ICFP Contest 2017

DNIWE :: a

August 2017

1 Оценка рек

Обозначим через $A_G \in \{0,1\}^{N \times N}$ — матрицу смежности для текущего графа G с N рёбер (рек). Через $X \in \mathbb{R}^{N \times D}$ — набор признаков ребра (его вес, принадлежит ли оно текущему игроку, соединяет ли с шахтой, и т.п.).

Для простой оценки привлекательности реки e можно использовать функцию $f(e) = X_e^\intercal W$ (простое скалярное произведение двух векторов), где $W \in \mathbb{R}^D$ — настраиваемые параметры.

1.1 Учёт соседей

Учитывать признаки соседей в модели можно просто добавив в качестве признака сумму признаков соседей, что достигается простым перемножением матрицы смежности и текущего набора признаков $A_G X$. Умножив A_G на получившуюся матрицу, получим учём вторых соседей, и так далее (циклы, к сожалению, будут вносить свою лепту в оценку).

Эти добавочные признаки обозначим через P.

1.2 Нелинейности

Простая линейная модель имеет сильно меньше предсказательной силы. К счастью добавление нелинейности в нашу модель жутко простое, и это можно сделать даже не в одном месте.

1.3 Изменение признакового пространства

Добавив в качестве признаков попарные произведения уже существующих, мы перенесём модель в другое бОльшее пространство, где будет проще решать задачу.

В итоге признаки X превратятся в X', и все штуки выше применяются уже к X'.

1.4 Нелинейность в модели

У нас в модели есть конкатенация вида $\phi(x) = g(g(x))$, где $g(x)$ — линейная функция. В такие вещи очень просто добавить нелинейность, просто впухнув нелинейную функцию активации $\psi(x)$ (например, ReLU[1]), изменив таким образом функцию $g(x)$: $g'(x) = \psi(g(x))$.

Все выводы выше уже применяются к $g'(x)$.

[1] `https://en.wikipedia.org/wiki/Rectifier_(neural_networks)`

2 Оптимизация параметров

О-очень много статей и штук есть про оптимизацию подобных задач. Смысла гнатся за всем нет, поэтому далее опишу основные подходы, начиная с самых простых, заканчивая более муторными. Так уж получилось, что для простых подходов шаг оптимизации — целая игра, так что искать с их помощью параметры может быть долго. Для простоты будем считать, что наши параметры W ограничены, например, единичной сферой, т.е. $\|W\|_2 \leq 1$.

2.1 Случайное блуждание

Для каждого игрока генерируется свой набор параметров случайным образом (можно сэмплировать из нормального распределения, мат. ожидания которого берётся из сетки). Из двух игроков выбирается тот, кто выиграл в своей игре. Если оба выиграли или оба проиграли, то выбирается тот, кто набрал большее число очков (score).

Тут можно исхитрятся способом сэмплирования и т.п., чтобы ускорить перебор параметров, например использовать Rapidly-exploring random tree[2].

2.2 Методы отдомерной оптимизации

По окончанию игры мы получаем одно число — наше число очков. Можно применить методы одномерной оптимизации к нашей проблеме. Поскольку функция получения числа очков не является дифференциируемой (надо проиграть всю игру, ходы дискретны, и зависят от других игроков) по нашим параметрам, нам остаётся методы нулевого порядка. Хорошим методом является метод отжига[3], который похож на эволюционный алгоритм. Ещё можно попробовать метод колоний[4].

2.3 Обучение с подкреплением

Очень крутая вещь, хорошо применима к нашей задаче, и даже легко выписывается на бумаге. Одна загвоздка, что я не видел пока ни одной библиотеки на Python, которая реализовала нужный нам алгоритм REINFORCE[5].

Вообще говоря всё банально: есть признаки X', есть нелинейная функция преобразования $g'(x)$, можно добавить комбинации ещё всяческих вещей, которые дифференциируемы и прочее, чтобы на выходе иметь оценку привлекательности для каждого ребра, на её основе получить распределение вероятностей выбрать конкретное ребро (например, softmax), а потом использовать REINFORCE для обучения. Т.е. из распределения сэмплируется ход, на его основе считается награда (здесь это важно, надо считать награду за каждый ход), прогоняется назад, используя chain rule для градиентов конпозиций (сложной функции).

[2] `https://en.wikipedia.org/wiki/Rapidly-exploring_random_tree`
[3] `https://en.wikipedia.org/wiki/Simulated_annealing`
[4] `https://en.wikipedia.org/wiki/Particle_swarm_optimization`
[5] `https://www.cs.cmu.edu/afs/cs/project/jair/pub/volume4/kaelbling96a-html/node37.html`