Санкт-Петербургский государственный университет
Прикладная математика и информатика

Кизеев Данил Владимирович
Группа № 321

«Анализ статьи по увеличению скорости работы МДМ-метода»

Курсовая работа

Научный руководитель:
доктор техн. наук Фрадков Александр Львович
Кафедра теоретической кибернетики

Санкт-Петербург
2020

Введение

В данной работе был разобрана статья [1], в которой описан метод, ускоряющий работу МДМ-метода [2]. В статье показано, что для выпуклой оболочки точек, содержащей в себе ноль, задача МДМ-метод сходится медленно из-за образующихся "циклов". Предложен способ - находить явно такие циклы и использовать их для нового приближения к оптимальной точке, ближайшей к началу координат. Хоть и в статье написано, что метод даёт выигрыш на вполне больших данных, поскольку требует тяжёлых вычислительных затрат на поиск "цикла мы покажем, что уже на небольшом примере - 30 точках в $\mathbb{R}^2$ метод будет давать выигрыш в два раза по итерациям.
Программа была оформлена на языке *Python 3.7*.

Краткое описание МДМ-метода.

1°.

Пусть в пространстве $\mathbb{R}^n$ заданы m точек,

$$H = \{a_i\}_{i=1}^m.$$

Обозначим через G выпуклую оболочку множества H.

Ставится задача: *найти точку из G, ближайшую(в евклидовой норме) к началу координат.* Задачу можно записать так:

$$||v||^2 \to \min_{[v \in G]}. \tag{1}$$

Задача (1) имеет решение и оно единственно, обозначим его v_*.

2°.

Обозначим через матрицу со столбцами $a_1, ..., a_m$. Тогда любой вектор v из выпуклой оболочки G множества H допускает представление.

$$v = Ap,\ p \geq (O) \sum_{i=1}^{m} p[i] = 1. \tag{2}$$

Носитель вектора p обозначим $M_+(p)$ так, что

$$M_+(p) = \{i \in 1 : m | p[i] > 0\}.$$

Также введём величину

$$\Delta(p) = \max_{i \in M_+(p)} \langle a_i, v \rangle - \min_{i \in 1:m} \langle a_i, v \rangle,$$

где $v = Ap$. Вектор удовлетворяет (2).

3°.

Возьмём начальное приближение $v_0 \in G$. Мной была выбрана точка из границы выпуклой оболочки, также в программе есть параметр, который позволяет выбрать любое начальное приближение и анализировать сходимость.
Пусть дано k-е приближение $v_k = Ap_k$. Вектор v_{k+1} строится следующим образом: Найдём индексы $i'_k \in M_+(p_k)$ и i''_k такие, что

$$\max_{i \in M_+(p)} \langle a_i, v_k \rangle - \min_{i \in 1:m} \langle a_i, v_k \rangle = \langle a_{i'_k}, v_k \rangle,$$
$$\min_{i \in 1:m} \langle a_i, v_k \rangle - \min_{i \in 1:m} \langle a_i, v_k \rangle = \langle a_{i''_k}, v_k \rangle.$$

В этом случае

$$\Delta_k := \Delta(p_k) = \langle a'_k - a''_k, v_k \rangle.$$

Если $\Delta_k = 0$, то v_k - решение, иначе обновляем вектор v_k :

$$v_{k+1}(t) = v_k - t_k p_k[i'_k](a'_k - a''_k), \quad t \in [0,1], \tag{3}$$

$$\text{где } t_k = \max\left\{1, \frac{\Delta_k}{p_k[i'_k] \, \|a'_k - a''_k\|^2}\right\}. \tag{4}$$

$$p_{k+1}[i] = \begin{cases} p_k[i], & \text{при } i \neq i'_k, i \neq i''_k, \\ (1 - t_k)p_k[i'_k], & \text{при } i = i'_k, \\ p_k[i''_k] + t_k p_k[i'_k], & \text{при } i = i''_k. \end{cases}$$

Проблема сходимости.

Метод быстро сходится, когда выпуклая оболочка отделена от нуля. Когда же многогранник включает в себя начало координат, всё не так радужно - алгоритм петляет, сходясь по "спирали". (см. Рис. 1). На данном рисунке можно заметить

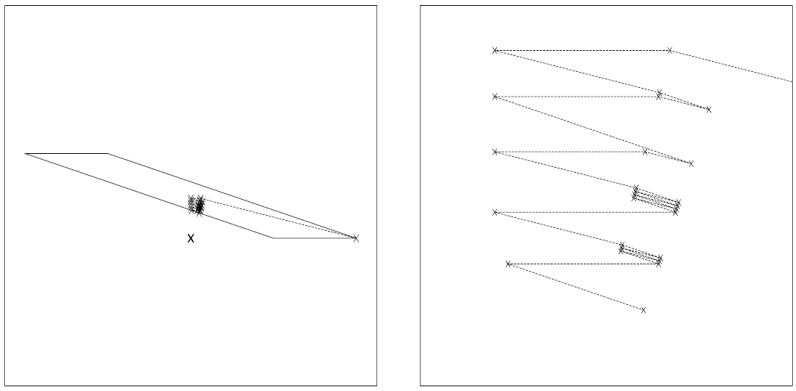

Рис. 1: **МДМ-методу может потребоваться много итераций даже для простых задач. Справа показано, что они имеют циклическую структуру.**

повторяющиеся зигзаги - это векторы, которые в (3) задавались как $(a'_k - a''_k)$. Обозначим эти векторы за $-D_T$. Именно проблема сходимости и её решение были описаны в [1]. На практике было замечено, что, если вектор D_T повторился через K шагов из основной последовательности векторов D_{T-K}, то очень вероятно, что в следующих итерациях он повторится снова на шагах $D_{T-j}, 1 \leq j \leq K-1$. Получим $D_{t-j} = D_{T-j}$ для некоторого $t > T$. Это можно увидеть и на Рис.1, который также показывает возможный выход из данной ситуации. Заметим, если взять $V = \lambda_1 D_1 + \lambda_2 D_2$, где $\lambda_k = t_k p_k[i'_k] \forall k \in 1 : m$ в (4), в качестве нового направления для обновления вектора v_k, то таким образом мы выберемся из цикла и, снова вернувшись к МДМ-методу, моментально сойдёмся к нулю. Далее будем пользоваться обозначениями λ_k. Для более общего цикла $D_{T-K}, D_{T-K+1}, ..., D_{T-1}, D_T = D_{T-K}$, определим $V := \sum_{j=1}^{K} \lambda_{T-j} D_{T-j}$ и, в отличие от стандартных обновлений вектора v_k, рассмотрим обновление $v_{k+1} = v_k + \lambda_T V$, где λ_T минимизирует норму $\|v_T + \lambda V\|^2$. Легко видеть, что оптимальное значение λ_T достигается, когда $\lambda_T = -\frac{v_T \cdot V}{\|V\|^2}$.
Коэффициенты $p_k[i]$ обновляются следующим образом:

$$p_k[i_k] = \begin{cases} p_k[i_k], & \text{при } i \neq i'_k, i \neq i''_k, \\ p_k[i'_k] - \lambda_T \cdot \lambda_{T-j}, & \text{при } i = i'_k; \\ p_k[i''_k] + \lambda_T \cdot \lambda_{T-j}, & \text{при } i = i''_k. \end{cases}$$

Также вспомним, что каждый новый коэффициент $p_{k+1}[i]$ должен лежать в отрезке $[0, 1]$ - тогда нужно сделать ограничение, чтобы наши коэффициенты не

выходили за эти границы, т.е: $0 \leq p_k[i'(')_k] \pm \lambda_T \cdot \lambda_{T-j} \leq 1$. Тогда нужно сделать дополнительную проверку на λ_T : $\lambda_T \leq \begin{cases} \min\left\{\frac{1-p_k[i''_k]}{\lambda_{T-j}}\right\}, \\ \min\left\{\frac{-p_k[i'_k]}{\lambda_{T-j}}\right\}. \end{cases}$

Из вышеприведённого можно сделать выводы, что в программе нужно будет сохранять коэффициенты λ_{T-j} для каждого нового обновления вектора.

После того, как мы преодолели этот цикл, вернёмся к обычным обновлениям МДМ-метода: без цикла он быстро сойдётся.

Техническая структура задачи.

Программа была написана на языке *Python 3.7*. В процессе были использованы библиотеки *numpy, matplotlib.pyplot, scipy.spatial*. Для структур, графиков и нахождения минимальной выпуклой оболочки соответственно.
Для случаев размерности $\mathbb{R}^2, \mathbb{R}^3$ программа строит графики с выпуклой оболочкой и красной точкой - результатом работы программы, а то есть, началом координат.
В программе есть возможность сгенерировать новое множество точек, для которых мы будем решать задачу, либо же выполнить программу на множестве, которое задано по умолчанию. Если пользователь решил задать новое множество, юзеру предлагается ввести количество точек, каким(ускоренным или обычным) МДМ-методом решать задачу и размерность пространства.
Далее программа по данным точкам строит наименьшую выпуклую оболочку, обращаясь к подключенным библиотекам.
После выполняется само обращение к солверу, который работает следующим образом: если пользователь решил использовать ускоренный МДМ-метод, то программа сначала выполняет первые итерации до того, пока не найдёт и построит цикл. Далее программа совершает обновление приближения результирующего вектора по правилу, описанному в предыдущем пункте. После всего этого снова включается МДМ-метод.
В процессе всех итераций на печать выводятся: разность D_T - **Difference**, величина $\Delta_p := \langle a'_k - a''_k, v_k \rangle$ - **delta_p**, норма $\|D_T\|$ - **np.linalg.norm(diff)**, приближение результирующего вектора на данном шаге v_k - **Vector current**, носитель вектора $M_+(p)$ - **Supp_vector**.

Анализ.

Метод с поиском цикла действительно ускоряет МДМ-метод - это можно проверить, сначала выполнив нашу программу при помощи обычного метода, а затем ускоренного. Уже на примере по умолчанию программа даёт выигрыш по итерациям в два раза, увеличивая точность. В количество итераций мы также включили итерации, необходимые программе для нахождения и построения цикла.

Очевидно, что истинный выигрыш ускоренный МДМ-метод будет иметь на больших данных - там ускорение работы и уменьшение итераций будет в разы больше.

Далее представлен код программы, как она есть:

MDMmethod

```
import numpy as np
from mpl_toolkits import mplot3d
import matplotlib.pyplot as plt
from scipy.spatial import ConvexHull

def GenerPoints(a, alpha, num_points, dim):
    return alpha * np.random.rand(num_points, dim) + a

def GetHullandPlot(points, dim):
    hull = ConvexHull(points)
    if dim == 2:
        plt.plot(points[:, 0], points[:, 1], 'o')
        for simplex in hull.simplices:
            plt.plot(points[simplex, 0], points[simplex, 1], 'b-')
    elif dim == 3:
        fig = plt.figure()
        ax = plt.axes(projection='3d')
        plt.plot(points[:, 0], points[:, 1], points[:, 2], 'o')
        for simplex in hull.simplices:
            plt.plot(points[simplex, 0], points[simplex, 1], points[simplex, 2],
                'b-')
    else:
        print('There is no way to plot graph in dim > 3, but hull has found
            successfully!')
    return hull


class MDM(object):
    __class__ = 'MDM'
    __doc__ = """
        This is an implementation the accelerated Mitchell-Demyanov-Malozemov
            method for
        finding nearest to coordinates beginning point.
        Also plots convex-hull and optimal solution in 2- and 3-dimensional
            cases.
    """

    def __init__(self, points, hull, dim, accel):
        self._dim = dim
```

```
        self._points = points.copy()
        self._hull = hull
        self._A_matrix = points.copy().transpose()
        self.isAccelerated = accel                    #which method we're using
        self.iterations = None
        self.delta_p = None
        self.p_vector = None
        self.vector_current = None
        self.supp_vector = None                       #supp for vector p (i.e. {i
            \in 0 : dim - 1 | p[i] > 0} )


    def solve(self):
        V = 0
        iterations = 0

        delta_p = 1
        p_vector = [0 for i in range(0, len(self._points))]
        supp_vector = []
        t_param_vector =[]

        MIN_set = []
        MAX_set = []
        diff_vector = []                              #for cycles finding
        P_vectors = []                                #matrix of p_vectors
        V_vectors = []
        cycle_constructed = False
        cycle_is_constructing = False
        special_upd_done = False                      #whether special update Wn
            = W + lambdaV is done
        cycle_current_size = 0                        #we will search actual size of
            cycle

        initial_approximation = 1                      #it can be changed for
            lowering iterations sake;
        #for first approximation we'll just take point from a board of hull -
            cause it's easy reduced
        vector_current =
            self._points[self._hull.vertices[initial_approximation]].copy()
            #need copy() there for non-changing _points
        supp_vector.append(self._hull.vertices[initial_approximation])
```

```
                                #approximation => get vect_0
        p_vector[self._hull.vertices[initial_approximation]] = 1
                                        #working right concat
71      #then we need to find vect_{k+1} iteratively

73      while delta_p > 0.000001 and iterations < 500 and len(supp_vector) != 0:
            if self.isAccelerated is True and cycle_constructed is True and
                special_upd_done is False:
75              for i in range(cycle_size):   #constructing V as linear
                    combination of D's that we used previously
                    V += -1 * t_param_vector[cycle_start + i] *
                        diff_vector[cycle_start + i]
77              p_vector = P_vectors[cycle_start]                    #returning to
                    value where cycle had begun
                vector_current = V_vectors[cycle_start]       #returning
79              supp_vector = []                              #returning
                for i in range(len(p_vector)):                #returning
81                  if p_vector[i] > 0.0000001:
                        supp_vector.append(i)
83
                lambda_t = -np.dot(vector_current, V) / np.linalg.norm(V) ** 2
85              for i in range(cycle_size):
                    if t_param_vector[i] > 0:
87                      if lambda_t > (1 - p_vector[MIN_set[i]]) /
                            t_param_vector[i]:
                            lambda_t = (1 - p_vector[MIN_set[i]]) /
                                t_param_vector[i]
89                  elif t_param_vector[i] < 0:
                        if lambda_t > -p_vector[MAX_set[i]] /
                            t_param_vector[i]:
91                          lambda_t = -p_vector[MAX_set[i]] /
                                t_param_vector[i]
                vector_current += lambda_t * V
93              for i in range(cycle_size):
                    p_vector[MAX_set[i]] -= lambda_t * t_param_vector[i]
95                  p_vector[MIN_set[i]] += lambda_t * t_param_vector[i]
                special_upd_done = True  #once it's done we're forgiving about
                    that
97


99      mult = np.dot(self._points[supp_vector], vector_current)
```

```
                ind_max = np.argmax(mult)          #finding max for indices in
                    supp_vector
101             ind_max = supp_vector[ind_max]  #finding max general in our mult
                    product

103             mult = np.matmul(vector_current, self._A_matrix)
                ind_min = np.argmin(mult)
                                                                # i'' _k
105             if  self.isAccelerated  is  True and cycle_constructed is False:
                    MIN_set.append(ind_min)
107                 MAX_set.append(ind_max)
                diff  = self._points[ind_max] - self._points[ind_min]
109             print('\nDifference:  ' + str(diff))
                delta_p = np.dot(diff, vector_current)
111
                if  delta_p > 0.000001:                    #if not bigger, then we've
                    found a solution
113                 print('delta_p:  ' + str(delta_p))
                    print('p_vector[ind_max] = ' + str(p_vector[ind_max]) +
                        '\nnp.linalg.norm(diff)):  '
115                     + str(np.linalg.norm(diff)))
                    t_param = delta_p /(p_vector[ind_max] * (np.linalg.norm(diff))
                        ** 2) # recounting all variables
117                 if  t_param >= 1:
                        t_param = 1
119
                    if  self.isAccelerated  is  True:          #if using accelerated
                        MDM-method
121                     if  iterations  > 0 and cycle_is_constructing is False:
                            #constructing cycle(active finding cycle, i mean,
                            active-active)
                            contains = np.where(np.all(diff_vector == diff, axis =
                                1))[0]     #finds if diff_vector contains diff
123                         if  len(contains) != 0:       #found first element of cycle
                                cycle_is_constructing = True        #cycle is
                                    constructing now
125                             cycle_start = contains[0]                   #index
                                    of first element of cycle; not changing
                                cycle_size = iterations - cycle_start          #not
                                    changing
127                             cycle_current_size += 1        #this var for checking
```

```
                    if all variables actually are cycle
                P_vectors.append(p_vector.copy())
                V_vectors.append(vector_current.copy())
                t_param_vector.append(t_param) #saving t_params for
                    constructing V in the future
                diff_vector.append(diff)              #saving D_i
            elif cycle_is_constructing is True and cycle_constructed is
                False:
                if cycle_current_size < cycle_size and \
                        np.where(np.all(diff_vector == diff, axis = 1))[0]
                            \
                        == (cycle_start + cycle_current_size):
                    cycle_current_size += 1
                    diff_vector.append(diff)
                    t_param_vector.append(t_param)
                else:
                    cycle_constructed = True
                    print('CYCLE FOUND AND CONSTRUCTED
                        SUCCESSFULLY!')
            elif iterations == 0:
                P_vectors.append(p_vector.copy())
                V_vectors.append(vector_current.copy())
                t_param_vector.append(t_param)
                diff_vector.append(diff)


    vector_current -= t_param * p_vector[ind_max] * diff
    supp_vector = []                    #recounting
    temp1 = t_param * p_vector[ind_max]
    temp2 = (1 - t_param)
    p_vector[ind_min] += temp1
    p_vector[ind_max] *= temp2

    for i in range(len(p_vector)):
        if p_vector[i] > 0.0000001:
            supp_vector.append(i)
print('Vector current: ' + str(vector_current))
iterations += 1
print('Iterations : ' + str(iterations))
print('Supp_vector: ' + str(supp_vector))
```

```
        return vector_current
165
167
169
points =np.array([[ -73.337555 ,     -4.82192605],
171         [    9.36299101,    14.79378288],
            [   33.74875017,    10.02043701],
173         [  133.04981839,    92.18760616],
            [-105.00396348,  -69.46640213],
175         [   32.54560694,    43.96449265],
            [  -78.01174375,   61.08025333],
177         [   92.03366094,  -51.6208306 ],
            [   17.22114877,    54.92524147],
179         [  -87.14266467, 128.5875058 ],
            [  -35.76597696, -161.63324815],
181         [  156.36709765,  -55.60266369],
            [   41.00897625,  -54.92133061],
183         [  129.50005618,  -39.14660553],
            [  101.99767049,     5.91893179],
185         [  120.62635591,    39.32842524],
            [   58.91037616,  -29.52086718],
187         [-116.99548555,  -35.64041842],
            [  -49.26778003,   18.11377985],
189         [   91.22017504,    26.95527778],
            [    5.98350205,  -29.65544224],
191         [   73.8606758 ,   -67.33527561],
            [  -57.11269196,  -23.38066312],
193         [   10.29413585,    19.91249178],
            [  -76.57980277,   36.15112039],
195         [   40.91217006,  -17.81387299],
            [   51.88700332,  -69.65988091],
197         [   57.41048001, -119.28130887],
            [  -66.49323658,  -92.43371661],
199         [   10.46455101,  -80.23934518]])

201 dim = 2; number_of_points = 30

203 isManualEnter = False
    isAccelerated  = True
```

```
205
inp = input('Use manual enter or use default parameters? M/D.')
207 if inp == 'M':
    isManualEnter = True
209 if isManualEnter is True:
    gener = input('Use generator or manual input of points? G/M')
211     if gener == 'M':
        print('Enter the data (points values) in the \"data.txt\" file .')
213         points = []
        number_of_points = 0
215         with open("data.txt") as f:
            for line in f:
217                 temp = [float(x) for x in line.split()]
                points.append(temp)
219                 number_of_points += 1
        points = np.array(points)
221         dim = len(points[0])
    elif gener == 'G':
223         dim = int(input('Enter number of dimensions: '))
        number_of_points = int(input('Enter number of points: '))
225         points = GenerPoints(3, 68, number_of_points, dim)

227     temp = input('Use classic or accelerated MDM-method? C/A') #by default
        we're using accelerated method
    if temp == 'C':
229         isAccelerated = False

231 elif isManualEnter is False:
    print('Our DEFAULT values: \nNumber of dimensions is ' \
233         + str(dim) + '\nNumber of points is ' + str(number_of_points))


235
hull = GetHullandPlot(points, dim)
237 mdm = MDM(points, hull, dim, isAccelerated)
result = mdm.solve()                                    #returns a point in R^dim
239
if dim == 2 :
241     plt.plot([result[0], 0], [result[1], 0], 'ro')
elif dim == 3 :
243     plt.plot([result[0], 0], [result[1], 0], [result[2], 0], 'ro')
plt.show()
```

```
245 print('Result is : ' + str(result))
```

Список литературы

[1] Álvaro Barbero, Jorge López Lázaro,José R. Dorronsoro. *An accelerated MDM algorithm for SVM training.* Conference: ESANN 2008, 16th European Symposium on Artificial Neural Networks, Bruges, Belgium, April 23-25, 2008, Proceedings.

[2] В.Н.Малозёмов. *МДМ-методу 40 лет.*