

Национальный исследовательский университет ИТМО
Факультет информационных технологий и программирования
Прикладная математика и информатика

Методы оптимизации

Отчет по лабораторной работе №3

⟨Собрано 6 июня 2023 г.⟩

Работу выполнили:

Бактурин Савелий Филиппович М32331

Вереня Андрей Тарасович М32331

Сотников Максим Владимирович М32331

Преподаватель:

Ким Станислав Евгеньевич

Решение задачи нелинейной регрессии

Часто решая задачу создания регрессионной модели мы сталкиваемся с тем, что по жизни очень немногие рассматриваемые функции оказываются не представимы в виде обобщенной линейной зависимости или полиномиальной некоторой конечной степени k . Такая же ситуация часто случается и с некоторым набором данных, который нужно как-то обобщить. Именно в таких случаях к нам на помощь приходит более частный случай регрессионного анализа – *нелинейная регрессия*.

Идея построения нелинейной регрессии как и в случае с полиномиальной заключается в том, чтобы найти математическую функцию, которая максимально точно описывает зависимость между независимой переменной и зависимой от нее. Например, для построения нелинейной регрессии можно использовать функции типа полинома, логарифмической или экспоненциальной зависимости.

В целом весь процесс нахождения нелинейной регрессионной модели можно поделить на два этапа:

- ▷ Определить регрессионную модель $f(w, x)$, которая зависит от параметров $w = (w_1, \dots, w_W)$ и свободной переменной x .
- ▷ Решить задачу по нахождению минимума суммы квадратов регрессионных остатков:

$$S = \sum_{i=1}^m r_i^2, \quad r_i = y_i - f(w, x_i)$$

Однако, решая в лоб такую задачу, мы сталкиваемся с оптимизационной задачей нахождения параметров нелинейной регрессионной модели. Тут к нам и приходят на помощь различные методы нахождения, в том числе и рассматриваемые ниже: *Gauss-Newton* и *Powell Dog Leg*.

Gauss-Newton

Напомним, что мы решаем следующую задачу: дана нелинейная модель $f(w, x)$, где $w \in \mathbb{R}^m$, тогда сумма квадратов регрессионных остатков высчитывается как

$$S = \sum_{i=1}^{\text{sizeof } X} (f(w, x_i) - y_i)^2 \rightarrow \min$$

Итак, пусть $n = \text{sizeof } X$ и введем некоторые новые объекты для решения задачи, пусть $w^0 = (w_0^0, w_1^0, \dots, w_m^0)$ – начальное приближение, и

$$\mathbf{J} = \left(\frac{\partial f}{\partial w_j}(w^i, x_i) \right)_{n \times m} \quad - \text{Якобиан, или матрица первых производных}$$

$$\vec{f}_i = (f(w^i, x_i))_{n \times 1} \quad - \text{вектор значений функции } f$$

$$\delta_i = \text{const} \quad - \text{размера шага}$$

Тогда, формула i -й итерации рассматриваемого метода будет высчитываться как

$$w^{i+1} \leftarrow w^i - \underbrace{\delta_i \cdot (\mathbf{J}_i^T \mathbf{J}_i)^{-1}}_{\beta} \mathbf{J}_i^T (\vec{f}_i - y),$$

где β – это псевдообратная матрица к матрице \mathbf{J}_i , или решение некоторой задачи многомерной линейной регрессии, где мы ищем такой вектор β , что

$$\left\| \mathbf{J}_i \beta - (\vec{f}_i - y) \right\|^2 \rightarrow \min,$$

где y – вектор правильных/настоящих ответов нашей модели. Получается, для решения задачи, мы, так называемую, *невязку* пытаемся приблизить линейной комбинацией вектора из матрицы Якобиана так, что при следующем шаге итерации получить такой w^{i+1} , который бы сократил нам расстояние невязки. Причем, заметим, что на каждом шаге, задача будет новой, так как \mathbf{J}_i зависит от текущего приближения, чтобы решить задачу многомерной регрессии.

Заметим, что здесь, по алгоритму, мы видим достаточно очевидное ограничение: $m \geq n$, в ином случае для $\mathbf{J}_i^T \mathbf{J}_i$ не будет существовать обратной матрицы и, в следствии, решения к уравнению.

Исследования

Powell Dog Leg

Trust-region method — это метод решения оптимизационных задач, который основывается на вычислении региона, в котором квадратичная модель аппроксимирует целевую функцию. Сам этот метод представляет из себя смесь сразу двух алгоритмов, решающих задачу:

- ▷ Линейный поиск используется для определения направления поиска и дальнейшего нахождения оптимального шага вдоль выбранного вектора пути.
- ▷ Сам по себе trust-region используется для определения области вокруг текущей итерации, в которой модель достаточно аппроксимирует целевую функцию. Причем, стоит заметить, что для поиска следующего радиуса рассматриваемого региона также будет использоваться линейный поиск.

В общем случае Trust-region на каждой итерации решает следующую квадратичную задачу:

$$\min_{p \in \mathbb{R}^n} m_k(p) = f_k + p^T g_k + \frac{1}{2} p^T B_k p,$$

где $f_k = f(x_k)$, $g_k = \nabla f_k$, $B_k = \nabla^2 f_k$ и $\nabla_k > 0$ – изменяющийся радиус региона, причем всё это, при условии, что $|p| \leq \nabla_k$. Заметим, что в таком простейшем виде мы получаем безусловно почти бесполезный алгоритм: он чрезвычайно медленный из-за появления B_k – Гессиана функции. С другой стороны, если он положительно определен и $|B_k^{-1} \nabla f_k| \leq \nabla_k$, то решение легко определить: $p_k^B = -B_k^{-1} \nabla_k$. Но, опять же, высчитывать еще и обратную матрицу – дело долгое и медленное, поэтому, начиная отсюда и до конца все лабораторной работы, мы будем то и дело пытаться приближать наши значения к реальным/по настоящему посчитанным значениям Гессиан-функции.

Здесь мы рассмотрим один из методов оптимизации при аппроксимации квадратичной модели – *Powell Dog Leg*. Начнем, пожалуй, с определения радиуса рассматриваемого доверительного региона: в алгоритме dogleg обычно выбирают основываясь на сходстве функции m_k (та, что мы решаем изначально) и оригинальной

функции f на предыдущей итерации. Зададим ρ_k следующим образом:

$$\rho_k = \frac{f_k - f_k^*}{m_k(0) - m_k(p_k)},$$

где $f_k^* = f(x_k + p_k)$. А теперь посмотрим на то, как именно лучше поменять шаг: в том случае, если ρ_k меньше нуля, то это значит, что наша модель далека от функции и нужно обязательно уменьшить радиус; в том случае, изменение функции почти не изменилось и мы попали на границу региона, то есть смысл увеличить радиус; в ином другом случае – остается неизменным.

$$\Delta_{k+1} = \begin{cases} \frac{1}{4}\Delta_k, & \rho_k < \frac{1}{4} \\ \min(2\Delta_k, \Delta_{\max}), & \rho_k > \frac{3}{4} \wedge \|p_k\| = \Delta_k \\ \Delta_k, & \text{в ином другом случае} \end{cases}$$

Наконец, начинается самое интересное со стороны Powell Dog Leg. Итак, мы находимся на некоторой точки нашей модели, есть подсчитанный Δ -радиуса доверительного региона, и посмотрим на полный шаг $p^B = -B^{-1}g$. Если p^B лежит в окружности региона, то мы можем его взять и более закончить алгоритм. В ином случае, рассмотрим анти-градиент $-g$ и попробуем вдоль нее поискать минимум квадратичной модели, то есть решить

$$\min_{\|-\tau g\| \leq \Delta} m(-\tau g)$$

Для её решения мы можем взять некую новую точку без каких-либо ограничений в направлении анти-градиента и найти минимум модели

$$p^U = -\frac{g^T g}{g^T B g} g$$

Здесь снова две ситуации, где может находиться т. p^U :

- ▷ Если она находится вне рассматриваемой области, то мы можем взять точку на границе и шагнуть туда.
- ▷ Если же она находится в окружности, то построим отрезок $p^U p^B$ и начнем искать минимум вдоль этих двух линий $\left(\begin{array}{c} \text{текущая} \rightarrow p^U \text{ и } p^U \rightarrow p^B \end{array} \right)$.

Наконец, вдоль пути мы рассматриваем траекторию $\hat{p}(\tau)$

$$\hat{p}(\tau) = \begin{cases} \tau p^U, & 0 \leq \tau \leq 1 \\ p^U + (\tau - 1)(p^B - p^U), & 1 \leq \tau \leq 2 \end{cases}$$

Подытожим. Мы получили, на самом деле, в чем-то схожий на метод Гаусса-Ньютона алгоритм нахождения схождения, в частности, кстати, точка p^B – это то, куда бы шагнул метод Гаусса-Ньютона, но при этом, если эта точка удовлетворяет нашим потребностям, то мы действуем как Гаусс-Ньютон, в ином случае – чуть по другому. Причем под «немного другим» способом предполагается, на самом деле, хитрая комбинация Гаусса-Ньютона и градиентного спуска (так как при маленьком доверительном регионе мы пойдем по направлению, близкому градиентному спуску).

Исследования

BFGS

BFGS, или Алгоритм Бройдена - Флетчера - Гольдфарба - Шанно – это тоже оптимизационный итерационный алгоритм для нахождения локального экстремума для не представимых данных или функций в линейном/полиномиальном виде.

Один из известных квазиньютоновских методов (то есть, тех, которые основаны на получении информации о кривизне функции). Как и в случае с Powell Dog Leg, данный метод, в отличии от многих квазиньютоновских, использует аналог довольно медлительного постоянного переопределения Гесссиана функции. Но если предыдущий механизм никак не взаимодействовал с явным Гесссианом, то BFGS, наоборот, ускоряет работу на порядок: ибо он не явно каждый раз высчитывает матрицу, а лишь приближает к ней значения, при этом посчитав по честному Гесссиан лишь один раз.

Рассмотрим идею этого алгоритма. Обозначим за $x_i = \{x_i^0, x_i^1, \dots, x_i^{n-1}\}$ – координата в пространстве, где n – размерность соответствующего пространства. Пусть дана нам некоторая функция $f(x)$ и, как обычно, решаем задачу оптимизации нахождения $\arg\min_x f(x)$. Тогда, зададим некую начальную точку x_0 и $H_0 = B_0^{-1}$ – начальное приближение, где B_0^{-1} – обратный Гесссиан функции, который или может быть посчитан в точке x_0 , или выбран как \mathbf{I} – обратная матрица. Наконец, сам алгоритм:

- 0) Пусть k – текущий номер итерации алгоритма.
- 1) Находим точку, в направлении которой будем производить поиск, она определяется следующим образом

$$p_k = -H_k \times \nabla f_k,$$

где здесь и далее $f_k = f(x_k)$.

- 2) Вычисляем x_{k+1} через рекуррентное соотношение следующего вида:

$$x_{k+1} = x_k + \alpha \cdot p_k,$$

где α – коэффициент, удовлетворяющий условиям Вольфа, которые, напомним, выглядят вот так

$$\begin{aligned} f(x_k + \alpha \cdot p_k) &\leq f(x_k) + c_1 \cdot \alpha \cdot \nabla f_k^T p_k \\ \nabla f(x_k + \alpha \cdot p_k)^T p_k &\geq c_2 \cdot \nabla f_k^T p_k \end{aligned}$$

- 3) Теперь определим размер шага алгоритма после данной итерации и изменение градиента следующими соответствующими образами

$$\begin{aligned} s_k &= x_{k+1} - x_k \\ y_k &= \nabla f_{k+1} - \nabla f_k \end{aligned}$$

- 4) Наконец, обновим Гесссиан функции, зная, что $\lambda = \frac{1}{y_k^T s_k} \in \mathbb{R}$

$$H_{k+1} = (\mathbf{I} - \lambda s_k y_k^T) H_k (\mathbf{I} - \lambda y_k s_k^T) + \lambda s_k s_k^T$$

Исследования

L-BFGS

L-BFGS, или BFGS с ограниченной памятью – это оптимизационный алгоритм, который аппроксимирует оригинальный алгоритм BFGS с использованием заданного ограниченного объема памяти.

L-BFGS как и BFGS использует приближенную оценку Гесссиана, при этом в явном виде посчитав только один раз, а все остальные шаги лишь преобразовывая. Проблема: BFGS хранит всегда $n \times n$ приближений к обратному Гесссиану. Решение: хранить несколько векторов, которые неявно представляют приближение, представляющие из себя историю последних m обновлений положения x и градиента $\nabla f(x)$. При этом, m обычно выбирается небольшим ($m < 10$).

Рассмотрим идею этого алгоритма. Во многом она будет совпадать с предыдущим, поэтому пропустим обозначения и перейдем сразу алгоритму.

- 0) Пусть k – текущий номер итерации алгоритма, здесь и далее будем иметь ввиду $g_k = \nabla f(x_k)$.

- 1) Также как и в BFGS находим точку, в направлении которой будем производить поиск:

$$p_k = -H_k \times \nabla f_k,$$

здесь и далее $f_k = f(x_k)$.

- 2) Пусть мы сохранили m обновлений вида:

$$s_k = x_{k+1} - x_k$$

$$y_k = g_{k+1} - g_k$$

Заметим, что при последующих итерациях алгоритма $k \geq m$ произведение из первого шага можно получить выполнив последовательность скалярных произведений и суммирования векторов, включающую ∇f_k и пары $\{s_i, y_i\}$. После вычисления новой итерации самая старая пара векторов в наборе пар $\{s_i, y_i\}$ заменяется новой парой, который получается из данного шага.

- 3) Наконец, пожалуй, самая идейная часть – обновление Гесссиана. На итерации k у нас определен x_k и пары $\{s_i, y_i\} \forall i \in [k-m, k-m+1, \dots, k-1]$. Выберем некоторое начальное Гесссианское приближение H_k^0 (в отличие от стандартной итерации BFGS, это начальное приближение может меняться от итерации к итерации) и путем повторного применения формулы, заданной изначально в BFGS, получаем

$$\begin{aligned} H_k = & (V_{k-1}^T \cdot \dots \cdot V_{k-1}^T) H_k^0 (V_{k-m} \cdot \dots \cdot V_{k-1}) \\ & + \rho_{k-m} (V_{k-1}^T \cdot \dots \cdot V_{k-m+1}^T) s_{k-m} s_{k-m}^T (V_{k-m+1} \cdot \dots \cdot V_{k-1}) \\ & + \rho_{k-m+1} (V_{k-1}^T \cdot \dots \cdot V_{k-m+2}^T) s_{k-m+1} s_{k-m+1}^T (V_{k-m+2} \cdot \dots \cdot V_{k-1}) \\ & + \dots \\ & + \rho_{k-1} s_{k-1} s_{k-1}^T, \end{aligned}$$

где $\rho_k = \frac{1}{y_k^T s_k}$, $V_k = \mathbf{I} - \rho_k y_k s_k^T$. Из всего вышеописанного мы можем провести произведение $H_k \times \nabla f_k$ более эффективно следующим образом

```

 $q \leftarrow \nabla f_k;$ 
 $\forall i = k-1, k-2, \dots, k-m \text{ do}$ 
     $\alpha_i \leftarrow \rho_i s_i^T q;$ 
     $q \leftarrow q - \alpha_i y_i;$ 
 $\text{end}$ 
 $r \leftarrow H_k^0 q;$ 
 $\forall i = k-m, k-m+1, \dots, k-1 \text{ do}$ 
     $\beta \leftarrow \rho_i y_i^T r;$ 
     $r \leftarrow r + s_i(\alpha_i - \beta);$ 
 $\text{end}$ 
 $\text{return } r \equiv H_k \times \nabla f_k;$ 

```

Получение нового на данной итерации H_k^0 мы также сильно ускорим, лишь приблизив наши значения, используя формулу $H_k^0 = \gamma_k \mathbf{I}$, где

$$\gamma_k = \frac{s_{k-1}^T y_{k-1}}{y_{k-1}^T y_{k-1}}$$

4) Все последующие шаги аналогичны с оригинальным BFGS.

Итак, напомним идейный псевдокод алгоритма L-BFGS.

```

1  function is_convergence(f:  $\mathbb{R}^n \rightarrow \mathbb{R}$ ):
2      /*implementation defined*/
3
4  function Wolfe_coefficient(f:  $\mathbb{R}^n \rightarrow \mathbb{R}$ ,  $p_k$ ):
5      /*implementation defined*/
6
7  function get_prod(f:  $\mathbb{R}^n \rightarrow \mathbb{R}$ ):
8       $q \leftarrow \nabla f_k;$ 
9       $\forall i = k-1, k-2, \dots, k-m \text{ do}$ 
10          $\alpha_i \leftarrow \rho_i s_i^T q;$ 
11          $q \leftarrow q - \alpha_i y_i;$ 
12     end
13      $r \leftarrow H_k^0 q;$ 
14      $\forall i = k-m, k-m+1, \dots, k-1 \text{ do}$ 
15          $\beta \leftarrow \rho_i y_i^T r;$ 
16          $r \leftarrow r + s_i(\alpha_i - \beta);$ 
17     end
18     return  $r \equiv H_k \times \nabla f_k;$ 
19
20 function LBFGS():
21      $x_0 \leftarrow \text{INIT};$ 
22      $m \leftarrow i \in [5, 10];$ 
23      $q \leftarrow []$ 
24     while !is_convergence(f) do
25          $H_k^0 \leftarrow \left( \frac{s_{k-1}^T y_{k-1}}{y_{k-1}^T y_{k-1}} \right) \mathbf{I};$ 
26          $p_k \leftarrow \text{get\_prod}(f);$ 
27          $\alpha_k \leftarrow \text{Wolfe\_coefficient}(f, p_k);$ 
28          $x_{k+1} \leftarrow x_k + \alpha_k p_k$ 
29         if  $k > m$  then
30              $q.\text{remove}(\{s_{k-m}, y_{k-m}\});$ 

```

```
31       $s_k \leftarrow x_{k+1} - x_k$  ;  
32       $y_k \leftarrow \nabla f_{k+1} - \nabla f_k$  ;  
33      q.append({ $x_k, y_k$ }) ;  
34
```

Исследования