Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ»

Кафедра информатики

Отчет по предмету:

«Машинное обучение»

По лабораторной работе №1

«Линейная регрессия»

Выполнил: Сенькович Дмитрий Сергеевич

магистрант кафедры информатики

группы №858642

Проверил: Стержанов Максим Валерьевич

доцент, кандидат технических наук

Минск 2019

**Оглавление**

[1 Введение 2](#_Toc25600208)

[2 Предсказание прибыли ресторана 3](#_Toc25600209)

[2.1 Постановка задачи 3](#_Toc25600210)

[2.2 Чтение данных 3](#_Toc25600211)

[2.3 График зависимости прибыли ресторана от населения города 3](#_Toc25600212)

[2.4 Реализация функции потерь J(θ) 4](#_Toc25600213)

[2.5 Реализация функции градиентного спуска и построение модели 5](#_Toc25600214)

[2.6 Трехмерный график функции потерь и в виде изолиний 8](#_Toc25600215)

[3 Предсказание цены дома 11](#_Toc25600216)

[3.1 Постановка задачи 11](#_Toc25600217)

[3.2 Чтение данных 11](#_Toc25600218)

[3.3 Нормализация признаков 11](#_Toc25600219)

[3.4 Реализация функции потерь и градиентного спуска с использованием векторизации. Прирост производительности 16](#_Toc25600220)

[3.5 Сравнение различных значений скоростей обучения 19](#_Toc25600221)

[3.6 Реализация аналитического решения. Сравнение с качеством обучения градиентным спуском 20](#_Toc25600222)

# 1 Введение

Линейная регрессия - один из простейших методов машинного обучения с учителем для предсказания некоторого целевого вещественного значения по имеющимся признакам, т.е. регрессии. Несмотря на свою простоту и наглядность, модель может также использоваться для предсказания в случае нелинейных зависимостей путем добавления полиномиальных признаков - фич.

# 2 Предсказание прибыли ресторана

### 2.1 Постановка задачи

Имеется некоторый набор данных, каждый экземпляр которых представляет из себя информацию о некотором ресторане: населения города, в котором он находится, и прибыль, полученная этим рестораном в этом городе. Требуется использовать линейную регрессию в качестве модели предсказания прибыли ресторана.

### 2.2 Чтение данных

Считаем данные из файла ex1data1.txt:

data = [[], []]

with open('ex1data1.txt', 'r') as data\_file:

for line in data\_file:

data\_i = [float(x) for x in line.split(',')]

data[0].append(data\_i[0])

data[1].append(data\_i[1])

В итоге получится двумерный массив: первый массив будет представлять собой значения населения городов, а второй - значения прибыли ресторана.

### 2.3 График зависимости прибыли ресторана от населения города

Построим график зависимости прибыли от населения с помощью библиотеки matplotlib:

plt.plot(data[0], data[1], 'bo')

plt.show()

Получившийся график представлен ниже:

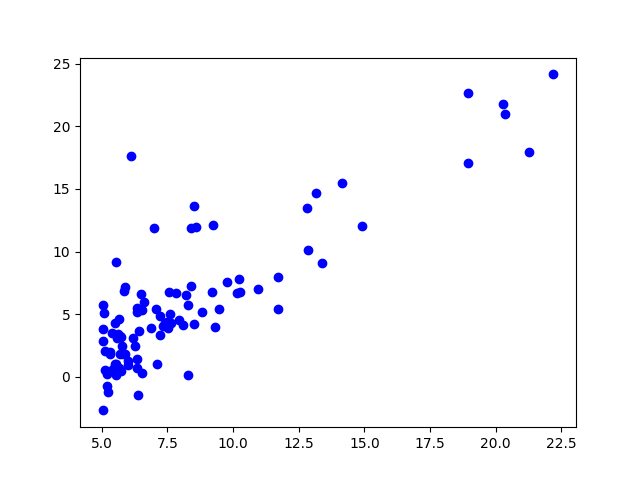


Рисунок 2.3.1. График зависимости прибыли ресторана от населения

### 2.4 Реализация функции потерь J(θ)

Для реализации функции потерь в первую очередь нужно реализовать функцию гипотезы линейной регрессии. Эта функция - наша гипотеза, модель представления зависимости целевой переменной (в нашем случае прибыли ресторана) от фич (населения города). В лабораторной работе используется простейший вид гипотезы для линейной регрессии: прямая. Итак, реализация функции гипотезы:

def h(o, o1, x):

return o + o1\*x

Далее, реализуем функции стоимости (потерь, ошибок). В нашем случае эта функция представляет из себя среднеквадратическую ошибку между реальными значениями прибыли и значениями, предсказанными нами. Формула такой функции ошибки представлена ниже:

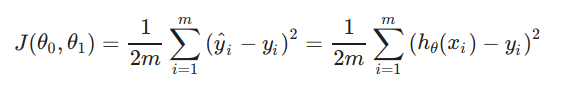


Рисунок 2.4.1. Формула среднеквадратичной функции потерь

Ниже представлена реализации такой функции стоимости:

def j(target, m, o, o1):

res = 0.0

for i in range(m):

x = target[0][i]

y = target[1][i]

res += (h(o, o1, x) - y) \*\* 2

return res / (2\*m)

### 2.5 Реализация функции градиентного спуска и построение модели

Градиентный спуск - алгоритм, уменьшающий стоимость некоторой функции потерь путем изменения параметров гипотезы. Для оптимизации параметров следует взять частные производные по каждой из переменной веса, использующихся в функции потерь, и, взвесив гиперпараметром скорости обучения, обновить веса. Таким образом, мы сможем обеспечить движение алгоритма по поверхности функции стоимости к ее локальному минимуму. В нашем случае, единственный минимум такой функции и является глобальным - искомым.

Формула, соответствующая оптимизации весов гипотези в ходе градиентного спуска, представлена ниже:



Рисунок 2.5.1. Формула оптимизации параметров модели

Реализуем функции частных производных функции стоимости по весам гипотезы:

def j\_o\_der(target, m, o, o1):

res = 0.0

for i in range(m):

x = target[0][i]

y = target[1][i]

res += (h(o, o1, x) - y)

return res / m

def j\_o1\_der(target, m, o, o1):

res = 0.0

for i in range(m):

x = target[0][i]

y = target[1][i]

res += (h(o, o1, x) - y) \* x

return res / m

Далее, реализуем сам алгоритм градиентного спуска. Алгоритм будет итеративно подсчитывать частные производные функции потерь по весам модели, обновлять веса модели в соответствии с формулой выше и закончит свою работу, когда некоторая минимальная ошибка, удовлетворяющая нас, будет достигнута. Еще один критерий остановы такого алгоритма - ограничение на количество итераций - будет использоваться в последующих лабораторных работах. Итак, функция градиентного спуска:

def gd(target):

o = 0.0

o1 = 0.0

error = 1000000

its = 0

m = len(target[0])

print('m %s' % m)

while error > max\_error:

t\_o = o - alpha \* j\_o\_der(target, m, o, o1)

t\_o1 = o1 - alpha \* j\_o1\_der(target, m, o, o1)

o = t\_o

o1 = t\_o1

error = j(target, m, o, o1)

its += 1

print('its %s' % its)

print('error %s' % error)

return o, o1, its

Здесь alpha - скорость обучения, взята 0.001; max\_error - ошибка, при достижении которой обучение закончится, взята 4.477. Такой подход выбора остановы алгоритма плох тем, что значение ошибки приходится подбирать эмпирически, т.е. путем многократного запуска алгоритма.

Запустим алгоритм:

o, o1, its = gd(data)

print('its %s' % its)

print('o %s' % o)

print('o1 %s' % o1)

График получившейся модели и исходных данных будет построен будет изображения прямой - модели - на отрезке [0; 25] с использованием 1000 точек, и самих данных:

x = np.linspace(0, 25, 1000)

y = o + o1\*x

plt.plot(data[0], data[1], 'bo')

plt.plot(x, y, '-r')

plt.show()

Получившийся график представлен ниже:

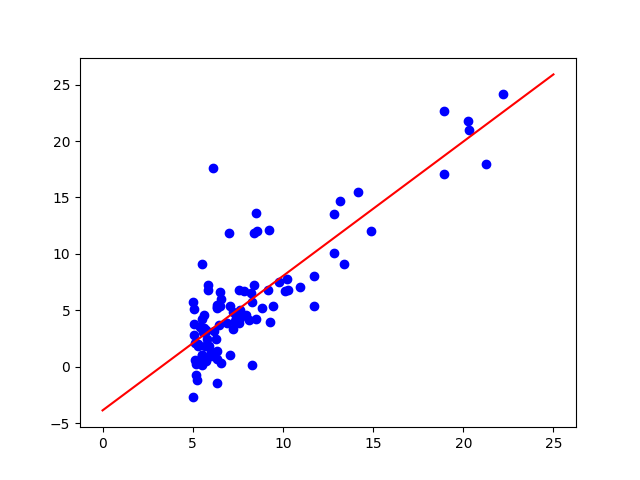


Рисунок 2.5.2. График модели и исходных данных

### 2.6 Трехмерный график функции потерь и в виде изолиний

Построим трехмерный график функции стоимости в зависимости следующим образом: возьмем значения x и y - фич и целевого значения - в промежутке от -4 до 4 с шагом 0.01. Для получившейся сетки посчитаем и покажем на графике соответствующее значение функции потерь:

fig = plt.figure()

ax = plt.axes(projection="3d")

X = np.arange(-4, 4, 0.01)

Y = np.arange(-4, 4, 0.01)

X, Y = np.meshgrid(X, Y)

m = len(data[0])

points\_count = len(X)

Z = np.array([j(data, m, X[i], Y[i]) for i in range(points\_count)])

ax.plot\_wireframe(X, Y, Z, color='green')

plt.show()

Ниже представлен трехмерный график функции потерь, полученный способом выше:

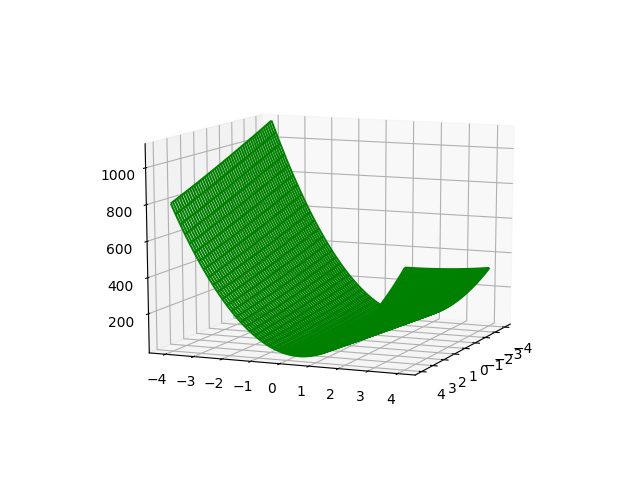


Рисунок 2.6.1. Трехмерный график функции потерь

Для большей наглядности построим график изолиний функции потерь. Такой график представляет из себя проекцию 3d пространства на 2d, показывая в цвете с подписью линии, соответствующие различной “высоте” функции в 3d, то есть в нашем случае функции потерь. Такой способ более приемлем для визуализации 3d графиков, так как более быстр в построении и является более наглядным:

fig, ax = plt.subplots()

X = np.arange(-4, -3.5, 0.01)

Y = np.arange(1, 1.5, 0.01)

X, Y = np.meshgrid(X, Y)

m = len(data[0])

points\_count = len(X)

Z = np.array([j(data, m, X[i], Y[i]) for i in range(points\_count)])

CS = ax.contour(X, Y, Z)

ax.clabel(CS, inline=1, fontsize=10)

plt.show()

Ниже представлен получившийся график:

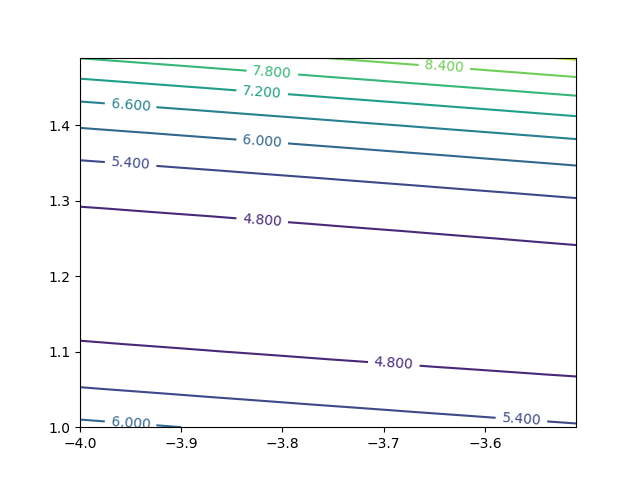


Рисунок 2.6.2. График функции потерь в виде изолиний

На графике в виде линий представлены разные значения функции потерь. Разные цвета показывают разные значения функции. Чем ярче цвет, тем большее значение принимает функция потерь на данной линии. Линии представляют из себя, в нашем случае, кольца одинаковой “высоты”, т.е. линиям соответствуют одинаковые значения функции потерь в координатах (x, y) в этих линиях. Таким образом, можно видеть, что наша ошибка - 4.477 - довольно близка к оптимальному значению.

# 3 Предсказание цены дома

### 3.1 Постановка задачи

В этой части лабораторной работы дана информация о стоимости дома и соответствующих этому дому его площади и количеству комнат в нем. В работе будет не только построена модель линейной регрессии, но и будет рассмотрен вопрос нормализации признаков, векторизации вычислений, сравнение различных значений скорости обучения, аналитическое решение.

### 3.2 Чтение данных

Считаем данные из файла ex1data2.txt:

x = []

y = []

with open('ex1data2.txt', 'r') as data\_file:

for line in data\_file:

data\_i = [float(x) for x in line.split(',')]

x.append([1, data\_i[0], data\_i[1]])

y.append(data\_i[2])

В итоге получатся два массива: x - содержащий фичи модели и y - целевые значения.

### 3.3 Нормализация признаков

Один из способов ускорения схождения градиентного спуска: нормализация признаков. Нормализация представляет из себя, как правило, вычитание из каждой фичи смещения (среднего значения значений фичи) - mean normalization и деления получившего значения, например, на разницу между наибольшим и наименьшим значением фичи в выборке или разброс (среднеквадратичное отклонение) - feature scaling. Вот так выглядит процесс схождения градиентного спуска в случае использования денормализованных данных (слева) и нормализованных (справа):

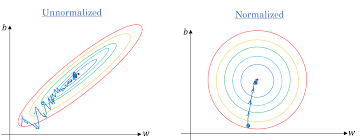


Рисунок 3.3.1. Примеры схождения градиентного спуска без и с нормализацией

Реализуем нормализацию для случая обычной (без использования векторизации) реализации градиентного спуска:

def normalize\_x(x):

dim = len(x[0])

m = len(x)

for dimi in range(1, dim):

sum\_dimi = 0.0

min\_dimi = float('inf')

max\_dimi = float('-inf')

for i in range(m):

sum\_dimi += x[i][dimi]

if x[i][dimi] < min\_dimi:

min\_dimi = x[i][dimi]

if x[i][dimi] > max\_dimi:

max\_dimi = x[i][dimi]

mean = sum\_dimi / m

for i in range(m):

x[i][dimi] = (x[i][dimi] - mean) / (max\_dimi - min\_dimi)

Здесь проведена mean normalization и feature scaling с использованием более простой стратегии: деления на разницу наибольшего и наименьшего значений фичи.

Для проверки разницы скорости сходимости в случае реализации с использованием нормализации и без для начала реализуем сам алгоритм градиентного спуска:

def h(o, x):

res = 0.0

for i in range(len(x)):

res += o[i] \* x[i]

return res

def j(x, y, m, o):

res = 0.0

for i in range(m):

xi = x[i]

yi = y[i]

res += (h(o, xi) - yi) \*\* 2

return res / (2 \* m)

def j\_oi\_der(x, y, m, o, feature\_i):

res = 0.0

for i in range(m):

xi = x[i]

yi = y[i]

res += (h(o, xi) - yi) \* xi[feature\_i]

return res / m

def update\_o(x, y, m, o):

o\_len = len(o)

new\_o = [0] \* o\_len

for i in range(o\_len):

j\_oi\_der\_val = j\_oi\_der(x, y, m, o, i)

new\_o[i] = o[i] - alpha \* j\_oi\_der\_val

return new\_o

Реализация градиентного спуска, по сути, такая же, как и в предыдущей задачи лабораторной работы с небольшим изменением с учетом большего количества фич.

Запустим градиентный спуск без нормализации:

start = timer()

o, its, its\_hist, err\_hist = gd(x, y)

end = timer()

print(timedelta(seconds=end-start))

print('its %s' % its)

print('o %s' % o)

Результатом будет следующая ошибка:

Traceback (most recent call last):

File "/media/dmitry/FA9ABE859ABE3E47/ml\_labs/lab1/2/lab.py", line 176, in <module>

o, its, its\_hist, err\_hist = gd(x, y)

File "/media/dmitry/FA9ABE859ABE3E47/ml\_labs/lab1/2/lab.py", line 82, in gd

error = j(x, y, m, o)

File "/media/dmitry/FA9ABE859ABE3E47/ml\_labs/lab1/2/lab.py", line 24, in j

res += (h(o, xi) - yi) \*\* 2

OverflowError: (34, 'Numerical result out of range')

Это означает, что в ходе алгоритма среднеквадратическая ошибка достигла значения, которые не “помещается” в диапазон вещественных значений переменной в Python 3. Стоит заметить, что такие случаи аномальны, ведь значения вещественной переменной в Python 3 на машине, исполнявшей алгоритм, следующие:

>>> import sys

>>> sys.float\_info

sys.float\_info(max=1.7976931348623157e+308, max\_exp=1024, max\_10\_exp=308, min=2.2250738585072014e-308, min\_exp=-1021, min\_10\_exp=-307, dig=15, mant\_dig=53, epsilon=2.220446049250313e-16, radix=2, rounds=1)

Такая проблема решается с помощью использования класса decimal.Decimal, что, впрочем, является неприемлемым решением в таких случаях ввиду значительного снижения скорости расчетов.

Чтобы посмотреть, как поведет себя алгоритм с нормализацией, запустим предыдущий код с нормализацией вначале:

normalize\_x(x)

В этом случае алгоритм со скоростью обучения 0.001 достигает значения ошибки 2043380050 через 32-34 секунды приблизительно за 261.000 итерацию.

График ошибки в зависимости от количества итераций представлен ниже:

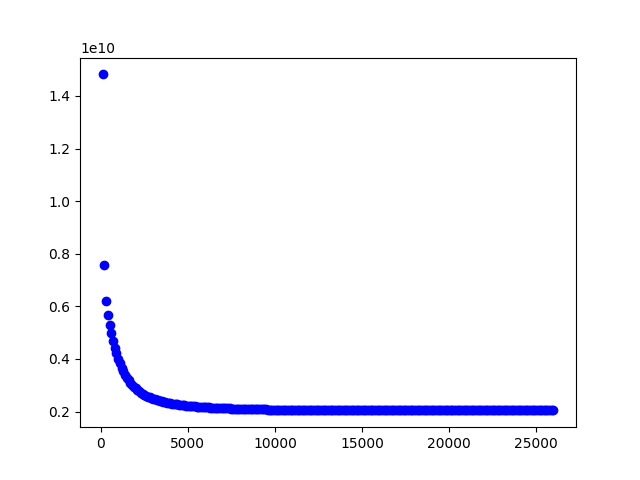


Рисунок 3.3.2. График зависимости ошибки от количества итераций в случае реализации градиентного спуска с использованием нормализации

### 3.4 Реализация функции потерь и градиентного спуска с использованием векторизации. Прирост производительности

Еще один способ ускорить схождение и расчеты алгоритма - векторизация. Особенно сильно видна разница при большой количестве данных и фич. Также, векторизация потенциально дает возможность к распараллеливанию вычислений на нескольких ядрах одного процессора или даже нескольких компьютеров.

Ниже представлены несколько формулы для векторизации реализации алгоритма, представленного выше:

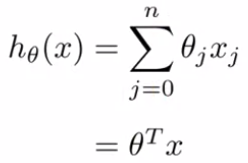


Рисунок 3.4.1. Формула векторизации вычислений гипотезы для одного экземпляра выборки

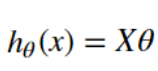


Рисунок 3.4.2. Формула векторизации вычислений гипотезы для всей выборки

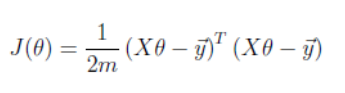


Рисунок 3.4.3. Формула векторизации вычислений функции потерь

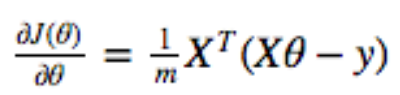


Рисунок 3.4.4. Формула векторизации вычислений функции потерь

Приведем ниже листинг кода реализации алгоритма с векторизацией:

def h\_vect(ot, x):

return x \* ot

def j\_vect(x, yt, m, ot):

h\_vec\_val = h\_vect(ot, x)

hy\_diff = h\_vec\_val - yt

return (hy\_diff.transpose() \* hy\_diff)[0].item(0) / (2 \* m)

def j\_oi\_der\_vect(x, xt, y, yt, m, ot):

h\_vec\_val = h\_vect(ot, x)

return (xt \* (h\_vec\_val - yt)) / m

def update\_o\_vect(x, xt, y, yt, m, ot):

j\_oi\_der\_val = j\_oi\_der\_vect(x, xt, y, yt, m, ot)

return ot - alpha \* j\_oi\_der\_val

def gd\_vect(x, y):

o = np.matrix([0.0, 0.0, 0.0])

ot = o.transpose()

error = 1000000000000000000

its = 0

m = len(x)

print('m %s' % m)

its\_hist = []

err\_hist = []

xt = x.transpose()

yt = y.transpose()

while error > max\_error:

ot = update\_o\_vect(x, xt, y, yt, m, ot)

error = j\_vect(x, yt, m, ot)

its += 1

if (its % 100 == 0):

its\_hist.append(its)

err\_hist.append(error)

if (its % 10000 == 0):

print(error)

print('its %s' % its)

print('error %s' % error)

return ot, its, its\_hist, err\_hist

Покажем, что векторизация результирует в приросте производительности, для чего запустим градиентный спуск с векторизацией:

x\_np = np.matrix(x)

y\_np = np.matrix(y)

normalize\_x\_np(x\_np)

start = timer()

o, its, its\_hist, err\_hist = gd\_vect(x\_np, y\_np)

end = timer()

print(timedelta(seconds=end-start))

print('its %s' % its)

print('o %s' % o)

В этом случае время работы алгоритма при тех же условиях будет занимать порядка 25 секунд. Считая время работы алгоритма без векторизации равной 33 секунды, получаем, что удалось ускорить алгоритм на 24%.

### 3.5 Сравнение различных значений скоростей обучения

Сравним графики со значениями alpha, равными 0.001 и 0.01. Оставим при этом приемлемую ошибку такой же:

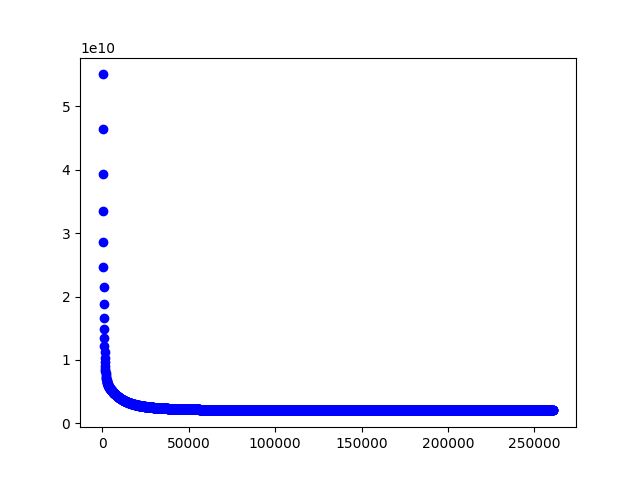


Рисунок 3.5.1. Зависимость ошибки от количества итераций при alpha = 0.001

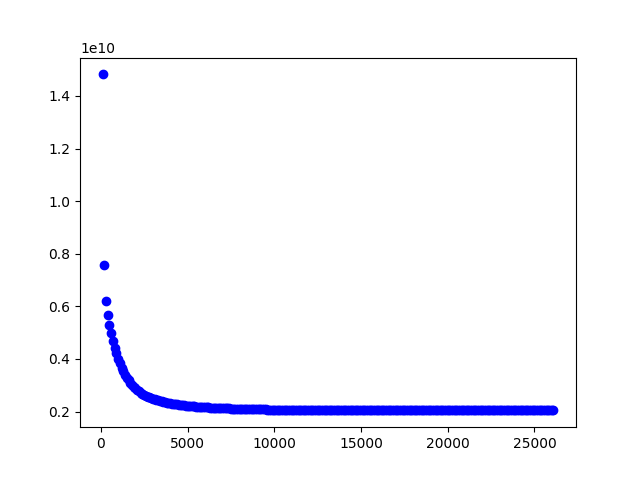


Рисунок 3.5.2. Зависимость ошибки от количества итераций при alpha = 0.01

Ввиду того, что скорость обучения во второй варианте в 10 раз быстрее, чем в первом, получаем более быструю сходимость, а именно в 10 раз: алгоритму нужно порядка 260.000 итераций в первом случае и около 26.000 во втором.

### 3.6 Реализация аналитического решения. Сравнение с качеством обучения градиентным спуском

Для линейной регрессии в случае только линейных фич может быть найдено аналитическое решение по следующей формуле:

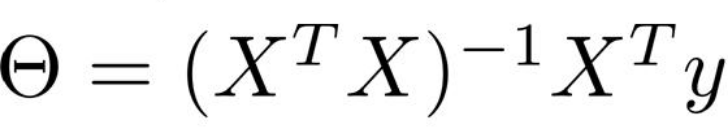


Рисунок 3.6.1. Нахождение оптимальных параметров модели методом наименьших квадратов

Преимущества метода очевидны: не нужно подбирать скорость обучения, количество итераций. Однако в реальности данных метод не используется, потому что он очень медленный, потому что сложность вычисления обратной матрицы составляет O(n^3), где n -количество фич, в то время как сложность градиентного спуска O(n^2).

Реализуем аналитическое решение задачи:

x\_np = np.matrix(x)

y\_np = np.matrix(y)

normalize\_x\_np(x\_np)

o = np.linalg.inv(x\_np.transpose() \* x\_np) \* (x\_np.transpose()) \* y\_np.transpose()

print(o)

Скорость работы алгоритма: 0:00:00.000155 секунд, что несравнимо мало даже с версией градиентного спуска с векторизацией - 25 секунд.

Сравним решения в обоих случаях. Первые значения: решение через градиентный спуск, вторые: через аналитическое решение:

340412.65957443905, 502710.3587624075, -32294.8374541408

340412.65957447, 504777.90398791, -34952.07644931

Как видим, значения параметров отличаются незначительно.