

Практическое задание 1: Методы безусловной оптимизации.

Сроки сдачи: мягкий дедлайн - 16 июня 2021 23:59 (среда), жесткий дедлайн - 19 июня 2021 23:59 (суббота)

1 Алгоритмы

1.1 Методы спуска: Общая концепция

Рассматриваем задачу гладкой безусловной оптимизации:

$$\min_{x \in \mathbb{R}^n} f(x).$$

Методы спуска итеративно строят последовательность точек $(x_k)_{k=0}^\infty$ из \mathbb{R}^n по правилу

$$x_{k+1} = x_k + \alpha_k d_k.$$

Число $k = 0, 1, \dots$ называется *номером итерации* метода. Скаляр $\alpha_k \geq 0$ называется *длиной шага*, а вектор $d_k \in \mathbb{R}^n$ называется *направлением поиска*. В методах спуска требуется, чтобы направление поиска d_k являлось *направлением спуска* для функции f в точке x_k , т. е. удовлетворяло неравенству

$$\langle \nabla f(x_k), d_k \rangle < 0.$$

В этом случае можно гарантировать, что для всех достаточно маленьких α_k значение функции f в новой точке x_{k+1} уменьшится:

$$f(x_{k+1}) < f(x_k).$$

Общая схема метода спуска приведена ниже:

Алгоритм 1 Общая схема метода спуска

Вход: Начальная точка x_0 ; максимальное число итераций K .

- 1: **for** $k \leftarrow 0$ **to** K **do**
- 2: *(Вызов оракула)* Вычислить $f(x_k)$, $\nabla f(x_k)$ и пр.
- 3: *(Критерий остановки)* Если выполнен критерий остановки, то выход.
- 4: *(Вычисление направления)* Вычислить направление спуска d_k .
- 5: *(Линейный поиск)* Найти подходящую длину шага α_k .
- 6: *(Обновление)* $x_{k+1} \leftarrow x_k + \alpha_k d_k$.
- 7: **end for**

Выход: Последняя вычисленная точка x_k

1.2 Критерий остановки

Идеальным критерием остановки в методе является проверка условия $f(x_k) - f^* < \tilde{\varepsilon}$, где f^* — минимальное значение функции f , а $\tilde{\varepsilon} > 0$ — заданная точность. Такой критерий целесообразно использовать, если оптимальное значение функции f^* известно. К сожалению, зачастую это не так, и поэтому нужно использовать другой критерий. Наиболее популярным является критерий, основанный на норме градиента: $\|\nabla f(x_k)\|_2^2 < \tilde{\varepsilon}$. Квадрат здесь ставят за тем, что для «хороших» функций невязка по функции $f(x_k) - f^*$ имеет тот же порядок, что и $\|\nabla f(x_k)\|_2^2$, а не $\|\nabla f(x_k)\|_2$;¹ например, если $\|\nabla f(x_k)\|_2 \sim 10^{-5}$, то $f(x_k) - f^* \sim 10^{-10}$. Наконец, для того, чтобы критерий не зависел от того, измеряется ли функция f «в метрах» или «в километрах» (т. е. не изменялся при переходе от функции f к функции tf , где $t > 0$), то имеет смысл использовать следующий относительный вариант критерия:

$$\|\nabla f(x_k)\|_2^2 \leq \varepsilon \|\nabla f(x_0)\|_2^2, \quad (1.1)$$

¹Например, это верно для сильно-выпуклых функций с липшицевым градиентом.

где $\varepsilon \in (0, 1)$ — заданная *относительная* точность. Таким образом, критерий останова (1.1) гарантирует, что метод уменьшит начальную невязку $\|\nabla f(x_0)\|_2$ в ε^{-1} раз. В этом задании Вам нужно будет во всех методах использовать критерий останова (1.1).

1.3 Линейный поиск

Рассматривается функция

$$\phi_k(\alpha) := f(x_k + \alpha d_k).$$

Заметим, что

$$\phi'_k(\alpha) = \langle \nabla f(x_k + \alpha d_k), d_k \rangle.$$

Поскольку d_k является направлением спуска, то $\phi'(0) = \langle \nabla f(x_k), d_k \rangle < 0$.

Условием Армихо для α называется выполнение следующего неравенства:

$$\phi_k(\alpha) \leq \phi_k(0) + c_1 \alpha \phi'_k(0),$$

где $c_1 \in (0, 0.5)$ — некоторая константа.

Для поиска точки α , удовлетворяющей условию Армихо, обычно используют следующую процедуру — метод дробления шага (бэктрекинг):

Алгоритм 2 Метод дробления шага

Вход: Функция $\phi_k : \mathbb{R}_+ \rightarrow \mathbb{R}$. Начальная точка: $\alpha_k^{(0)}$.

- 1: $\alpha \leftarrow \alpha_k^{(0)}$.
- 2: **while** $\phi_k(\alpha) > \phi(0) + c\alpha\phi'_k(0)$ **do**
- 3: $\alpha \leftarrow \alpha/2$.
- 4: **end while**

Выход: α

«Адаптивный» метод подбора шага запоминает величину α_k , найденную на текущей итерации и на следующей итерации начинает процедуру дробления с $\alpha_{k+1}^{(0)} := 2\alpha_k$. Исключение здесь составляют ньютоновские и квазиньютоновские методы — в этих методах процедуру дробления шага всегда нужно начинать с $\alpha_k^{(0)} := 1$.

Сильные условия Вульфа:

$$\begin{aligned} \phi_k(\alpha) &\leq \phi(0) + c_1 \alpha \phi'_k(0) \\ |\phi'_k(\alpha)| &\leq c_2 |\phi'_k(0)| \end{aligned}$$

Здесь $c_1 \in (0, 0.5)$, $c_2 \in (c_1, 1)$.

Самостоятельно реализовывать схему для сильных условий Вульфа не нужно. Используйте библиотечную реализацию (функция `scalar_search_wolfe2` из модуля `scipy.optimize.linesearch`). В ней начальная длина шага $\alpha_k^{(0)}$ автоматически выбирается равной 1.

1.4 Градиентный спуск

Градиентный спуск:

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k)$$

Можно рассматривать как метод спуска, в котором направление поиска d_k равно антиградиенту $-\nabla f(x_k)$. Длина шага α_k выбирается с помощью линейного поиска.

1.5 Метод Ньютона

Метод Ньютона:

$$x_{k+1} = x_k - \alpha_k [\nabla^2 f(x_k)]^{-1} \nabla f(x_k).$$

Для метода Ньютона очень важно использовать единичный шаг $\alpha_k = 1$, чтобы обеспечить локальную квадратичную сходимость. Поэтому в алгоритмах линейного поиска нужно всегда первым делом пробовать единичный шаг. Теория гарантирует, что в зоне квадратичной сходимости метода Ньютона единичный шаг будет удовлетворять условиям Армихо/Вульфа, и поэтому автоматически будет приниматься. Если единичный шаг не удовлетворяет условиям Армихо/Вульфа, то алгоритмы линейного поиска его уменьшат и, тем самым, обеспечат глобальную сходимость метода Ньютона.

Вычисление Ньютоновского направления $d_k = -[\nabla^2 f(x_k)]^{-1} \nabla f(x_k)$ эквивалентно решению линейной системы уравнений:

$$\nabla^2 f(x_k) d_k = -\nabla f(x_k).$$

Если гессиан — положительно определённая матрица: $\nabla^2 f(x_k) \succ 0$, то предпочтительным методом решения такой системы является *разложение Холецкого*, которое также, как и метод Гаусса, работает за $O(n^3)$, но является вычислительно более эффективным. Если матрица системы не является положительно определённой, то метод Холецкого сможет обнаружить и сообщить об этом.

1.6 Метод L-BFGS

Метод BFGS принадлежит классу *квазиньютоновских* методов, которые на каждом шаге аппроксимируют настоящий гессиан $\nabla^2 f(x_k)$ с помощью некоторой матрицы B_k и выбирают направление спуска d_k как решение следующей системы (аналогичной ньютоновской):

$$B_k d_k = -\nabla f(x_k) \quad \Leftrightarrow \quad d_k = -H_k \nabla f(x_k), \quad \text{где } H_k := B_k^{-1}.$$

Дальше, из текущей точки x_k , как обычно, выполняется шаг в этом направлении:

$$x_{k+1} = x_k + \alpha_k d_k,$$

где $\alpha_k > 0$ — длина шага, настраиваемая с помощью линейного поиска.

Основная работа на каждой итерации квазиньютоновского метода затрачивается на построение аппроксимации гессиана и вычисление направления поиска.

Начиная с $B_0 = I$, алгоритм пересчитывает аппроксимацию гессиана по правилу $B_{k+1} = B_k + U_k$, где U_k — некоторое низкоранговое обновление. Маленький ранг U_k необходим для построения эффективной процедуры вычисления обратной матрицы $H_{k+1} = B_{k+1}^{-1} = (B_k + U_k)^{-1}$. Конкретный вид обновления U_k следует из выполнения нескольких требований. Основное из них — *уравнение секущей*: $B_{k+1}(x_{k+1} - x_k) = \nabla f(x_{k+1}) - \nabla f(x_k)$, справедливое для всех квазиньютоновских методов. Одного этого уравнения недостаточно, чтобы однозначно определить B_{k+1} . Конкретную квазиньютоновскую схему получают с помощью наложения на аппроксимацию гессиана дополнительных требований.

Наиболее популярным и устойчивым на практике является *правило BFGS* (по фамилиям авторов: Бройден–Флетчер–Гольдфарб–Шанно). Обозначим:

$$\begin{aligned} s_k &:= x_{k+1} - x_k, \\ y_k &:= \nabla f(x_{k+1}) - \nabla f(x_k). \end{aligned}$$

Тогда, схема пересчета BFGS имеет следующий вид:

$$\begin{aligned} B_{k+1} &= B_k - \frac{B_k s_k s_k^T B_k}{\langle B_k s_k, s_k \rangle} + \frac{y_k y_k^T}{\langle y_k, s_k \rangle}, \\ H_{k+1} &= \left(I_n - \frac{s_k y_k^T}{\langle y_k, s_k \rangle} \right) H_k \left(I_n - \frac{y_k s_k^T}{\langle y_k, s_k \rangle} \right) + \frac{s_k s_k^T}{\langle y_k, s_k \rangle}. \end{aligned} \tag{1.2}$$

Метод L-BFGS является модификацией метода BFGS для случаев, когда не удаётся поместить матрицу H_k в память. Для этого на каждой итерации метода поддерживается история $\mathcal{H}_k := ((s_{k-i}, y_{k-i}))_{i=1}^l$ из последних l векторов, где l — некоторый параметр (типичное значение $l = 10$; при $k < l$ история \mathcal{H}_k состоит только из k пар.) Далее в качестве матрицы H_k выбирается матрица, полученная с помощью l -кратного рекуррентного применения формулы обновления (1.2), где в качестве начальной матрицы выбирается

$$H_{k-l} := \gamma_0^{(k)} I_n, \quad \text{где } \gamma_0^{(k)} := \frac{\langle y_{k-1}, s_{k-1} \rangle}{\langle y_{k-1}, y_{k-1} \rangle}. \quad (1.3)$$

Таким образом, метод L-BFGS является усеченной версией BFGS со специальным выбором начальной матрицы по формуле (1.3) (так называемое *правило Барзилая–Борвейна*).

Преимуществом такой схемы является то, что для ее реализации *никаких матриц хранить в памяти не требуется*. Действительно, напомним, что методу сама матрица H_k не нужна, а нужен вектор $d_k = -H_k \nabla f(x_k)$. Таким образом, нужна процедура, вычисляющая результат умножения L-BFGS матрицы H_k на произвольный вектор $v \in \mathbb{R}^n$. Умножая обе части (1.2) на v , видно, что такую процедуру можно организовать рекурсивно без явного формирования каких-либо матриц в памяти:

Алгоритм 3 Рекурсивное умножение L-BFGS матрицы на вектор

```

function BFGS_MULTIPPLY( $v, \mathcal{H}, \gamma_0$ )
  if  $\mathcal{H} = \emptyset$  then
    return  $\gamma_0 v$ 
  end if
   $(s, y) \leftarrow$  последняя пара из  $\mathcal{H}$ .
   $\mathcal{H}' \leftarrow \mathcal{H}$  без последней пары.
   $v' \leftarrow v - \frac{\langle s, v \rangle}{\langle y, s \rangle} y$ 
   $z \leftarrow$  BFGS_MULTIPPLY( $v', \mathcal{H}', \gamma_0$ )
  return  $z + \frac{\langle s, v \rangle - \langle y, z \rangle}{\langle y, s \rangle} s$ .
end function

```

Имея в распоряжении указанную процедуру, направление d_k вычислить легко:

Алгоритм 4 Вычисление направления поиска d_k в методе L-BFGS

```

function LBFGS_DIRECTION
   $(s, y) \leftarrow$  последняя пара из  $\mathcal{H}_k$ 
   $\gamma_0 \leftarrow \frac{\langle y, s \rangle}{\langle y, y \rangle}$ 
  return BFGS_MULTIPPLY( $-\nabla f(x_k), \mathcal{H}_k, \gamma_0$ )
end function

```

1.7 (Бонусная часть) Оптимизация вычислений

Рассмотрим случай $f(x) = \psi(Ax)$.

В этом случае

$$\nabla f(x) = A^T \nabla \psi(Ax).$$

Для линейного поиска:

$$\phi(\alpha) = \psi(Ax_k + \alpha Ad_k), \quad \phi'(\alpha) = \langle \nabla \psi(Ax_k + \alpha Ad_k), Ad_k \rangle.$$

Алгоритм 5 Общая схема метода спуска для $f(x) = \psi(Ax)$

```
1: for  $k \leftarrow 0$  to  $K - 1$  do
2:   (Вызов оракула) Вычислить  $f(x_k) = \psi(Ax_k)$ ,  $\nabla f(x_k) = A^T \nabla \psi(Ax_k)$  и пр.
3:   (Вычисление направления) Вычислить направление спуска  $d_k$ .
4:   (Линейный поиск) Найти подходящую длину шага  $\alpha_k$ :
5:     Вычислить  $\phi(0) = \psi(Ax_k)$ ,  $\phi'(0) = \langle \nabla \psi(Ax_k), Ad_k \rangle$ .
6:     Вычислить  $\phi(\bar{\alpha}_1) = \psi(Ax_k + \bar{\alpha}_1 Ad_k)$ ,  $\phi'(\bar{\alpha}_1) = \langle \nabla \psi(Ax_k + \bar{\alpha}_1 Ad_k), Ad_k \rangle$ .
7:     ...
8:     Вычислить  $\phi(\bar{\alpha}_s) = \psi(Ax_k + \bar{\alpha}_s Ad_k)$ ,  $\phi'(\bar{\alpha}_s) = \langle \nabla \psi(Ax_k + \bar{\alpha}_s Ad_k), Ad_k \rangle$ .
9:   (Обновление)  $x_{k+1} \leftarrow x_k + \bar{\alpha}_s d_k$ .  $\triangleright Ax_{k+1} = Ax_k + \bar{\alpha}_s Ad_k$ 
10: end for
```

Таким образом, в хорошей реализации должно быть в среднем лишь два матрично-векторных произведения: одно — чтобы вычислить градиент $A^T \nabla \psi(Ax_k)$, второе — чтобы вычислить Ad_k . Сами матрично-векторные произведения Ax_k можно пересчитывать, используя Ad_k .

2 Модели

2.1 Двухклассовая логистическая регрессия

Логистическая регрессия является стандартной моделью в задачах классификации. Для простоты рассмотрим лишь случай бинарной классификации. Неформально задача формулируется следующим образом. Имеется обучающая выборка $((a_i, b_i))_{i=1}^m$, состоящая из m векторов $a_i \in \mathbb{R}^n$ (называемых *признаками*) и соответствующих им чисел $b_i \in \{-1, 1\}$ (называемых *классами*). Нужно построить алгоритм $b(\cdot)$, который для произвольного нового вектора признаков a автоматически определит его класс $b(a) \in \{-1, 1\}$.

В модели логистической регрессии определение класса выполняется по знаку линейной комбинации компонент вектора a с некоторыми фиксированными коэффициентами $x \in \mathbb{R}^n$:

$$b(a) := \text{sign}(\langle a, x \rangle).$$

Коэффициенты x являются параметрами модели и настраиваются с помощью решения следующей оптимизационной задачи:

$$\min_{x \in \mathbb{R}^n} \left\{ \frac{1}{m} \sum_{i=1}^m \ln(1 + \exp(-b_i \langle a_i, x \rangle)) + \frac{\lambda}{2} \|x\|_2^2 \right\},$$

где $\lambda > 0$ — коэффициент регуляризации (параметр модели).

2.2 Разностная проверка градиента и гессиана

Проверить правильность реализации подсчета градиента можно с помощью конечных разностей:

$$[\nabla f(x)]_i \approx \frac{f(x + \varepsilon_1 e_i) - f(x)}{\varepsilon_1},$$

где $e_i := (0, \dots, 0, 1, 0, \dots, 0)$ — i -й базисный орт, а ε_1 — достаточно маленькое положительное число: $\varepsilon_1 \sim \sqrt{\varepsilon_{\text{mach}}}$, где $\varepsilon_{\text{mach}}$ — машинная точность ($\approx 10^{-16}$ для типа **double**).

Вторые производные:

$$[\nabla^2 f(x)]_{ij} \approx \frac{f(x + \varepsilon_2 e_i + \varepsilon_2 e_j) - f(x + \varepsilon_2 e_i) - f(x + \varepsilon_2 e_j) + f(x)}{\varepsilon_2^2}$$

Здесь $\varepsilon_2 \sim \sqrt[3]{\varepsilon_{\text{mach}}}$.

2.3 Разностная проверка произведения гессиана на вектор

Проверить правильность подсчета произведения гессиана $\nabla^2 f(x) \in \mathbb{S}^n$ на заданный вектор $v \in \mathbb{R}^n$ можно с помощью конечных разностей:

$$[\nabla^2 f(x)v]_i \approx \frac{f(x + \epsilon_2 v + \epsilon_2 e_i) - f(x + \epsilon_2 v) - f(x + \epsilon_2 e_i) + f(x)}{\epsilon_2^2},$$

где $e_i := (0, \dots, 0, 1, 0, \dots, 0)$ — i -й базисный орт, а $\epsilon_2 \sim \sqrt[3]{\epsilon_{\text{mach}}}$, где ϵ_{mach} — машинная точность ($\approx 10^{-16}$ для типа `double`).

3 Формулировка задания

- 1 Скачайте коды, прилагаемые к заданию:

<https://github.com/arodomanov/cmc-mipt17-opt-course/tree/master/task1>

Эти файлы содержат прототипы функций, которые Вам нужно будет реализовать. Некоторые процедуры уже частично или полностью реализованы.

- 2 Реализовать метод градиентного спуска (функция `gradient_descent` в модуле `optimization`) и процедуру линейного поиска (метод `line_search` в классе `LineSearchTool` в модуле `optimization`).

Рекомендация: Для поиска точки, удовлетворяющей сильным условиям Вульфа, воспользуйтесь библиотечной функцией `scalar_search_wolfe2` из модуля `scipy.optimize.linesearch`. Однако следует иметь в виду, что у этой библиотечной функции имеется один недостаток: она иногда не сходится и возвращает значение `None`. Если библиотечный метод вернул `None`, то запустите процедуру дробления шага (бэктрекинг) для поиска точки, удовлетворяющей условию Армихо.

- 3 Получить формулы для градиента и гессиана функции логистической регрессии. Выписать их в отчет в матрично-векторной форме² с использованием поэлементных функций, но без каких-либо суммирований. Также выписать в отчет выражение для самой функции логистической регрессии в матрично-векторной форме (без явных суммирований).

- 4 Реализовать оракул логистической регрессии (класс `LogRegL2Oracle` в модуле `oracles`). Также доделать реализацию вспомогательной функции `create_log_reg_oracle` в модуле `oracles`.

Замечание: Реализация оракула должна быть полностью векторизованной, т. е. код не должен содержать никаких циклов.

Замечание: Ваш код должен поддерживать как плотные матрицы A типа `np.array`, так и разреженные типа `scipy.sparse.csr_matrix`.

Замечание: Нигде в промежуточных вычислениях не стоит вычислять значение $\exp(-b_i \langle a_i, x \rangle)$, иначе может произойти переполнение. Вместо этого следует напрямую вычислять необходимые величины с помощью специализированных для этого функций: `np.logaddexp` для $\ln(1 + \exp(\cdot))$ и `scipy.special.expit` для $1/(1 + \exp(\cdot))$.

- 5 Реализовать подсчет разностных производных (функции `grad_finite_diff` и `hess_finite_diff` в модуле `oracles`). Проверить правильность реализации подсчета градиента и гессиана логистического оракула с помощью реализованных функций. Для этого сгенерируйте небольшую модельную выборку (матрицу A и вектор b) и сравните значения, выдаваемые методами `grad` и `hess`, с соответствующими разностными аппроксимациями в нескольких пробных точках x .

²В матрично-векторной форме допускается использование операций матричного сложения/произведения, умножения на скаляр, транспонирования, стандартного скалярного произведения, поэлементного произведения (*произведение Адамара*; символ \odot), а также применения ко всем элементам вектора некоторой скалярной функции. Кроме этого, допускается использование стандартных матриц/векторов (заданного размера): единичная матрица I_n , нулевая матрица $0_{m \times n}$, нулевой вектор 0_n , вектор из всех единиц $1_n := (1, \dots, 1)$.

6 Реализовать метод Ньютона (функция `newton` в модуле `optimization`).

Замечание: Для поиска направления в методе Ньютона не нужно в явном виде обращать гессиан (с помощью функции `np.linalg.inv`) или использовать самый общий метод для решения системы линейных уравнений (`numpy.linalg.solve`). Вместо этого следует учесть тот факт, что в рассматриваемой задаче гессиан является симметричной положительно определенной матрицей и воспользоваться разложением Холецкого (функции `scipy.linalg.cho_factor` и `scipy.linalg.cho_solve`).

7 Реализуйте метод L-BFGS (функция `lbfgs` в модуле `optimization`).

8 Провести эксперименты, описанные ниже. Написать отчет.

9 (Бонусная часть) Реализовать оптимизированный оракул логистической регрессии, который запоминает последние матрично-векторные произведения (класс `LogRegL2OptimizedOracle` в модуле `optimization`). Оптимизированный оракул отличается от обычного в следующих трех пунктах:

1. При последовательных вычислениях значения функции (метод `func`), градиента (метод `grad`) и гессиана (метод `hess`) в одной и той же точке x , матрично-векторное произведение Ax не вычисляется повторно.
2. В процедурах `func_directional` и `grad_directional` выполняется предподсчет матрично-векторных произведений Ax и Ad . Если эти процедуры вызываются последовательно для одних и тех же значений точки x и/или направления d , то матрично-векторные произведения Ax и/или Ad заново не вычисляются. Если перед вызовом или после вызова `func_directional` и/или `grad_directional` присутствуют вызовы `func` и/или `grad` и/или `hess` в той же самой точке x , то матрично-векторное произведение Ax не должно вычисляться повторно.
3. Методы `func_directional` и `grad_directional` запоминают внутри себя последнюю тестовую точку $\hat{x} := x + \alpha d$, а также соответствующее значение матрично-векторного произведения $A\hat{x} = Ax + \alpha Ad$. Если далее одна из процедур `func`, `grad`, `hess`, `func_directional`, `grad_directional` вызывается в точке \hat{x} , то соответствующее матрично-векторное произведение $A\hat{x}$ заново не вычисляется.

3.1 Эксперимент: Траектория градиентного спуска на квадратичной функции

Проанализируйте траекторию градиентного спуска для нескольких квадратичных функций: придумайте две-три квадратичные *двумерные* функции, на которых работа метода будет отличаться, нарисуйте графики с линиями уровня функций и траекториями методов.

Попробуйте ответить на следующий вопрос: *Как отличается поведение метода в зависимости от числа обусловленности функции, выбора начальной точки и стратегии выбора шага (константная стратегия, Армизо, Вульф)?*

Для рисования линий уровня можете воспользоваться функцией `plot_levels`, а для рисования траекторий — `plot_trajectory` из файла `plot_trajectory_2d.py`, прилагающегося к заданию.

Также обратите внимание, что оракул квадратичной функции `QuadraticOracle` уже реализован в модуле `oracles`. Он реализует функцию $f(x) = (1/2)\langle Ax, x \rangle - \langle b, x \rangle$, где $A \in \mathbb{S}_{++}^n$, $b \in \mathbb{R}^n$.

3.2 Эксперимент: Зависимость числа итераций градиентного спуска от числа обусловленности и размерности пространства

Исследуйте, как зависит число итераций, необходимое градиентному спуску для сходимости, от следующих двух параметров: 1) числа обусловленности $\kappa \geq 1$ оптимизируемой функции и 2) размерности пространства n оптимизируемых переменных.

Для этого для заданных параметров n и κ сгенерируйте случайным образом квадратичную задачу размера n с числом обусловленности κ и запустите на ней градиентный спуск с некоторой фиксированной требуемой точностью. Замерьте число итераций $T(n, \kappa)$, которое потребовалось сделать методу до сходимости (успешному выходу по критерию остановки).

Рекомендация: Проще всего сгенерировать случайную квадратичную задачу размера n с заданным числом обусловленности κ следующим образом. В качестве матрицы $A \in \mathbb{S}_{++}^n$ удобно взять просто диагональную матрицу $A = \text{Diag}(a)$, у которой диагональные элементы сгенерированы случайно в пределах $[1, \kappa]$, причем $\min(a) = 1$, $\max(a) = \kappa$. В качестве вектора $b \in \mathbb{R}^n$ можно взять вектор со случайными элементами. Диагональные матрицы удобно рассматривать, поскольку с ними можно эффективно работать даже при больших значениях n . Рекомендуется хранить матрицу A в формате разреженной диагональной матрицы (см. `scipy.sparse.diags`).

Зафиксируйте некоторое значение размерности n . Переберите различные числа обусловленности κ по сетке и постройте график зависимости $T(\kappa, n)$ против κ . Поскольку каждый раз квадратичная задача генерируется случайным образом, то повторите этот эксперимент несколько раз. В результате для фиксированного значения n у Вас должно получиться целое семейство кривых зависимости $T(\kappa, n)$ от κ . Нарисуйте все эти кривые одним и тем же цветом для наглядности (например, красным).

Теперь увеличьте значение n и повторите эксперимент снова. Вы должны получить новое семейство кривых $T(n', \kappa)$ против κ . Нарисуйте их все одним и тем же цветом, но отличным от предыдущего (например, синим).

Повторите эту процедуру несколько раз для других значений n . В итоге должно получиться несколько разных семейств кривых — часть красных (соответствующих одному значению n), часть синих (соответствующих другому значению n), часть зеленых и т. д.

Обратите внимание, что значения размерности n имеет смысл перебирать по логарифмической сетке (например, $n = 10$, $n = 100$, $n = 1000$ и т. д.).

Какие выводы можно сделать из полученной картинки?

3.3 Эксперимент: Выбор размера истории в методе L-BFGS

Исследуйте, как влияет размер истории в методе L-BFGS на поведение метода.

Прежде всего, оцените размер требуемой памяти и сложность итерации метода L-BFGS в зависимости от размера истории l и размерности пространства n . (Здесь не нужно учитывать сложность оракула.)

Рассмотрите несколько вариантов выбора размера истории (например, $l = 0$, $l = 1$, $l = 5$, $l = 10$, $l = 50$, $l = 100$) и постройте следующие графики:

- Зависимость относительного квадрата нормы градиента $\|\nabla f(x_k)\|_2^2 / \|\nabla f(x_0)\|_2^2$ (в логарифмической шкале) против номера итерации.
- Зависимость относительного квадрата нормы градиента $\|\nabla f(x_k)\|_2^2 / \|\nabla f(x_0)\|_2^2$ (в логарифмической шкале) против реального времени работы.

При этом разные варианты выбора размера истории нужно рисовать на одном и том же графике.

В качестве тестовой функции возьмите логистическую регрессию с l^2 -регуляризатором на данных *gisette* или *news20.binary* с сайта LIBSVM³. Коэффициент регуляризации и начальную точку выберите стандартным образом: $\lambda = 1/m$, $x_0 = 0$.

Какие выводы можно сделать?

3.4 Эксперимент: Сравнение методов на реальной задаче логистической регрессии

Сравните метод Ньютона, метод L-BFGS и градиентный спуск на реальной задаче логистической регрессии. В качестве реальных данных используйте следующие наборы данных с сайта LIBSVM:

³<http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/>.

w8a, gisette, real-sim, news20.binary, rcv1.binary. Коэффициент регуляризации возьмите стандартным: $\lambda = 1/m$. Параметры всех методов возьмите равными параметрам по умолчанию. Начальную точку выберите $x_0 = 0$.

Постройте следующие графики:

- (a) Зависимость значения функции против номера итерации метода.
- (b) Зависимость значения функции против реального времени работы.
- (c) Зависимость относительного квадрата нормы градиента $\|\nabla f(x_k)\|_2^2 / \|\nabla f(x_0)\|_2^2$ (в логарифмической шкале) против реального времени работы.

При этом все методы нужно рисовать на одном и том же графике.

Какие выводы можно сделать по результатам этого эксперимента? Какой из методов лучше и в каких ситуациях?

3.5 (Бонусная часть) Эксперимент: Оптимизация вычислений в градиентном спуске

Сравнить градиентный спуск на логистической регрессии для обычного оракула и оптимизированного.

В качестве выборки использовать модельную с размерами $m = 10000$, $n = 8000$. Пример генерации модельной выборки из стандартного нормального распределения:

```
np.random.seed(31415)
m, n = 10000, 8000
A = np.random.randn(m, n)
b = np.sign(np.random.randn(m))
```

Коэффициент регуляризации выбрать стандартным $\lambda = 1/m$.

Параметры метода взять равными параметрам по умолчанию. Начальную точку выбрать $x_0 = 0$.

Нарисовать графики:

- (a) Зависимость значения функции от номера итерации.
- (b) Зависимость значения функции от реального времени работы метода.
- (c) Зависимость относительного квадрата нормы градиента $\|\nabla f(x_k)\|_2^2 / \|\nabla f(x_0)\|_2^2$ (в логарифмической шкале) против реального времени работы.

При этом оба метода (с обычным оракулом и с оптимизированным) нужно рисовать на одном и том же графике.

Объясните, почему траектории обоих методов на первом графике совпадают.

3.6 (Бонусная часть) Эксперимент: Стратегия выбора длины шага в градиентном спуске

Исследовать, как зависит поведение метода от стратегии подбора шага: константный шаг (попробовать различные значения), бэктрэкинг (попробовать различные константы c), условия Вульфа (попробовать различные параметры c_2).

Рассмотрите квадратичную функцию и логистическую регрессию с модельными данными (сгенерированными случайно).

Запустите для этих функций градиентный спуск с разными стратегиями выбора шага *из одной и той же начальной точки*.

Нарисуйте кривые сходимости (относительная невязка по функции в логарифмической шкале против числа итераций – для квадратичной функции, относительный квадрат нормы градиента в логарифмической шкале против числа итераций – для логистической регрессии) для разных стратегий на одном графике.

Попробуйте разные начальные точки. Ответьте на вопрос: *Какая стратегия выбора шага является самой лучшей?*

3.7 (Бонусная часть) Эксперимент: Какая точность оптимизации нужна в реальных задачах?

В реальных задачах целевая функция, которая оптимизируется методами оптимизации, как правило, не является конечным критерием качества решения задачи. Например, рассмотрим задачу классификации и модель логистической регрессии. В этом случае оптимизируемой функцией является логистическая функция потерь. Однако само значение логистической функции потерь, с точки зрения решения задачи классификации, представляет мало интереса. А что, действительно, представляет интерес — это, например, процент ошибок при классификации на тестовой выборке. Возникает естественный вопрос: как влияет точность оптимизации целевой функции на итоговое качество решения самой задачи?

В этом эксперименте Вам предлагается исследовать этот вопрос для задачи бинарной классификации и модели логистической регрессии с l^2 -регуляризатором. Для этого выберите несколько реальных наборов данных и выполните следующий эксперимент.

Возьмите любой метод оптимизации (например, L-BFGS) и запустите его (на обучающей выборке) с разными параметрами требуемой точности⁴ ϵ . Для каждого ϵ возьмите итоговую точку \hat{x} , которую вернул метод, и сравните качество прогноза логистической регрессии $\hat{b}_{\text{test}} := \text{sign}(A_{\text{test}}\hat{x}_{\text{test}})$ с истинными значениями меток b_{test} . При сравнении используйте процент ошибок — среднее число позиций, в которых векторы b_{test} и \hat{b}_{test} отличаются. Нарисуйте график зависимости процента ошибок против точности оптимизации ϵ (в логарифмической шкале). Коэффициент регуляризации и начальную точку возьмите стандартными: $\lambda = 1/m$ и $x_0 = 0$.

В чем разница между маленькой точностью оптимизации и большой? Какие выводы можно сделать?

Рекомендация. Параметр ϵ имеет смысл перебрать по логарифмической сетке: от самой маленькой точности $\epsilon = 1$ (никакой оптимизации, вернуть начальную точку x_0) до самой большой точности $\epsilon = 10^{-6}$ или $\epsilon = 10^{-8}$ (оптимизировать функцию до «машинной» точности).

4 Оформление задания

Результатом выполнения задания являются

- (a) Файлы `optimization.py`, `oracles.py` и `utils.py` с реализованными методами и оракулами.
- (b) Полные исходные коды для проведения экспериментов и рисования всех графиков. Все результаты должны быть воспроизводимыми. Если вы используете случайность — зафиксируйте `seed`.
- (c) Отчет в формате PDF о проведенных исследованиях.

Каждый проведенный эксперимент следует оформить как отдельный раздел в PDF документе (название раздела — название соответствующего эксперимента). Для каждого эксперимента необходимо сначала написать его описание: какие функции оптимизируются, каким образом генерируются данные, какие методы и с какими параметрами используются. Далее должны быть представлены результаты

⁴Напомним, что во всех наших методах оптимизации $\epsilon \in (0, 1)$ задает относительную точность по квадрату нормы градиента: $\|\nabla f(\hat{x})\|_2^2 \leq \epsilon \|\nabla f(x_0)\|_2^2$.

соответствующего эксперимента — графики, таблицы и т. д. Наконец, после результатов эксперимента должны быть написаны Ваши выводы — какая зависимость наблюдается и почему.

Важно: Отчет не должен содержать никакого кода. Каждый график должен быть прокомментирован — что на нем изображено, какие выводы можно сделать из этого эксперимента. Обязательно должны быть подписаны оси. Если на графике нарисовано несколько кривых, то должна быть легенда. Сами линии следует рисовать достаточно толстыми, чтобы они были хорошо видимыми.

5 Проверка задания

Перед отправкой задания обязательно убедитесь, что Ваша реализация проходит автоматические *предварительные* тесты `presubmit_tests.py`, выданные вместе с заданием. Для этого, в зависимости от того, реализовывали ли Вы бонусную часть или нет, запустите одну из следующих команд:

- (a) Для базовой части (проверяются только базовые тесты):

```
nosetests3 presubmit_tests.py
```

- (b) Для бонусной части (проверяются как базовые, так и бонусные тесты):

```
nosetests3 presubmit_tests.py -a 'bonus'
```

Важно: Файл с тестами может измениться. Перед отправкой обязательно убедитесь, что ваша версия `presubmit_tests.py` — последняя.

Важно: Решения, которые не будут проходить тесты `presubmit_tests.py`, будут автоматически оценены в 0 баллов. Проверяющий не будет разбираться, почему Ваш код не работает и читать Ваш отчет.

Оценка за задание будет складываться из двух частей:

- (a) Правильность и эффективность реализованного кода.
- (b) Качество отчета

Правильность и эффективность реализованного кода будет оцениваться автоматически с помощью независимых тестов (отличных от предварительных тестов). Качество отчета будет оцениваться проверяющим. При этом оценка может быть субъективной и апелляции не подлежит.

За реализацию модификаций алгоритмов и хорошие дополнительные эксперименты могут быть начислены дополнительные баллы. Начисление этих баллов является субъективным и безапелляционным.

Важно: Практическое задание выполняется самостоятельно. Если вы получили ценные советы (по реализации или проведению экспериментов) от другого студента, то об этом должно быть явно написано в отчёте. В противном случае «похожие» решения считаются плагиатом и все задействованные студенты (в том числе те, у кого списали) будут сурово наказаны.