1 Одномерная линейная регрессия

В этой части работы используется линейная регрессия с одной переменной для предсказания прибыли, полученной в результате торговли в разных городах. Предположим, что нужно рассмотреть некоторые города для открытия новой торговой точки.

Имеются данные о прибыли, которую можно получить в этих городах, а также численность населения этих городов. Нужно использовать эти данные, чтобы выбрать, какой город использовать следующим.

Справка

Регрессионный анализ – статистический метод исследования влияния одной или нескольких независимых переменных на зависимую переменную.

Линейная регрессия – используемая в статистике регрессионная модель с линейной функцией зависимости (рис. 1).

В простейшем случае линейную модель можно представить так:

$$y_i = w_0 + w_1 x_i + \varepsilon_i,$$

где w_0 – математическое ожидание зависимой переменной y_i, когда переменная x_i равна нулю;

w_1 – ожидаемое изменение зависимой переменной y_i при изменении x_i на единицу;

ε_i – случайная ошибка.

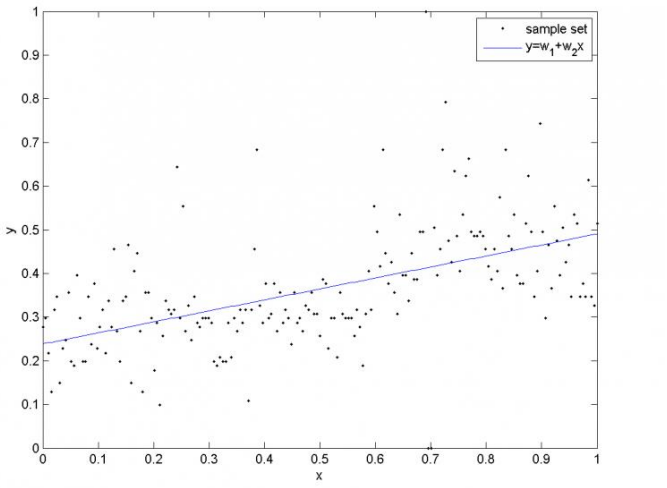

Рисунок 1 – Пример графика одномерной линейной регрессии

Файл **ex1data1.txt** содержит набор данных для нашей задачи.

Первый столбец – численность населения города (в 10 000 чел.), второй столбец – прибыль, которую можно получить в этом городе (в 10 000-кратном размере). Отрицательное значение для прибыли свидетельствует об убыточности торговой точки.

Задание 1. Напишите скрипт для загрузки этих данных в Python. Для этого рекомендуется создать текстовый файл с расширением **.py**, в котором вы будете записывать команды скрипта. Файлы скриптов Python должны располагаться в файлах именно с таким расширением.

При написании скрипта полезно бывает вставлять комментарии в код, поясняющие принцип работы скрипта. В Python для обозначения однострочного комментария используются символ #. Например

```
# пример создания матрицы NumPy в Python
a = np.array([[1, 2, 3], [4, 5, 6]])
```

Для загрузки данных из файла используется команда **loadtxt** из библиотеки **numpy**, в которую передаётся имя файла и параметр **delimiter** (разделитель значений в текстовом файле). В данной лабораторной работе в качестве файла с данными будет использоваться обычный текстовый файл с запятыми и переносами строк в качестве разделителей.

Если в качестве файла с данными используется обычный текстовый файл с разделителями, то возвращаемое функцией **loadtxt** значение необходимо присвоить некоторой переменной. Так как в файле хранится матрица значений, используем функцию **matrix** из **numpy**.

```
# в этом случае данные из файла будут сохранены в переменной data.
data = np.matrix(np.loadtxt('ex1data1.txt', delimiter=','))
```

Для того, чтобы загрузить данные из файла, может быть полезным перейти в директорию, содержащую данный файл. Для этого используется функция **chdir** модуля **os**. В качестве параметра передаётся абсолютный или относительный путь к нужной директории:

```
>>> import os
>>> os.chdir('D:')
```

Для проверки текущей рабочей директории используется команда **getcwd** модуля **os**:

```
>>> os.getcwd()
'D:\\'
```

Задание 2. Нарисуйте данные, загруженные в задании 1 с помощью функции **plot** библиотеки **matplotlib**. Программно подпишите оси. Так как в нашей задаче всего 1 переменная, можно изобразить их в виде точечного графика на плоскости.

Для построения графиков в Python используется мощная библиотека **matplotlib**. Чтобы использовать её в своём скрипте, необходимо подключить её командой

```
import matplotlib.pyplot as plt
```

Для построения простых графиков проще всего использовать функцию **plot**. В качестве параметров ей передаётся либо 1 вектор (массив значений), либо 2, тогда первый будет содержать значения по оси абсцисс, а второй – значения по оси ординат. В качестве третьего параметра можно указать, как именно будет выглядеть график. Третий параметр представляет собой строку из 2 символов. Первый символ отвечает за цвет, например, **r (red)**, **g (green)**, **b (blue)**, **k (black)**. Второй символ отвечает за вид графика: **– (непрерывная линия)**, **o (кружочки)**, **. (точки)**, *** (звёздочки)**. Примеры: **'ro'** – график в виде красных кружочков, **'b-'** – график в виде непрерывной синей линии, **'b.'** – график в виде чёрных точек.

Важно помнить, что если в функцию **plot** передаются данные по оси абсцисс и по оси ординат, то размер этих данных должен быть одинаковым, то есть, например, это должны быть вектора одинаковой длины.

Чтобы подписать оси и график, используются функции **xlabel**, **ylabel** и **title** соответственно. В качестве параметров все функции принимают строку. Для рисования сетки используется функция **grid**.

Пример построения точечного графика. Значения будут отмечены синими точками:

```
import matplotlib.pyplot as plt

from matplotlib import rc
font = {'family': 'Verdana', 'weight': 'normal'}
rc('font', **font)

plt.plot(X, y, 'b.')
plt.title('Зависимость прибыльности от численности')
plt.xlabel('Численность')
plt.ylabel('Прибыльность')
plt.grid()
plt.show()
```

Обратите внимание на строки 2-4. Этот код необходим для того, чтобы на графике корректно отображались русские символы.

Итоговый график показан на рисунке 2.

Рисунок 2 – График зависимости прибыльности от численности города

Чтобы нарисовать такой график, вам потребуется выбрать значения из данных, которые вы загрузили из текстового файла в предыдущем задании. Первый столбец данных, отвечающий за численность города, необходимо сохранить в переменной **X**, второй столбец, отвечающий за прибыль, в переменной **y**.

В библиотеке NumPy присутствуют богатые возможности по выбору строк, столбцов и отдельных элементов из матриц:

```
>>> a = np.matrix([[1, 2], [3, 4], [5, 6]])
>>> a[0, 0] # выбрать первый элемент в матрице
1
```

Если в качестве одного из индексов указано двоеточие (**:**), значит подходят элементы с любым индексом. Так можно выбирать из матрицы целые строки или столбцы:

```
a[:, 1]  # выбираем второй столбец (индекс строки – любой, индекс столбца – второй. Индексы начинаются с нуля)
a[2, :]  # выбираем третью строку (индекс строки – третья, индекс столбца – любой)
```

Более того, в качестве индекса можно использовать массивы:

```
a[[0, 2], :]  # берём из матрицы a первую и третью строки
```

Так же можно выбирать диапазоны значений:

```
a[0:2, 3:6]  # из матрицы a будет выбрана подматрица с 1 по 3 строку и с 4 по 7 столбец
```

Используя полученные знания, выберите первый столбец из данных файла в переменную **X**, а второй столбец в переменную **y**.

Функция стоимости (ошибка регрессии)

Для проведения градиентного спуска (алгоритма определения оптимальных коэффициентов, описывающих линию регрессии) нужно определить функцию стоимости (оценки), в качестве неё будем использовать квадратичную ошибку, формула 1:

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2, \quad (1)$$

где гипотеза $h_\theta(x)$ получена из линейной модели, формула 2:

$$h_\theta(x) = \theta^T x = \theta_0 + \theta_1 x_1. \quad (2)$$

Формулу выше можно переписать следующим образом (формула 3):

$$h_\theta(x) = \theta_0 x_0 + \theta_1 x_1. \quad (3)$$

Где x_0 вводят искусственно, и этот параметр всегда равен единице. Этот приём используют для того, чтобы можно было вычислить значение гипотезы $h_\theta(x)$ в векторной форме (через перемножение векторов). Этот приём называется векторизацией вычислений и даёт помимо уменьшения кода по вычислению $h_\theta(x)$ ещё и значительное увеличение быстродействия расчётов, так как во многих языках программирования (и в библиотеке NumPy в том числе) векторные и матричные операции хорошо оптимизированы и выполняются быстрее, чем написанные вручную циклы.

Для понимания процедуры векторизации рассмотрим пример. Допустим, требуется вычислить значение гипотезы $h_\theta(x)$ для коэффициентов θ 1 и 2 и для значения x равного 3. По формуле 2 вычислим значение $h_\theta(x)$:1+2*3=7.

Теперь выполним то же самое, но с векторизацией вычислений. Опишем коэффициенты θ в виде вектора-столбца:

$$\theta = \begin{bmatrix}1\\2\end{bmatrix}.$$

Значение x можно тоже записать в виде вектора-столбца. Для того, чтобы можно было применять векторизацию вычислений, к x добавим значение $x_0 = 1$. Тогда вектор x будет выглядеть так:

$$x = \begin{bmatrix}1\\3\end{bmatrix}.$$

Тогда значение гипотезы $h_\theta(x)$ может быть вычислено по формуле 2 в векторной форме:

$$h_\theta(x) = \theta^T x = \begin{bmatrix}1 & 2\end{bmatrix} \cdot \begin{bmatrix}1\\3\end{bmatrix} = 1 \cdot 1 + 2 \cdot 3 = 7.$$

Как видим, в векторной форме мы получили тот же результат, что и при поэлементном перемножении. Важно помнить, что для векторизации вычислений необходимо добавлять единичное значение $x_0 = 1$.

В библиотеке NumPy для транспонирования матрицы или вектора используется метод **transpose()** (или краткий вариант записи функции: **T**):

```
>>> theta = np.matrix([1, 2])
>>> theta
matrix([[1, 2]])
>>> theta.transpose()
matrix([[1],
        [2]])
>>> theta.T
matrix([[1],
        [2]])
```

Следует добавить ещё одно важное замечание. Мы считали значение гипотезы для отдельного значения x, которое представляло собой вектор-столбец. Но в нашей задаче существует целый набор значений x. И к этому столбцу значений нужно присоединить в начале столбец единиц, по размеру совпадающий с размером x. Это можно сделать с помощью следующего кода Python:

```
m = X.shape[0]  # количество элементов в X (количество городов)
X_ones = np.c_[np.ones((m, 1)), X]  # добавляем единичный столбец к X
theta = np.matrix('[1; 2]')  # коэффициенты theta представляют собой вектор-столбец из 2 элементов
# альтернативные способы создания вектора-столбца theta:
# theta = np.matrix([1, 2]).reshape(2, 1)
# theta = np.matrix([[1], [2]])
# theta = np.matrix([1, 2]).transpose()
h_x = X_ones * theta  # так можно вычислить значение гипотезы для всех городов сразу (подумайте почему и поэкспериментируйте с этим кодом в консоли Python).
```

Обратите внимание на 3 строку приведённого выше кода. Элементы вектора **theta** задаются в виде строки в одинарных кавычках. Такая запись позволяет создавать матрицы, используя более компактный синтаксис языка MatLab.

Задание 3. Реализуете функцию **computeCost** для вычисления $J(\theta)$ для заданных коэффициентов θ_j и входных данных X и y.

Проверьте её работу, инициализировав θ_j нулевыми значениями.

Объявление функции Python начинается с ключевого слова **def**. Далее идёт имя функции и в скобках список её параметров. После закрывающей скобки ставится двоеточие.

Вот как должен выглядеть прототип создаваемой вами функции для вычисления ошибки:

```
def compute_cost(X, y, theta):
  # здесь будет размещаться ваш код вычисления ошибки cost
  return cost
```

Функция принимает 3 параметра:

1) **X** – матрица, содержащая **m** строк и **2** столбца. **m** соответствует числу примеров (городов, для которых известна прибыль). Первый столбец в **X** состоит

из единиц. Второй столбец соответствует численности каждого города. Единичный столбец в данном случае необходим для того, чтобы можно было использовать векторизацию вычислений.

2) **y** – прибыльность каждого города. **y** представляет собой вектор-столбец, состоящую из **m** элементов (по числу городов, для которых известна прибыль). **y** содержит значения прибыли в каждом городе.

3) **theta** – вектор-столбец коэффициентов θ. Всего в **theta** 2 элемента, которые полностью определяют форму регрессионной прямой. Первый отвечает за смещение прямой относительно начала координат, а второй за наклон прямой (по сути, второй коэффициент **theta** – это тангенс угла наклона прямой к оси абсцисс).

После того, как в функции **compute_cost** Вы реализуете вычисление гипотезы **h_x** для всех городов **X** в векторной форме, необходимо будет вычислить значение стоимости $J(\theta)$ по формуле 1. Для этого из каждого значения **h_x** нужно будет вычесть соответствующее значение **y**, разности возвести в квадрат, просуммировать и поделить на удвоенное число примеров (городов). Например, возведение в квадрат разностей двух векторов может выглядеть так:

```
np.power(h_x - y, 2)
```

Для суммирования элементов в векторе используется функция **sum** из библиотеки **NumPy**. Для определения количества примеров в матрице **X** используйте свойство **shape**.

Используя данную справочную информацию, реализуйте функцию **compute_cost**, которая будет вычислять ошибку между реальными данными **y** и предсказанными с помощью гипотезы **h_x** значениями по формуле 1.

Для проверки правильности реализованной вами функции вызовите её с параметрами **X_ones** (это столбец **X**, содержащий значения численности городов, к которому добавлен единичный столбец), **y** (прибыльность каждого города) и коэффициентами **theta = [1, 2]**

Если функция реализована правильно, вы должны получить значение **75.203**.

Градиентный спуск

Необходимо подобрать такой набор коэффициентов θ_j, чтобы значение $J(\theta)$ было минимальным. Один из способов сделать это – использование алгоритма градиентного спуска. Каждая итерация алгоритма обновляет параметры модели, формула 4:

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_j^{(i)}, \qquad (4)$$

где α – скорость обучения ($0 < \alpha < 1$).

С каждым шагом градиентного спуска коэффициенты θ_j приближаются к оптимальным значениям, которые позволят обеспечить наименьшую стоимость $J(\theta)$.

При выполнении градиентного спуска нужно контролировать сходимость путем вычисления общей стоимости.

Далее необходимо реализовать функцию градиентного спуска **gradient_descent**.

Помните, что стоимость $J(\theta)$ параметризуется вектором θ, а не X и y. То есть, мы минимизируем значение $J(\theta)$, изменяя значения вектора θ, не изменяя X или y. Хороший способ проверить, что градиентный спуск работает правильно, это посмотреть на значение $J(\theta)$ и убедиться, что оно уменьшается с каждым шагом, Значение $J(\theta)$ никогда не должны расти, и должны сходиться к постоянному значению к концу алгоритма.

В нормальном случае значения $J(\theta)$ сходятся как показано на рисунке 3.

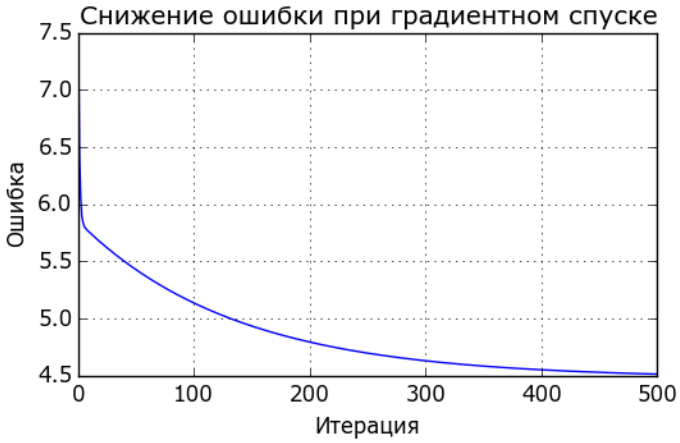

Рисунок 3 – Снижение уровня ошибки с каждой итерацией

В общем виде алгоритм градиентного спуска выглядит так:

1. инициализировать θ = (0 , 1 , … 1);
2. вычислить значение $h_\theta(x)$ для всех входов;
3. вычислить значение функции стоимости $J(\theta)$;
4. если значение функции стоимости $J(\theta)$ изменилось меньше нужного порога, то градиентный спуск завершен, перейти на п. 7, иначе перейти на следующий пункт;
5. изменить значения θ по формуле 3;
6. перейти на п. 2;
7. конец работы.

Для того чтобы избежать зацикливания, нужно сделать ограничение на максимальное количество итераций.

Если $J(\theta)$ растёт, то нужно подобрать другую скорость обучения α и попробовать другие начальные θ. Чем больше α, тем быстрее происходит сходимость, но можно пропустить оптимальную точку.

Задание 4. Реализуйте описанную выше функцию градиентного спуска **gradient_descent**. Входные параметры – *X*, *Y*, скорость обучения α (например, α = 0.01), количество итераций. Не забудьте добавить к X единичный столбец (θ_0 – свободный коэффициент).

Функция **gradient_descent** должна иметь следующий вид:

```
def gradient_descent(X, y, alpha, iterations):
  # определение m – количество городов в матрице X (матрица X должна содержать единичный столбец). Используйте свойство X.shape для определения m и n.
  # определение n – количество столбцов в матрице X (в нашей задаче n = 2)
  # создать вектор-столбец theta и инициализировать его первый элемент нулем, а остальные элементы единицами. Количество элементов в theta равно n.
  # создать вектор-столбец J_theta, размер которого равен количеству итераций, заполненный нулями
  # запустить цикл по итерациям: for i in range(iterations):
    # вычислить значение ошибки J_theta[i] для текущих значений theta
    # модифицировать коэффициенты theta по формуле 4 (не забудьте про использование временной переменной temp_theta)
```

Важное замечание по модификации коэффициентов theta. В формуле 4 коэффициенты θ меняются, но в то же самое время они и используются при вычислении гипотезы $h_\theta(x)$. Чтобы избежать этой ошибочной ситуации, исходные коэффициенты **theta** копируют во временную переменную **temp_theta**, которая будет меняться на каждой итерации цикла по формуле 4. Исходные коэффициенты **theta** при этом не изменяются. После того, как все коэффициенты **temp_theta** модифицированы, их значения присваивают исходным коэффициентам **theta**.

Запустите написанную функцию **gradient_descent**, передав в неё параметры **X_ones**, **y**, **alpha=0.02** и количество итераций **500** и проверьте, какие значения **theta** возвращает функция. Если всё реализовано верно, должны получиться значения около **-3.277, 1.131**.

С помощью функции **plot** нарисуйте график уменьшения ошибки **J_theta** с увеличением итераций. Программно подпишите график и оси. График должен выглядеть примерно как на рисунке 3.

Задание 5. Сделайте предсказание для двух неизвестных ранее городов, используя полученные выше коэффициенты θ_j.

Задание 6. Добавьте к изображению из задания 2 полученную линию регрессии (для рисования на существующем графике дважды вызывайте функцию **plot**). Для рисования линии регрессии используйте следующий код:

```
x = np.arange(min(X), max(X))
plt.plot(x, theta[1]*x + theta[0], 'g--')
```

Не забудьте программно подписать график и оси и отобразить сетку.

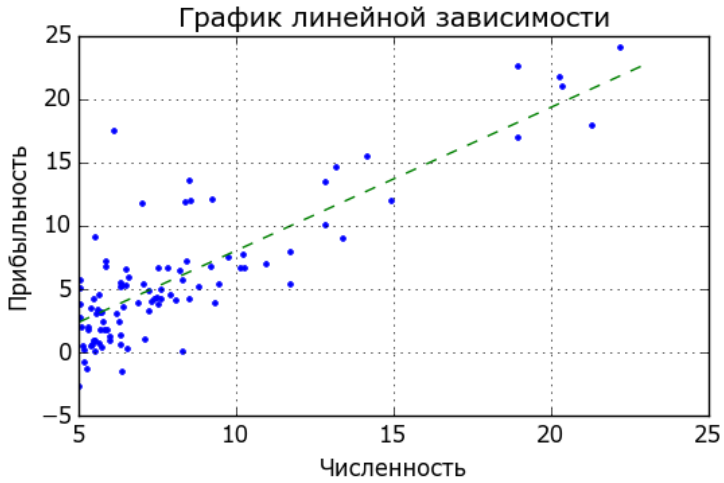

Рисунок 4 – Итоговый график зависимости прибыли от числа жителей

Вид итогового графика приведён на рисунке 4.

2 Многомерная линейная регрессия

В этой части работы Вы реализуете линейную регрессию с несколькими переменными для прогнозирования цены дома. Предположим, Вы продаете свой дом, и вы хотите знать, какая у него рыночная цена. Один из способов сделать это – сначала собрать информацию о последних проданных домах и построить модель цен.

Файл **ex1data2.txt** содержит обучающий набор цен на жилье. Первый столбец – площадь в футах, второй столбец – количество комнат, и третий столбец цена дома.

Обратите внимание, что размеры дома в абсолютных значениях на несколько порядков больше, чем количество комнат. Нормализация может ускорить градиентный спуск.

Задание 7. Загрузите входные данные. Для каждого входного столбца проведите нормализацию: вычислите среднее значение и вычтите его из набора данных. Посчитайте стандартное отклонение и разделите на него каждый элемент. Не забудьте сохранять полученные средние значения и стандартные отклонения. Они понадобятся чтобы восстановить модель.

Задание 8. Постройте модель многомерной линейной регрессии. Сделайте прогноз для двух неизвестных ранее квартир. Можно использовать ваши предыдущие реализации функций **compute_cost** и **gradient_descent**, если они работают с произвольной размерностью. Иначе сделайте их универсальными.

3 Метод наименьших квадратов

Согласно методу наименьших квадратов искомые коэффициенты θ_j могут быть найдены из формулы 4:

$$\theta = (X^T X)^{-1} X^T \vec{y}.$$

Используемая формула не требует каких-либо нормализаций данных и многократных итераций. Однако в некоторых случаях решение не может быть получено, так как $X^T X$ может оказаться необратимой (вырожденной) матрицей (это может получиться если входные столбцы линейно зависимы).

Задание 9. Постройте модель многомерной линейной регрессии с использованием МНК. Сделайте прогноз для двух неизвестных ранее квартир.

При реализации не забудьте добавить к **X** первый единичный столбец. Для транспонирования матрицы используйте её метод **transpose()** (или краткий вариант: **T**). Для вычисления обратной матрицы используйте функцию **np.linalg.pinv** библиотеки **NumPy**.

Задание 10. Сравните модели, полученные методом МНК и методом градиентного спуска.