Алгоритмы сегментации текста

mishadoff

Здравствуйте.

В контексте анализа данных из твиттера возникла задача обработки хештегов. Нужно было взять хештег и разбить его на отдельные слова (#habratopic => habra topic). Задача казалась примитивной, но, получается, я ее недооценил. Пришлось перебрать несколько алгоритмов пока не было найдено то, что надо.

Эту статью можно считать некой хронологией решения задачи с анализом преимуществ и недостатков каждого из использованных алгоритмов. Поэтому, если вам интересна данная тема, прошу под кат.

Стоит отметить, что задача разбивки большого текста без пробелов очень часто встречается в nlp. Это определение слов в немецких «длинных словах», которые, по сути, являются конкатенацией нескольких (**geschwindigkeitsbegrenzung** — *ограничение скорости*), определение слов в китайском письме, где редко пользуются пробелом (**城市人的心爱宠物** — *любимое домашнее животное городских жителей*) ну и так далее. Если во втором случае с китайским, самый простой алгоритм считает один иероглиф за слово и работает вполне приемлемо, то с немецким обстоит все намного сложнее.

Алгоритм 1. Minimum Matching

Проходимся по строке и находим первое слово которое матчится. Сохраняем это слово и повторяем процедуру для остатка строки. Если в последней строке не матчится ни одно слово, считаем что сегментация не найдена. Алгоритм очень быстрый (как раз то, что нам надо), но очень глуп.

Пример: *niceday => nice day*. Но, для *niceweather* сегментация не будет найдена, т.к. после найденого слова *nice*, алгоритм определит *we*, потом артикль *a*, далее *the*, а слова *r* в нашем словаре нету. Хм. Что мешает вместо первого слова, которое матчится, брать то, у которого максимальная длина?

Алгоритм 2. Maximum Matching или Greedy

Делаем все то же самое, как и в первом случае, но всегда выбираем слово с максимальной длиной. Алгоритм медленнее предыдущего, так как нам нужно идти с конца строки чтоб определить слово с максимальной длиной первым. Скорость заметно упадет если нужно будет обрабатывать очень длинные строки, но так как у нас данные из твиттера, забиваем на проблему. (На самом деле, если выбирать не всю строку, а первые n символов, где n — максимальная длина слова в словаре, то скорость будет в среднем такая же как и у первого алгоритма).

Пример: *niceweather => nice weather* Но, для *workingrass* сегментация опять же не будет найдена. Первое слово, которое заматчит наш алгоритм будет *working*, а не *work* и которое также поглотит первую букву в слове *grass*. Может нужно скомбинировать каким-то образом оба алгоритма? Но, как тогда быть со строкой *niceweatherwhenworkingrass*? В общем пришли к брутфорсу.

Алгоритм 3. Bruteforce

Генерируем все возможные варианты разбиения строки на слова обычной рекурсивной функцией. Таких вариантов будет 2^(N-1), где N — размер строки. Далее производим отсеивание тех вариантов, в которые попали подстроки не из словаря. И полученный вариант будет верным. Главная проблема алгоритма — скорость. Стоп! А зачем генерировать все, а затем производить фильтрацию, если можно генерировать сразу то, что нужно.

Алгоритм 4. Clever Bruteforce

Модифицируем нашу рекурсивную функцию так, чтобы рекурсивный вызов происходил тогда, когда мы уже заматчили слово из словаря. В таком случае, генерируется сразу нужная сегментация. Алгоритм очень быстр, дает нужный результат, и вообще я подумал, что задача решена. К сожалению, я упустил из виду неоднозначность (ambiguity). Дело в том что сегментация строки не уникальная и бывают случаи, когда существуют десятки равнозначных разбиений.

Пример: *expertsexchange* => (*expert sex change*, *experts exchange*)

Появилась новая подзадача: как выбрать «правильную» сегментацию? Перебрал варианты первую, случайную, последнюю, ту в которой больше слов, ту в которой меньше слов и результаты были, мягко говоря, не очень. Нужен был какой-то более умный алгоритм. Я же могу понять что *dwarfstealorcore* это скорей всего «дварф крадет руду орков», а не «дварф крадет или ядро», значит и машина должна понимать. Тут на помощь пришли алгоритмы machine learning.

Алгоритм 5. Clever Bruteforce with ambiguity resolving (unigram model)

Для того, чтобы научить нашу программу решать неоднозначности, мы скармливаем ей большой текстовый файл (трейн-сет), по которому она строит модель. В нашем случае, унограмная модель, это частоты употребления каждого слова в тексте. Тогда для каждого из кандидатов на сегментацию мы считаем вероятность, как произведение вероятностей каждого слова в кандидате. У кого вероятность больше, тот и выиграл. Все просто.

Пример: *input =>in put* Неожиданно? Просто в тексте очень часто встречается слово *in* и слово *put*, в то время как слово *input* всего 1 раз. Униграмная модель не знает ничего даже о самой примитивной связи между словами (для английской речи комбинация слов *in put* маловероятна).

Алгоритм 6. Clever Bruteforce with ambiguity resolving (bigram model)

Все то же самое, только теперь мы строим биграмную модель языка. Это значит, что мы считаем не частоты слов, а частоты всех пар слов которые идут подряд. Так, например предложение "*Kiev is the capital of Ukraine*" будет разбито на 5 биграмов: *Kiev is, is the, the capital, capital of, of Ukraine*. С таким подходом, модель хоть немного «понимает» какие слова могут стоять вместе, а какие нет. Теперь частота биграма *in put* в нашей модели нулевая.

Выводы

Алгоритм показывает неплохие результаты. Слабое место это словарь. Так как данные в твиттере, в основном, неформальные, имена людей, географические названия и т.п., словарь отсеивает много подходящих кандидатов. Поэтому одно из направлений развития алгоритма, это отказ от словаря. Вместо него можно использовать слова из трейн-сета. Второе слабое место это трейн-сет. Так как в ML алгоритмах все зависит от него, нужно иметь как можно больше релевантных данных. Здесь, как вариант, можно использовать трейн-сет из данных, полученых из того же твиттера.

Ссылки

Словарь с более чем 58 тысяч слов взят отсюда. В качестве трейн-сета был выбран файл с более чем миллионом слов, найденный на сайте Питера Норвига. Там еще

много всего интересного. Все это было реализовано на языке Clojure. Так что, кому интересно, github.