метрики, которые организаторы выработали, на опыте соревнования 2017 года. Метрики поменялись в 2018 году, но никакого общепринятого решения обозначенной нами проблемы там нет. Просто появились новые метрики, которые оценивают не только конечную работу пайплайна, т.е. его синтаксическую часть, но и то, что получилось на уровне морфологии.

Инструменты

Переходим к практической части, сейчас речь пойдёт об инструментах, предназначенных для автоматического парсинга в рамках грамматики зависимостей.
Важно заметить, что не стоит ограничиваться инструментами, о которых сейчас пойдет речь, нужно всегда проводить самостоятельное исследование со сравнением альтернатив под каждую целевую задачу, а также быть в курсе, что нового появляется в области. Впрочем, это применимо не к одному только синтаксису, а к любой области data science и, в частности, к обработке естественного языка.

Ранее, в разделе об оценке качества синтаксического разбора мы узнали, что существуют синтаксические дорожки, особые соревнования для определения лучшего парсера. Можно воспользоваться их результатами и посмотреть, какие именно парсеры насколько хорошо разбирают предложения.

Однако, поскольку в этих дорожках было много академических участников, а также закрытых разработок, не всегда получается так, что можно просто взять какую-то таблицу результатов определенного соревнования и отобрать лучшие инструменты для своих целей. Бывает так, что закрытые разработки находятся под nda (соглашение о неразглашении), а те, кто их разработал, не могут публиковать свой код, авторские права принадлежат компании.

Академические же разработки иногда отличаются тем, что ввиду разного подчас отношения к качеству выкладываемого программного кода, с авторской идеей бывает сложно разобраться и воспроизвести результат.

Поэтому, когда мы говорим об инструментах, на первый план выходят такие параметры, как удобство интерфейса и простота использования. По этим двум параметрам с большим отрывом побеждают два парсера, о которых пойдёт речь. Это UDPipe и SyntaxNet.

Ранее мы говорили об архитектурах; и SyntaxNet и UDPipe изначально transition-based парсеры, но не следует считать, что transition-based парсеры намного лучше graph-based парсеров. К примеру, на дорожке 2017 года победил graph-based парсер, который, кстати, является академической разработкой, и опыт показывает, что запустить выложенную авторами реализацию не так-то просто! После этого в дорожке 2018 года подавляющее большинство участников просто поменяло свою архитектуру.

Авторы UDPipe уже давно работают с graph-based подходом, но публичный релиз инструментов новой архитектуры еще не состоялся. Сейчас мы всё же поговорим о старой, но вполне конкурентоспособной версии UDPipe, то есть transition-based. Можно предположить и что SyntaxNet тоже совсем скоро станет graph-based. Не нужно забывать, что после такой резкой переориентации и ставки на graph-based архитектуру сейчас считается, что за ними будущее. У этого, правда, есть свои оговорки, но о них немного позже.

В качестве UDPipe брали на самом деле версию с graph-based архитектурой, для SyntaxNet последние выпущенные Google-ом доступные данные относятся к 2017 году. Поскольку мы, в первую очередь, работаем с русским языком, то данные собраны именно для него. Здесь видно, что и по UAS, и по LAS UDPipe превосходит SyntaxNet. И, на самом деле, самые важные для нас данные содержатся не в первых двух строках этой таблицы, а в последних трех.

	UDPipe (Future)	**Syntaxnet (parseysaurus-17)**
UAS (russian, syntagrus)	92.96%	92.67%
LAS (russian, syntagrus)	91.46%	88.68%
Время парсинга одного предложения	~ 3 ms	~ 100 ms
Возможность “распилить” пайплайн	+	-
Запуск напрямую без докера и др.	+	-

Когда работаем с высоконагруженными системами, а не исключительно академическими разработками, сравнивающими подходы скорее с теоретической точки зрения, на первый план подчас выходят не самые ценные на первый взгляд удобства. В первую очередь, это скорость работы системы. Видно, что UDPipe превосходит SyntaxNet в значительное число раз, на несколько порядков. Это очень важно. Представьте себе, что внутри какой-нибудь Алисы или Яндекс.Поиска встроен как один из этапов обработки текста синтаксический парсер, и в одном случае мы тратим лишь на него 3 миллисекунды, а в другом -- целых 100 миллисекунд. Кроме того, UDPipe предоставляет нам возможность распилить, так сказать, пайплайн, т.е., например, мы можем заменить в цепочке обработки встроенную токенизацию своей. Мы не обязаны учить модель на парсинг сырого текста, от строки до синтаксического разбора.

Мы можем подать на вход как на обучении, так и на этапе предсказания -- и уже токенизированный текст, и морфологически уже проанализированный, в зависимости от наших целей. А вот в случае SyntaxNet мы полагаемся на работу “черного ящика”, то есть мы подаем на вход строку и ждем, какой у нас будет выход в виде синтаксического проанализированного текста без возможности на что-то повлиять.

Также для запуска SyntaxNet, скорее всего, придётся скачать докер-образ, потом развернуть его локально, и как-то подать свой запрос; это напоминает черный ящик еще и по способу взаимодействия, не только в плане прозрачности и настраиваемости

работы; в то время как UDPipe позволяет задать запрос напрямую хоть из командной строки, хоть через Python-интерфейс, и все это можно модифицировать как вам удобно.

Обратите ещё раз внимание на последние три строчки, они, возможно, не менее важны, чем собственно качество работы парсера. Далее мы не будем говорить о SyntaxNet -- нам кажется, что табличка довольно ясно иллюстрирует, почему при работе с русским языком лучше пользоваться UDPipe. Это, разумеется, не значит, что не стоит никогда пользоваться SyntaxNet, но личный опыт говорит, что UDPipe на момент первой половины 2020 года лучше, быстрее, и надежнее.

Итак, UDPipe -- это пайплайн, название, собственно, так и расшифровывается «Universal Dependencies pipeline», обучаемый токенизации лемматизации, морфологическому тэггингу и парсингу, основанному на грамматике зависимостей. Здесь несколько ссылок на статьи об архитектуре, правда уже устаревшей. Напомню, UDPipe уже graph-based, но и про его новую архитектуру уже тоже есть статья; а говорим мы о старой в том числе потому, что последний релиз на момент первой половины 2020 года всё ещё transition-based, и, скорее всего, именно с ним вам предстоит столкнуться в задачах по курсу.

Для UDPipe есть готовые модели, т.е. если вам нужен какой-то синтаксический парсер для какого-то языка, можно не тратить время на обучение, а просто взять готовый парсер.

Когда мы учим модель, мы вынуждены часть “высокоуровневых” параметров задать самостоятельно, это так называемые гиперпараметры: число итераций, влияние регуляризации и так далее. Мы уже с ними сталкивались. У UDPipe тоже десятки параметров, чтобы мы не мучились с тем, чтобы воспроизвести ту самую модель UDPipe, которая показала определенное качество на определенном наборе данных, в репозитории указано, с какими именно параметрами училась модель. Всё это важно, ведь в машинном обучении много внимания уделяется воспроизводимости результата, а здесь, таким образом, она гарантирована и тем же набором данных, и тем же набором параметров, что у проводивших эксперимент. Кроме того, авторы потратили какое-то время, перебирая параметры, так что на предлагаемые ими значения можно ориентироваться если не как на лучшие, то как на обеспечивающие гарантированно достойное качество.

Пайплайн UDPipe работает следующим образом: сначала требуется разбить на предложения, потом разбить на слова (токенизировать), затем провести расстановку морфологических меток (в том числе частей речи), параллельно с этим выполняется

лемматизация, а после этого -- синтаксический разбор. Итак, по этапам:

1. На слова и предложения делит двунаправленная однослойная рекуррентная сеть GRU, которая для каждого символа предсказывает, последний ли он в предложении или в токене. Соответственно, если он последний в предложении, этот символ и является границей предложения. Если некоторый символ последний в токене, значит, здесь проходит граница между токенами.

2. Переходим к работе морфологической разметки, или теггинга. Устроена здесь она своеобразно: она ориентируется на последние 4 символа каждого слова. По ним и только по ним она пытается понять, какая это могла быть часть речи и какими морфологическими свойствами она могла бы при этом обладать. Дальше при помощи перцептрона, то есть неглубокой нейронной сети прямого распространения, выбирается лучший кандидат. Для примера рассмотрим форму глагола «переведи». Что говорит нам теггер UDPipe? Мы смотрим на последние 4 символа, т.е. “веди”, и думаем, чем это могло бы быть. Это, с одной стороны, может быть повелительным наклонением глагола, как в контексте «переведи бабушку через дорогу». С другой стороны, это еще похоже, скажем, на предлоги, такие, как “среди”, “посреди”. У предлогов нет изменяемых грамматических категорий, т.е. так и предсказываем -- только часть речи, предлог. И, соответственно, работа теггера состоит в генерации этих самых гипотез, оценке их уместности и выборе лучшего кандидата. Так, в нашем примере для «переведи» мы должны понять, что же это: глагол в повелительном наклонении или предлог.

3. При этом работает также и схожим образом лемматизатор. Он генерирует пары: слово -- потенциальная лемма. Мы смотрим на слово и пытаемся понять, что же это могло быть. Так, если мы видим незнакомые слова, которые пришли к нам из другого языка, например, допустим, мы никогда не видели слова “заюзать”. И вот перед нами пример “какой-то программист заюзал новую библиотеку”, и требуется лемматизировать слово «заюзал». Видим приставку “за”, видим в конце “л”, которая похожа на показатель прошедшего времени глагола мужского рода, и получается, что лемма -- глагол “заюзать”. В самых общих чертах, конечно, так же работает лемматизатор, который встроен в UDPipe. И точно так же он генерирует и отбирает лучшего кандидата. Как мы уже заметили, процесс предсказания тегов и процесс предсказания лемм -- это похожие процессы. Поэтому в UDPipe можно соединить эти модели в одну общую. Однако, как показывает практика, если вы работаете с языком с богатой морфологией, например, с русским, эти модели лучше разделять. Иначе

одна-единственная модель просто не успевает выучить всю морфологию, которой наш язык богат.

Кроме всего описанного, можно также подключать свой список лемм, это очень ценно, если система не может сама по вашим данным выучить какое-то правило лемматизации, то ей можно помочь, явно показав, какие еще правила образования лемм для каких слов существуют.

4. Переходим собственно к парсеру, он работает по так называемой архитектуре arc-standard, той самой, которую мы с вами рассматривали, когда говорили о transition-based парсерах.

Мы говорили, что за выбор действия парсера отвечает некоторый предсказательный механизм, который использует для этого текущую конфигурацию системы. Здесь он -- очень простая нейронная сеть, которая смотрит на 3 элемента на вершине стека, 3 элемента на вершине буфера и дальше -- по свойствам собственно уже набранным в будущее дерево элементов. Если есть три элемента в стеке, и при этом есть три элемента в буфере и есть какие–то будущие потомки, т.е. какая-то конфигурация, то всё это подаётся на вход сети, а она, в свою очередь, предсказывает следующий шаг, приводящий систему к новой конфигурации.

Кроме того, что важно и, как показывает практика, это помогает в синтаксическом парсинге, можно загрузить семантические эмбеддинги, такие, как упоминавшийся word2vec, т.е некоторые векторные представления форм или лемм. Зачем это нужно? Рассмотрим фразу «Мышь ест стол». Мы совершенно точно понимаем, о чем здесь идет речь, но машина, если она не обладает никакими семантическими представлениями или знаниями об окружающем мире, может и перепутать здесь объект и субъект. Тогда она сочтёт, что это стол является некой сущностью, которая поедает несчастную мышь. И именно функция загрузки и использования семантических эмбеддингов помогает модели правильно разделить субъект и объект.

Перед нами простая схема, как все это происходит. В модель грузится всё: текущая конфигурация дерева, грузятся части речи, грузятся сами слова, все это проходит через некоторый скрытый слой, и в дальнейшем последним слоем является softmax -- функция, которая выдаст нам распределение, какое действие мы должны предпринять. Их у нас, я напомню, всего четыре: SHIFT, LeftArc, RightArc, SWAP.

На этом мы заканчиваем практическую часть с обсуждением инструментов.