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

Методы оптимизации

Отчет по лабораторной работе №2

⟨Собрано 13 мая 2023 г.⟩

Работу выполнили:
Бактурин Савелий Филиппович M32331
Вереня Андрей Тарасович M32331
Сотников Максим Владимирович M32331

Преподаватель:
TBA

Стохастический градиентный спуск

Исследование с разными размерами батча

Реализуйте стохастический градиентный спуск для решения линейной регрессии. Исследуйте сходимость с разным размером батча (1 – SGD, 2, $\ldots$, $n-1$ – Minibatch GD, n – GD из предыдущей работы).

Стохастический градиентный спуск

Стохастический градиентный спуск — модификация к основному методу итерационного поиска минимума через антиградиент дифференцируемой функции в рассматриваемой плоскости $\mathbb{R}^n$. Идея: пусть есть множество M – какой-то полный набор данных вычисленных оценок при спуске к минимуму, тогда в рассматриваемой версии мы будем брать случайное значение из выбранного $M' \subset M$. Как правило, такой подход преуменьшает вычислительные ресурсы, в особенности, следует помнить, что `float` считается крайне медленно, и ускоряет итерацию по количествам эпохам, но при этом мы теряем точность сходимости.

Пусть $x_i = \{x_i^0, x_i^1, \ldots, x_i^{n-1}\}$ – координата в $\mathbb{R}^n$ и задана функция $f(x_i) : \mathbb{R}^n \to \mathbb{R}$. Мы хотим нашу задачу свести к исследованию на некоторых специальных образцах заданной функции, для каждой точки из которых мы будем минимизировать ошибку для дальнейшего нахождения приближенного минимума рассматриваемой функции $f(x_i)$. Мы хотим найти линейную регрессию, представляющая из себя полином 1-ой степени от n переменных. Для начала мы найдем функцию ошибки S по следующей формуле:

$$S(f) = (X'^{\mathrm{T}} \times X')^{-1} \times X'^{\mathrm{T}} \times Y \times \vec{x},$$

где Y – матрица значений при множестве X (определение), X' – это матрица X, но в 1-ой колонке забитый единицами. По другому мы можем записать данную формулу следующим образом:

$$S(f) = \sum_{i=1}^{N} (y_i - x_i \cdot w_i)^2,$$

где $y_i \in D(f(x_i))$ (образ функции), $w_i \in W$ – сгенерированные веса, коэффициенты при линейной функции.

Рассмотрим идею алгоритма. При итерации, пока мы не превысили максимальное количество шагов или не сведем нашу функцию потери до некоторого ε, мы будем обобщать экспериментальные данные в виде случайных точек в некоторую многомерную линию. На каждом шаге мы будем изменять функцию потерь от измененной w.

Напишем идейный псевдокод. Скажем, что $x_i = \{x_i^0,\ x_i^1,\ \ldots, x_i^{n-1}\}$ – координата в n-мерном пространстве $\mathbb{R}^n$, $y_i = \{y_i^0,\ y_i^1,\ \ldots, y_i^{n-1}\}$ – образ функции $f(x_i)$.

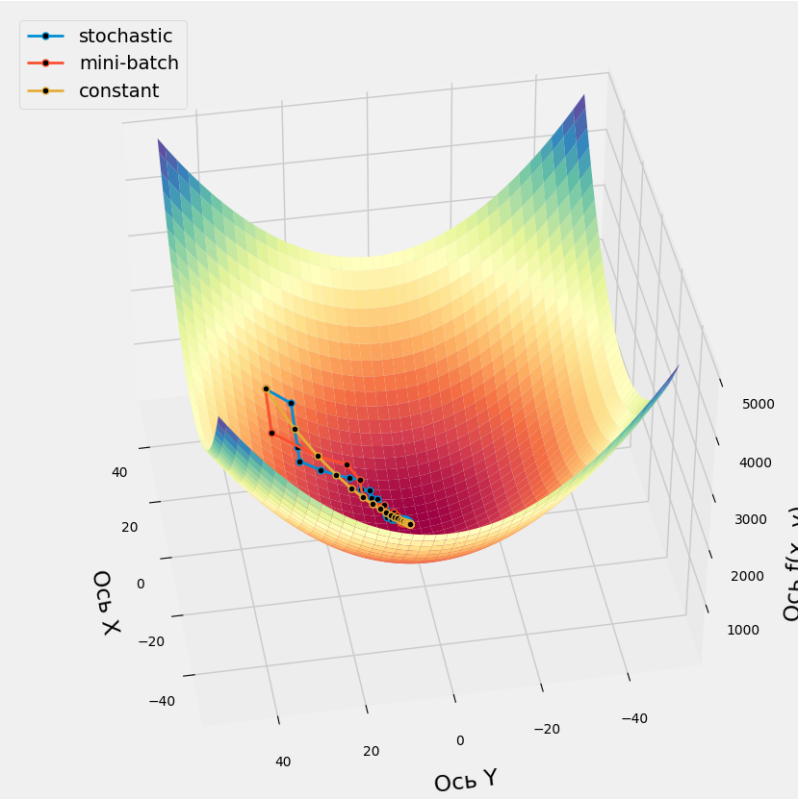
```
function S(x, y, w):
    return Σ_{i=1}^{N} (y_i − x_i · w_i)^2

function stochastic_descent(x, y):
    w ← [w_i ∈ ℝ] * n
```

```
    prev ← INIT, предыдущее значение функции потери
    next ← S(x, y, w), текущее значение функции потери
    α ← const
    while |prev − next| > ε:
        prev ← next
        i ← x ∈ [0, |Y|]
        T ← [0] * n
        ∀j ∈ |w| do
            T_j ← (y_i − x_i × w) · x_i^j
        w ← w + α · T
        next ← S(x, y, w)
    return w
```

Пример с SGD, Minibatch, GD. Для понимания контекста, давайте увидим разницу между рассматриваемым градиентным спуском, Minibatch и общим, с которым мы уже давно знакомы. Возьмем в качестве функции $f(x, y) = x^2 + y^2$, тогда его градиентом будет $\dfrac{\partial f}{\partial \vec{x}} = 2x + 2y$, также для GD мы выставим следующие параметры:

- ▷ `learning_rate` $= 0.1$
- ▷ $\varepsilon = 0.00001$
- ▷ `max_epoch` $= 1000$ – одно из условий остановки спуска, вторым – проверка, что мы сошлись.

Запустим и посмотрим на график всех трех джентльменов.



SGD vs. Minibatch vs. GD

Полученные данные, после прохода алгоритмов, выглядят так:

▷ Для стохастического нашлась $f(0.002658, 0.00836) \approx 0.000008$.

▷ Для Minibatch – $f(0.002658, 0.000219) \approx 0.000007$.

▷ Для градиентного спуска (constant) – $f(0.001701, 0.002042) \approx 0.000007$.

Точками отмечены шаги алгоритмов. Заметим, что в отличие GD, стохастический спуск уже с первого шага начинает идти не туда и, как можно приглядеться в минимуме функции, синяя полоса пробирается то левее, то правее GD небольшими шагами. Minibatch также начинает свое движение не совсем по траектории GD, но в отличие от своего предшественника, быстрее начинает сходится к настоящей траектории.

Minibatch градиентный спуск

Модификация *Minibatch* обобщает вариант стохастического градиентного спуска, тем, что во время итерации мы будем брать не одну случайную точку из посчитанных на предыдущем шаге и изменять функцию потери как бы относительно её, а теперь возьмем выборку $M' \subset M$, причем, обязательно, чтобы $|M'| > 1$ и $|M'| < |M|$.

Тогда псевдокод от предыдущего рассмотренного варианта почти ничем не отличается. Скажем также, что $x_i = \{x_i^0,\ x_i^1,\ \ldots, x_i^{n-1}\}$ – координата в n-мерном пространстве $\mathbb{R}^n$, $y_i = \{y_i^0,\ y_i^1,\ \ldots, y_i^{n-1}\}$ – образ функции $f(x_i)$; значение m – выступает в роли мощности подмножества M'.

```
function S(x, y, w):
    return Σ_{i=1}^{N} (y_i − x_i · w_i)^2

function stochastic_descent(x, y, m):
    w ← [w_i ∈ ℝ] * n
    prev ← INIT, предыдущее значение функции потери
    next ← S(x, y, w), текущее значение функции потери
    α ← const
    while |prev − next| > ε:
        prev ← next
        ∀i ∈ [0, m] do
            T ← [0] * n
            ∀j ∈ |w| do
                T_j ← (y_i − x_i × w) · x_i^j
            w ← w + α · T
        next ← S(x, y, w)
    return w

```

Градиентный спуск

Наконец, самый общий случай и являющийся самым быстроходным среди двух рассмотренных ранее, благодаря тому, что мы учитываем все точки $M' \equiv M$. Данный

метод работает крайне медленно, но является одним из самых быстрых в сходимости к приближенной точке минимума.

Пусть m – есть мощность множества M, тогда идейным псевдокодом-решением задачи будет являться тот же код, что и для предыдущего варианта, то есть

```
function S(x, y, w):
    return Σ_{i=1}^{N} (y_i − x_i · w_i)^2

function stochastic_descent(x, y, m):
    w ← [w_i ∈ ℝ] * n
    prev ← INIT, предыдущее значение функции потери
    next ← S(x, y, w), текущее значение функции потери
    α ← const
    while |prev − next| > ε:
        prev ← next
        ∀i ∈ [0, m] do
            T ← [0] * n
            ∀j ∈ |w| do
                T_j ← (y_i − x_i × w) · x_i^j
        w ← w + α · T
    next ← S(x, y, w)
    return w
```

Исследование

Learning rate scheduling

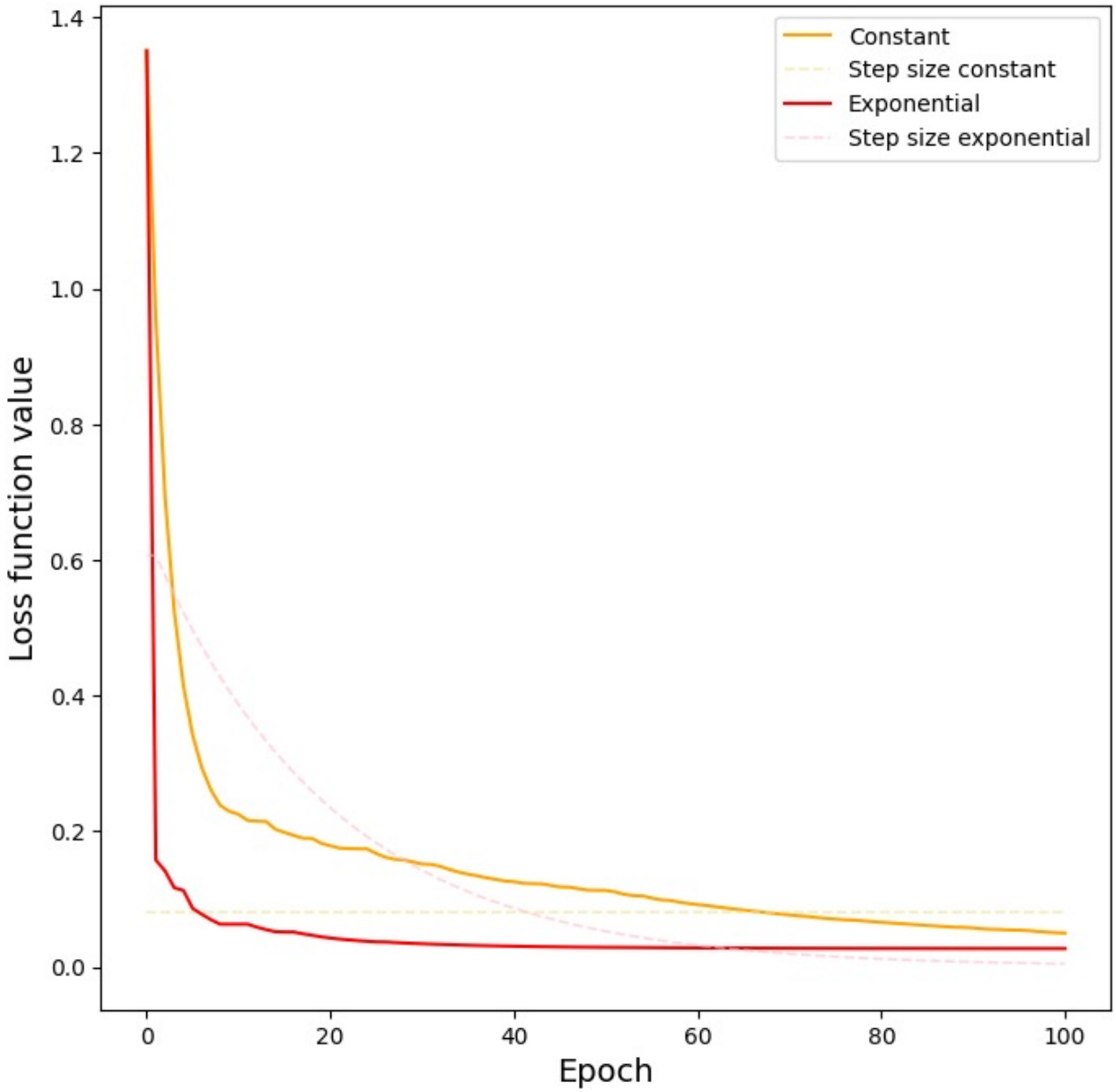
Задачей мы поставим подбор функции изменения шага, чтобы улучшить сходимость из предыдущего пункта. Тогда, мы использовали самый простой способ – *константный*, однако у него есть недостаток: иногда шаг в `const_learning_rate` может «перепрыгнуть» через минимум и/или начать «прыгать» через него бесконечно много раз, в таком случае мы хотим, чтобы шаг был уменьшен. Другой случай: когда такой шаг может привести к долгому ожиданию, пока алгоритм не дойдет до минимума. Вот тут и возникает задача подбора такой функции изменения шага, чтобы алгоритм за какое-то конечное k эпох добрался до минимума быстрее. В качестве такой мы возьмем *экспоненциальную функцию*. Экспоненциальная функция, в общем случае, для изменения сходимости выглядит так:

$$\texttt{learning\_rate} = \texttt{start\_learning\_rate} \cdot e^{-k \cdot \texttt{epoch}},$$

где `learning_rate` – текущий размер шага, `start_learning_rate` – стартовая длина, k – некий параметр, который может зависеть от размера выбранного батча, и, наконец, `epoch` – текущий номер эпохи.

Пример с $f(x) = 2x + 0$. Для линейной регрессии рассмотрим в качестве функции $f(x) = 2x + 0$, где $a = 2$ и $b = 0$. Сравним разницу между константным и экспоненциальной функциями потерь, где в качестве значения $k \leftarrow 0.05$ и `epoch` будем считать

следующим образом: (`current_epoch` + 10). Рассмотрим следующую зависимость: `значение_функции_потерь(количество_эпох)`.



Constant lr vs. Exponential lr

Как можно заметить, константный метод по очевидным причинам хуже сходится (параллельная линия оси ординат), экспоненциальная же понемногу (на самом деле, по многу) начинает быстрее сходится из-за присутствия «минуса» в степени и, соответственно, меньшим lr.

SGD и разные функции потерь. Как и в предыдущей задаче, мы рассмотрим линейную регрессию, но в качестве исследования мы возьмем других не менее из-

вестных представителей функций потерь:

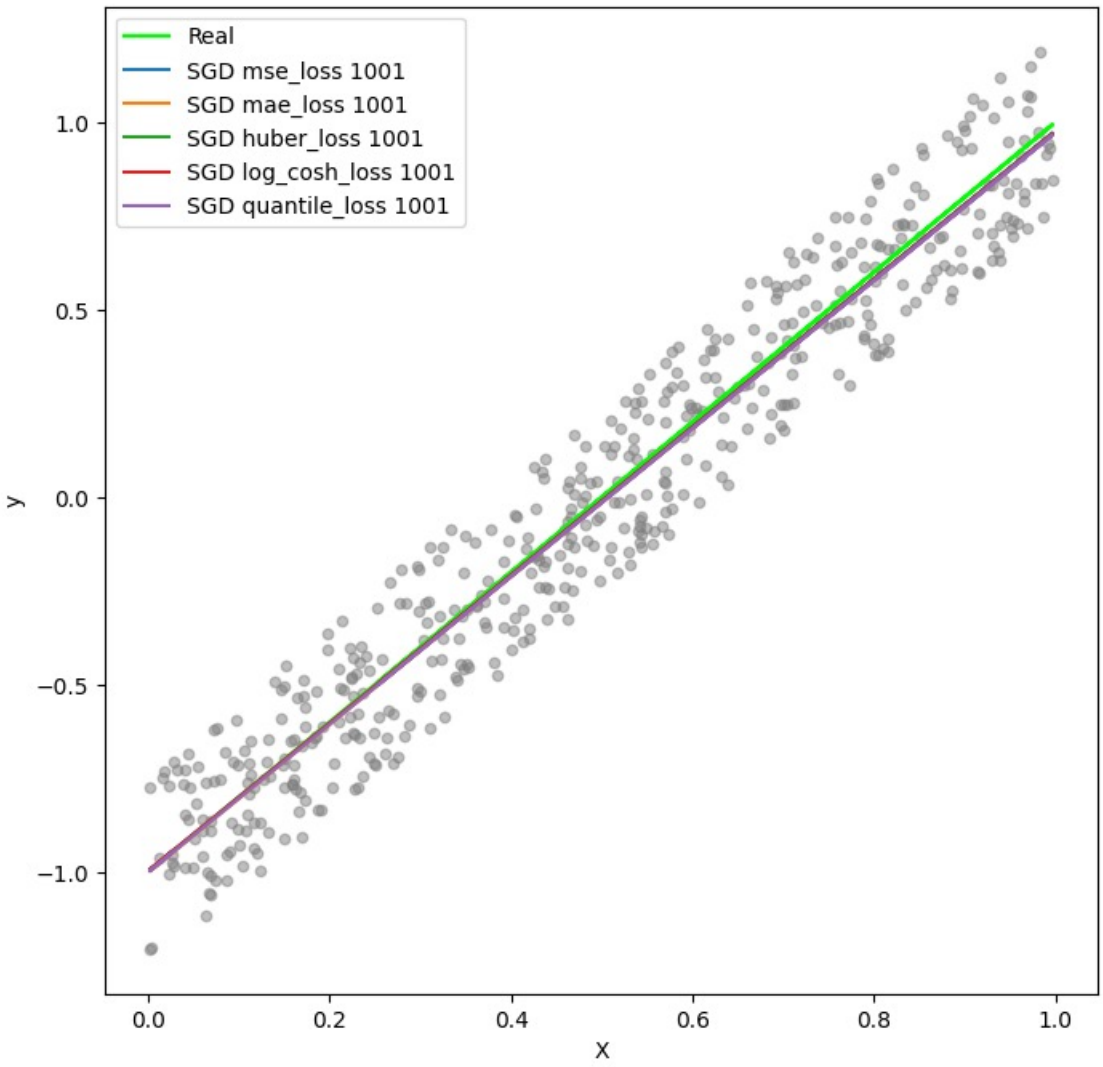
$$\text{MAE} = \frac{\sum\limits_{i=1}^{n} |y_i - x_i|}{n}$$

$$L_\delta(a) = \begin{cases} \frac{1}{2} \cdot a^2, \ |a| \leqslant \delta, \\ \delta \cdot \left(|a| - \frac{1}{2} \cdot \delta \right) \end{cases}, \text{ где } a = y - f(x)$$

$$\text{logcosh} = \log \left(\frac{e^{y_{\text{pred}} - y_{\text{real}}} + e^{-y_{\text{pred}} + y_{\text{real}}}}{2} \right)$$

$$\text{quantile} = \begin{cases} \alpha \cdot (y_{\text{real} - y_{\text{pred}}}), \ y_{\text{real} - y_{\text{pred}}} \geqslant 0 \\ (\alpha - 1) \cdot (y_{\text{real} - y_{\text{pred}}}) \end{cases}$$

На сей раз, мы рассмотрим линейную функцию $f(x) = 2x - 1$. Для каждого из функций потерь построим pred-функцию и сравним их эффективность в плане восстановления до исходной прямой.



Real vs. mae vs. mse vs. huber vs. logcosh vs. quantile

Для нее мы задали следующие настройки алгоритма:

▷ Количество генерируемых весов – 5000.

▷ $\varepsilon \leftarrow 0.0085$

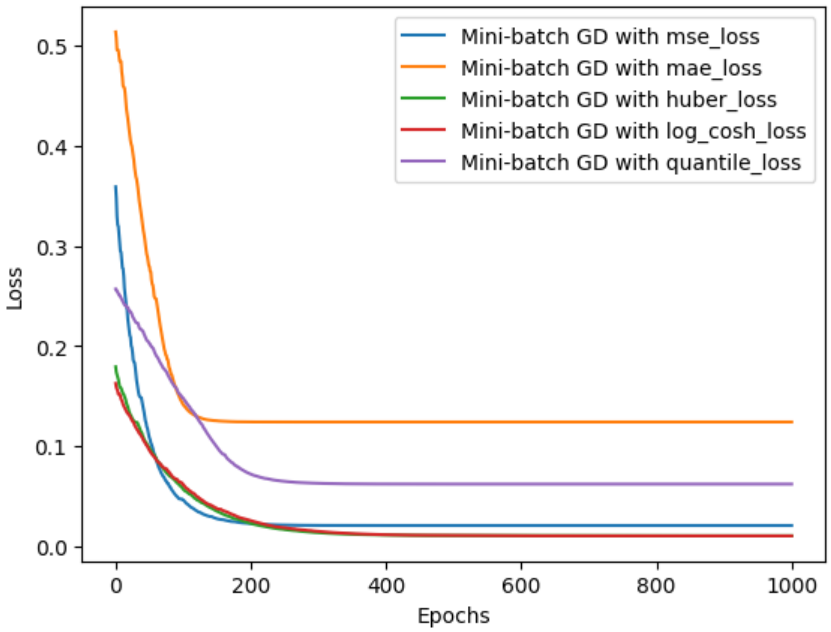
▷ `learning_rate` $\leftarrow 0.1$

▷ Максимальное число до сходимости – 1000.

Заметим, что почти все методы отработали почти одинаково и совпали с настоящей прямой. Полные результаты коэффициентов уравнений прямой такие:

$$\begin{aligned} \text{MSE} &\to y = 2.002 \cdot x - 1.000 \\ \text{MAE} &\to y = 2.005 \cdot x - 1.002 \\ \text{HUBER} &\to y = 1.999 \cdot x - 0.998 \\ \text{LOGCOSH} &\to y = 1.999 \cdot x - 0.998 \\ \text{QUANTILE} &\to y = 2.006 \cdot x - 1.002 \end{aligned}$$

Minibatch и разные функции потерь. Как и в прошлом примере, мы возьмем функцию $f(x) = 2x - 1$ и те же разные местные представители фауны из предыдущего пункта. Однако, теперь мы возьмем Minibatch GD и составим график на сей раз `значение_функции_потери`(`эпохи`). Настройки для алгоритма такие же, как и в прошлом пункте. Запустим и проанализируем:



Minibatch vs. loss functions

И мы получаем две категории победы у разных функций:

▷ По быстроте (то есть, по количеству эпох) сходимости (то есть, когда мы получаем параллельную прямую над осью ординат) – MAE, при этом эта функция проигрывает следующей категории.

▷ По аргументу (то есть, по значению функции потерь) – получаем вровень идущих $L_\delta(a)$ и logcosh, приближающееся к нулю.

Полные результаты значений функций потерь:

$$\text{MSE} \to \begin{cases} \text{REAL} = 0.02061322422787911 \\ \text{MSE} = 0.020611644279003895 \\ \text{DIFF} = -1.579948875216064e-06 \end{cases}$$

$$\text{MAE} \to \begin{cases} \text{REAL} = 0.12404634033734865 \\ \text{MAE} = 0.12403644707794269 \\ \text{DIFF} = -9.89325940596586e-06 \end{cases}$$

$$\text{HUBER} \to \begin{cases} \text{REAL} = 0.010306612113939555 \\ \text{HUBER} = 0.010306151224572697 \\ \text{DIFF} = -4.6088936685867443e-07 \end{cases}$$

$$\text{LOGCOSH} \to \begin{cases} \text{REAL} = 0.010242649302131148 \\ \text{LOGCOSH} = 0.01024226833627978 \\ \text{DIFF} = -3.809658513671127e-07 \end{cases}$$

$$\text{QUANTILE} \to \begin{cases} \text{REAL} = 0.06202317016867433 \\ \text{QUANTILE} = 0.062018214022529945 \\ \text{DIFF} = -4.956146144381723e-06 \end{cases}$$

Исследование различных модификаций

Исследуйте модификации градиентного спуска (Nesterov, Momentum, AdaGrad, RMSProp, Adam).

Momentum

Пусть нам дана функция $f(x_i)$, где $x_i = \{x_i^1,\ x_i^2,\ \ldots,\ x_i^n\}$ – координата точки x_i в n-мерном пространстве – и пусть у данной f есть множество локальных точек минимума, образующийся «впадиной» функции, и «горы» – максимумов. Рассмотрим некоторый *объект* $\mathfrak{O}$, который обладает свойством «скольжения» по поверхности нашей функции f, определяемый следующим образом:

a) если он «сходит» с горы, то он не останавливается при «схождении» с локального максимума;

b) если он «перескакивает» локальный минимум, то он либо остановится на идеально гладкой части поверхности функции f, либо он «сойдет» к локальному минимуму.

Идея модификации градиентного спуска *Momentum* в создании алгоритма, обладающий свойством $\mathfrak{O}$ в имении свойства импульса.

Рассмотрим последовательность $\{u_i\}$, которую мы назовем «скоростью», скажем, что $\{g_i\} \equiv \{-\nabla f(x_i)\}$, тогда определим скорость следующим образом:

$$v_{i+1} = \lambda \cdot v_i + (1 - \lambda) \cdot g_i,\ \lambda \in (0, 1)$$

и зададим изменение на $i+1$-ой итерации наших весов точек:

$$w_{i+1} = w_i - \alpha \cdot v_{i+1},\ \alpha = \texttt{const}$$

Nesterov

Модификация *Nesterov* по своей сути почти ничем не отличается ранее рассмотренного Momentum: единственное, что отделяет братьев по крови, так это то, что Nesterov считает градиент не в текущей точке, на которой мы стоим, а той, куда мы бы могли пойти, следуя импульсу. Тогда, наши основные формулы немного меняются, для скорости:

$$v_{i+1} = \lambda \cdot v_i + (1-\lambda) \cdot g(w_i - \alpha \cdot \lambda \cdot v_i),\ \alpha = \texttt{const}$$

И для весов:

$$w_{i+1} = w_i - \alpha \cdot v_{i+1}$$

AdaGrad

Рассмотрим некий `start_learning_rate` и стандартный метод стохастического спуска. Как мы уже видели, одним из недостатков такого метода является появления слишком больших или, наоборот, маленьких компонент относительно друг друга. Для исправления мы введём матрицу Ω, которую определим следующим образом:

$$\Omega = \sum_{j=1}^{i} g_j \times g_j^{\mathrm{T}},$$

где g_j – $\nabla f(x_j)$. Теперь рассмотрим основную формулу и поделим часть, где идет «шаг» алгоритма, на $\Omega_{i,i}$:

$$w_{i+1} = w_i - \frac{\alpha \cdot g_j}{\sqrt{\Omega_{i,i}}}$$

Теперь же, там, где мы потенциально делаем большие шаги, мы делим на большое $\Omega_{i,i}$ и таким образом уменьшаем количество больших скачков. Тут мы получаем глобальную проблема в виде потенциально слишком высокой скорости уменьшения шага.

RMSProp

Модификация *RMSProp* решает возникшую проблему с AdaGrad, в качестве решения было предложено усреднять значения. Пусть $(g_i)^2$ – покомпонентное возведение в квадрат, $\eth$ – некое число порядка $[10^{-10},\ 10^{-7}]$, дабы по формуле не было деления на ноль. Тогда, в качестве скорости возьмем:

$$s_{i+1} = \lambda \cdot s_i + (1-\lambda) \cdot (g_i)^2,\ \lambda \in (0,1)$$

Наконец, изменение весов:

$$w_{i+1} = w_i - \alpha \cdot \frac{g_i}{\sqrt{s_{i+1} + \eth}},\ \alpha = \texttt{const}$$

Adam

Начинаем объединение веселья предыдущих. В отличие от всех его предшественником модификация *Adam* является наиболее распространенным и используемым.

$$
\begin{aligned}
v_{i+1} &= \lambda_1 \cdot v_i + (1 - \lambda_1) \cdot g_i \\
s_{i+1} &= \lambda_2 \cdot s_i + (1 - \lambda_2) \cdot (g_i)^2 \\
\hat{v}_{i+1} &= \frac{v_{i+1}}{1 - \lambda_1^{i+1}} \\
\hat{s}_{i+1} &= \frac{s_{i+1}}{1 - \lambda_2^{i+1}}
\end{aligned}
$$

Наконец, изменение весов будем рассчитывать по следующей формуле:

$$
w_{i+1} = w_i - \lambda \cdot \frac{\hat{v}_{i+1}}{\sqrt{\hat{s}_{i+1} + \eth}},
$$

где $\lambda_1 = 0.9$, $\lambda_2 = 0.999$ и $\eth = 10^{-8}$.

Сравнение модификаций

Сходимость

Траектории

Полиномиальная регрессия

1. Реализуйте полиномиальную регрессию. Постройте графики восстановленной регрессии для полиномов разной степени.

2. Модифицируйте полиномиальную регрессию добавлением регуляризации в модель (L1, L2, Elastic регуляризации).

3. Исследуйте влияние регуляризации на восстановление регрессии.

Введение

Представим себе некую функцию $f : \mathbb{R} \to \mathbb{R}$ причем такую, что она не зависит от каких-то побочных факторов (случайные значения и тому подобное), и при этом получаемые экспериментальные данные были как бы «разбросаны» относительно некой обобщающей кривой. Нашей задачей состоит в понимании, что это за кривая – по сути, это множество пар $M = \{\langle x, y\rangle : x \in \mathbb{R},\ y \in \mathrm{E}(f(x))\}$, которые при переходе от заданного $x' \to x''$ по некоторому δ, подает предугадываемое значение y'' на основе полученных ранее начальных экспериментальных данных. Говоря простыми словами, с помощью данной прямой мы хотим получать прогноз.

Рассмотрим идею *полиномиальной регрессии*. Пусть даны $x_i \in X$ и $y_i \in Y$. Тогда уравнением полинома будет иметь следующий вид в нашей задаче:

$$y_i = \sum_{j=0}^{k} a_j \cdot x_i^j,$$

где $\{a\}$ – это коэффициенты полинома, где a_0, их мы ищем ровно тем же способом, что и в первой задаче – по методу наименьших квадратов. Наконец, по аналогии с задачей градиентного спуска, мы получаем следующее выражение:

$$L = \omega \cdot \left[\!\!\left[\sum_{i=1}^{n} (\mathbf{y}_i - y_i)^2 \right]\!\!\right] \to \min,$$

где $\{\mathbf{y}\}$ – это значения полинома в точках $\{x\}$, тогда наше выражение преобразуется:

$$L = \omega \cdot \left[\!\!\left[\sum_{i=1}^{n} \left(\sum_{j=0}^{k} \left(a_j \cdot x_i^j - y_i\right)^2 \right)^2 \right]\!\!\right] \to \min,$$

где $\omega = \dfrac{1}{n}$, для того чтобы усреднить квадратичные ошибки на всех обучающих примерах. Это позволяет получить более сбалансированную и интерпретируемую метрику ошибки, не зависящую от размера обучающей выборки. Для получения градиента исследуемой функции мы воспользуемся производной от L по некоторому коэффициенту $a_{\mathbf{i}}$, тогда

$$\frac{\partial L}{\partial a_{\mathbf{i}}} = 2 \cdot \omega \cdot \sum_{i=1}^{n} \left(\sum_{j=0}^{k} \left(a_j \cdot x_i^j - y_i\right) \cdot x_{\mathbf{i}}^i \right)$$

Исследование различных модификация

Как и в задаче с градиентным спуском нам бы хотелось получать более реальные «гладкие» оценки предсказывания значения, вместо имения «гор» и «впадин» кривой. Эта проблема может быть решена с помощью модификаций к методу, которые, по своей сути, представляют из себя отдельные слагаемые.

L1

Идея *L1 регуляризации*. Давайте с каждой новой итерации мы будем неким способом «наказывать» нашу модель за высокие веса, чтобы в будущем увеличить обобщающую способность. Эта регуляризация основана на добавлении еще одного слагаемого в основную сумму, равного абсолютному значению коэффициентов полинома.

$$L = \omega \cdot \sum_{i=1}^{n} \left(\sum_{j=0}^{k} \left(a_j \cdot x_i^j - y_i \right)^2 \right) + \lambda \cdot \sum_{i=0}^{k} \left(|a_i| \right),$$

где λ – есть общий коэффициент для регуляризации влияния штрафов на модель. Градиентом для модели будет выступать тогда такая формула:

$$\frac{\partial L}{\partial a_{\mathbf{i}}} = 2 \cdot \omega \cdot \sum_{i=1}^{n} \left(\sum_{j=0}^{k} \left(a_j \cdot x_i^j - y_i \right) \cdot x_{\mathbf{i}}^i \right) + \lambda$$

L2

Идея *L2 регуляризации*. Данная модификация является небольшим дополнением к предыдущему в виде большей гладкости производной и возможностью лучшей работой с градиентным спуском. Проблема L1 заключалась в том, что в некоторых местах, где производная не может быть установлена, имеет резкие «горы», «впадины» и вообще разрывы. Тогда, на помощью приходит немного иное штрафное значение функции – вместо суммы значений коэффициентов полинома мы возьмем квадраты. Отличает этот подход от L1 тем, что теперь мы штрафуем больше, но при этом никогда не зануляемся, как это было с предыдущей модификацией.

$$L = \omega \cdot \sum_{i=1}^{n} \left(\sum_{j=0}^{k} \left(a_j \cdot x_i^j - y_i \right)^2 \right) + \lambda \cdot \sum_{i=0}^{k} \left(a_i^2 \right),$$

где λ – есть общий коэффициент для регуляризации влияния штрафов на модель. Градиентом для модели будет выступать тогда такая формула:

$$\frac{\partial L}{\partial a_{\mathbf{i}}} = 2 \cdot \omega \cdot \sum_{i=1}^{n} \left(\sum_{j=0}^{k} \left(a_j \cdot x_i^j - y_i \right) \cdot x_{\mathbf{i}}^i \right) + \lambda \cdot a_{\mathbf{i}}$$

Elastic регуляризация

Идея *Elastic регуляризация*. Объединение двух предыдущих идей, как и полезных особенностей, свойств обоих методов, при этом, корреляция переменных такая, что

данная регуляризация не обнуляет некоторые из них, как в случае с L1.

$$L = \omega \cdot \sum_{i=1}^{n} \left(\sum_{j=0}^{k} \left(a_j \cdot x_i^j - y_i \right)^2 \right)^2 + \lambda_1 \cdot \sum_{i=0}^{k} \left(a_i^2 \right) + \lambda_2 \cdot \sum_{i=0}^{k} \left(a_i^2 \right),$$

где λ_1 и λ_2 – есть коэффициенты для регуляризации влияния штрафов L1 и L2 на модель соответственно. Градиентом для модели будет выступать тогда такая формула:

$$\frac{\partial L}{\partial a_{\mathbf{i}}} = 2 \cdot \omega \cdot \sum_{i=1}^{n} \left(\sum_{j=0}^{k} \left(a_j \cdot x_i^j - y_i \right) \cdot x_{\mathbf{i}}^i \right) + \lambda_1 + \lambda_2 \cdot a_{\mathbf{i}}$$