Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

**ЛАБОРАТОРНАЯ РАБОТА №1**

«Линейная регрессия»

Студент А. Ю. Омельчук

Преподаватель М. В. Стержанов

Минск 2019

## ХОД РАБОТЫ

**Задание.**

Набор данных ex1data1.txt представляет собой текстовый файл, содержащий информацию о населении городов (первое число в строке) и прибыли ресторана, достигнутой в этом городе (второе число в строке). Отрицательное значение прибыли означает, что в данном городе ресторан терпит убытки.

Набор данных ex1data2.txt представляет собой текстовый файл, содержащий информацию о площади дома в квадратных футах (первое число в строке), количестве комнат в доме (второе число в строке) и стоимости дома (третье число).

1. Загрузите набор данных ex1data1.txt из текстового файла.
2. Постройте график зависимости прибыли ресторана от населения города, в котором он расположен.
3. Реализуйте функцию потерь $J(\theta)$ для набора данных ex1data1.txt.
4. Реализуйте функцию градиентного спуска для выбора параметров модели. Постройте полученную модель (функцию) совместно с графиком из пункта 2.
5. Постройте трехмерный график зависимости функции потерь от параметров модели ($\theta 0$ и $\theta 1$) как в виде поверхности, так и в виде изолиний (contour plot).
6. Загрузите набор данных ex1data2.txt из текстового файла.
7. Произведите нормализацию признаков. Повлияло ли это на скорость сходимости градиентного спуска? Ответ дайте в виде графика.
8. Реализуйте функции потерь $J(\theta)$ и градиентного спуска для случая многомерной линейной регрессии с использованием векторизации.
9. Покажите, что векторизация дает прирост производительности.
10. Попробуйте изменить параметр $\alpha$ (коэффициент обучения). Как при этом изменяется график функции потерь в зависимости от числа итераций градиентного спуск? Результат изобразите в качестве графика.
11. Постройте модель, используя аналитическое решение, которое может быть получено методом наименьших квадратов. Сравните результаты данной модели с моделью, полученной с помощью градиентного спуска.

**Результат выполнения:**

1. Код выгрузки данных из файла представлен ниже (путь к файлу формируется с помощью os, чтобы данный код можно было запускать на любой операционной системе Windows/MacOS/Linux):

```
file_path = os.path.join(os.path.dirname(__file__), 'data', 'ex1data1.csv')
data_frames = pd.read_csv(file_path)

x = data_frames['population']
y = data_frames['profit']

x = list(x)  # np.array(x)
y = list(y)
```

2. График представлен ниже:

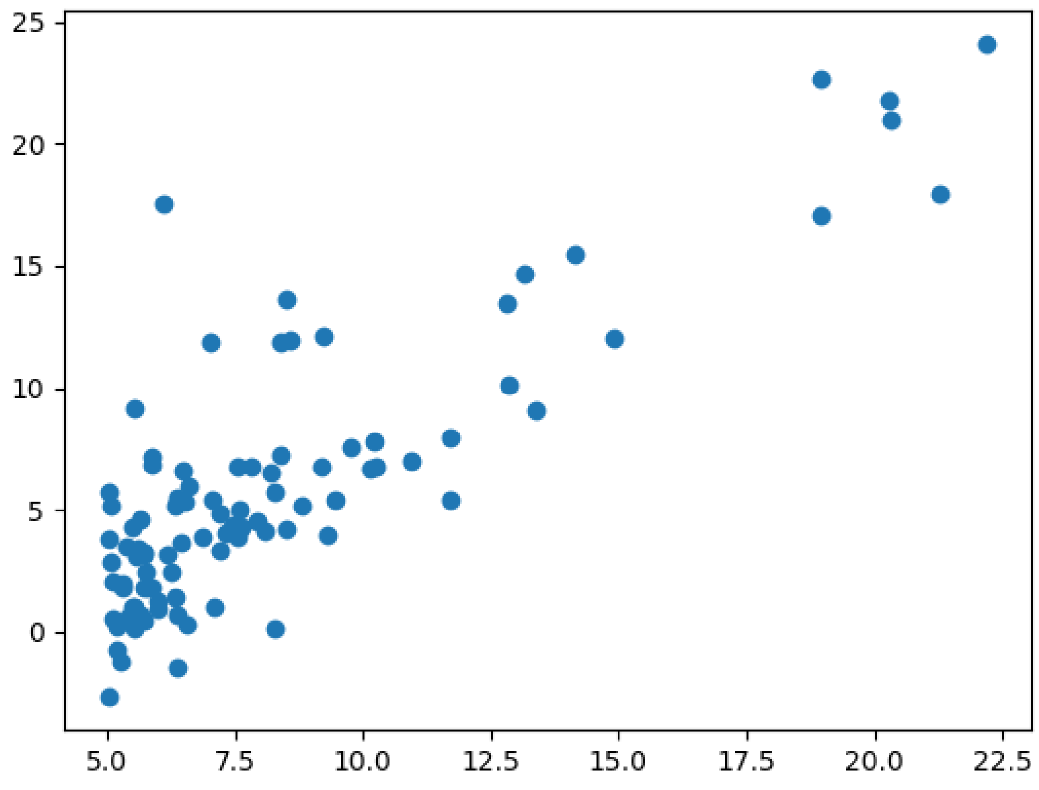


Рисунок 1 – график зависимости прибыли ресторана от населения (x – численность популяции, y – прибыль ресторана)

3. Код функции потерь:

```
def compute_cost(X, Y, theta):
    m = len(X)
```

```
    diff = []
    for i in range(0, m):
        val = pow(h0x(X[i], theta) - Y[i], 2)
        diff.append(val)
    cost = (1 / (2 * m)) * sum(diff)
    return cost
```

4. Функция градиентного спуска:

```
def gradient_descent(X, Y, theta, iterations, alpha):
    """
    From Andrew Ng implementation: without ones vector in X
    """
    m = len(X)
    J = []
    for i in range(iterations):
        val = np.zeros(len(theta))
        for j in range(0, m):
            val[0] += h0x(X[j], theta) - Y[j]
            for k in range(1, len(theta)):
                val[k] += (h0x(X[j], theta) - Y[j]) * X[j]
        for z in range(0, len(theta)):
            theta[z] = theta[z] - (alpha / m) * val[z]
        J.append(compute_cost(X, Y, theta))
    return [theta, J]
```

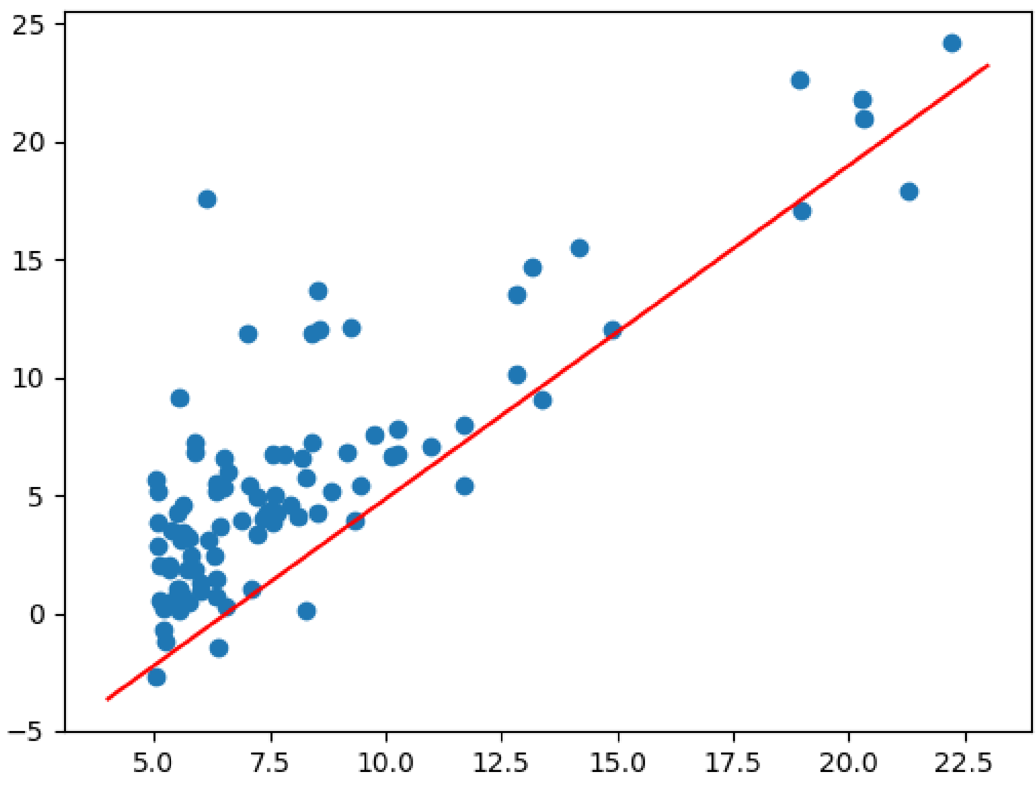


Рисунок 2 – график полученной модели

5. Графики приведены ниже:

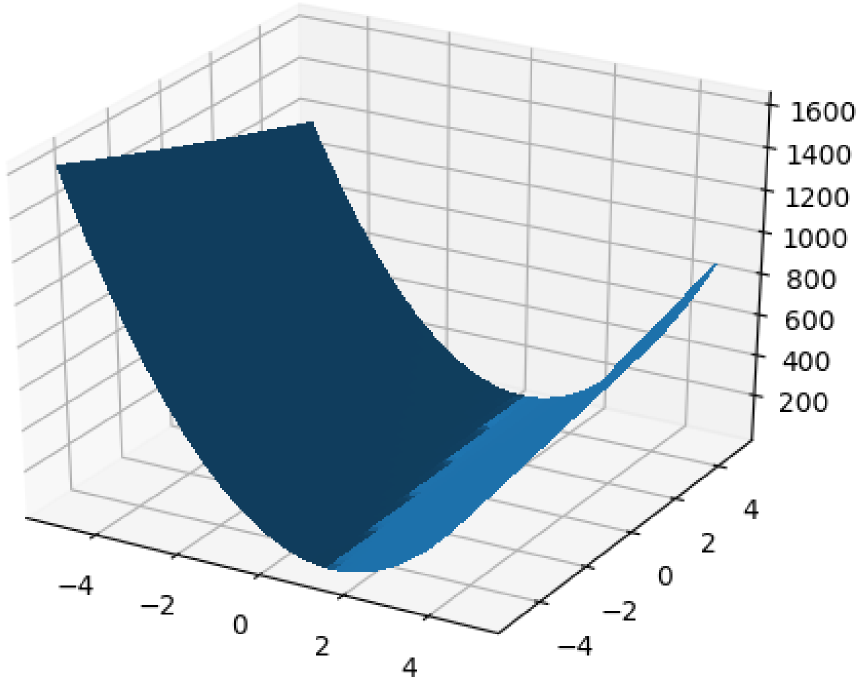


Рисунок 3 – трёхмерный график зависимости потерь от параметров модели в виде поверхности

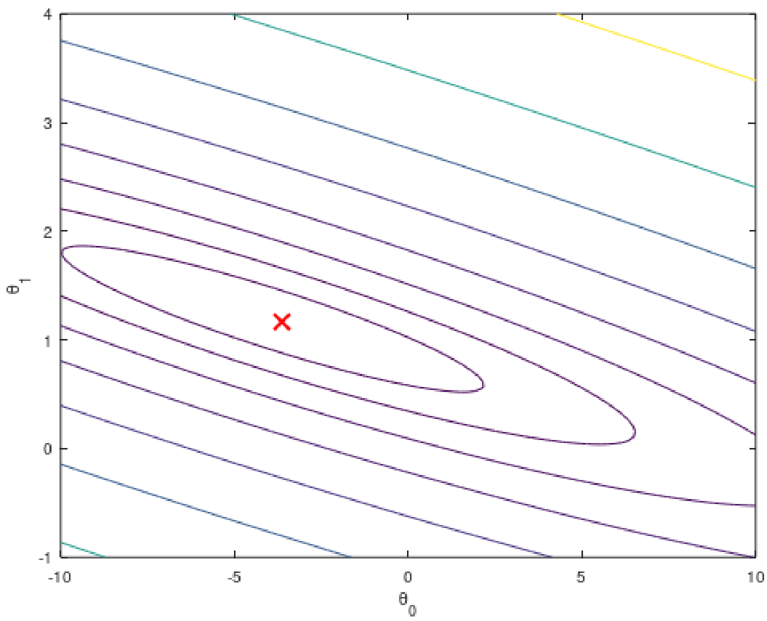


Рисунок 4 – график зависмости потерь от параметров модели в виде поверности

6. Код загрузки второго датасета:

```
    file_path = os.path.join(os.path.dirname(__file__), 'data', 'ex1data2.csv')
    data = pd.read_csv(file_path)
```

7. К сожалению, визуализация данной разницы не представляется возможной, ввиду того что в Python после 15-ой итерации числа начинают выходить за минимально допустимую точность, и, как результат, без нормализации не предоставляется возможным вычислить градиентный спуск. Один из возможных путей решения – использование decimal.Decimal класса. Однако очевиден тот факт, что самое лучшее решение – это использование нормализации признаков, потому что это позволяет сходиться градиентному спуску быстрее.

8. Код реализации:

```
def compute_cost_vectorized(X, Y, theta):
    # J = (1 / (2 * m)) * (X * theta - y)' * (X * theta - y); % equally (sum(power(X, 2)))
    m = len(X)
    temp = (h0x_vectorized(X, theta) - Y)
    return (1 / (2 * m)) * np.dot(temp.T, temp)[0][0]


def gradient_descent_vectorized(X, Y, theta, iterations, alpha):
    m = len(Y)
    J_history = []
    for i in range(iterations):
        # theta = theta - alpha * (1/m) * (((X*theta) - y)' * X)'; % Vectorized
        h0x = (h0x_vectorized(X, theta) - Y).T
        dt = np.dot(h0x, X).T
        a = alpha * (1 / m) * dt
        theta = theta - a
        J_history.append(compute_cost_vectorized(X, Y, theta))
    return [theta, J_history]
```

9. График приведён ниже:

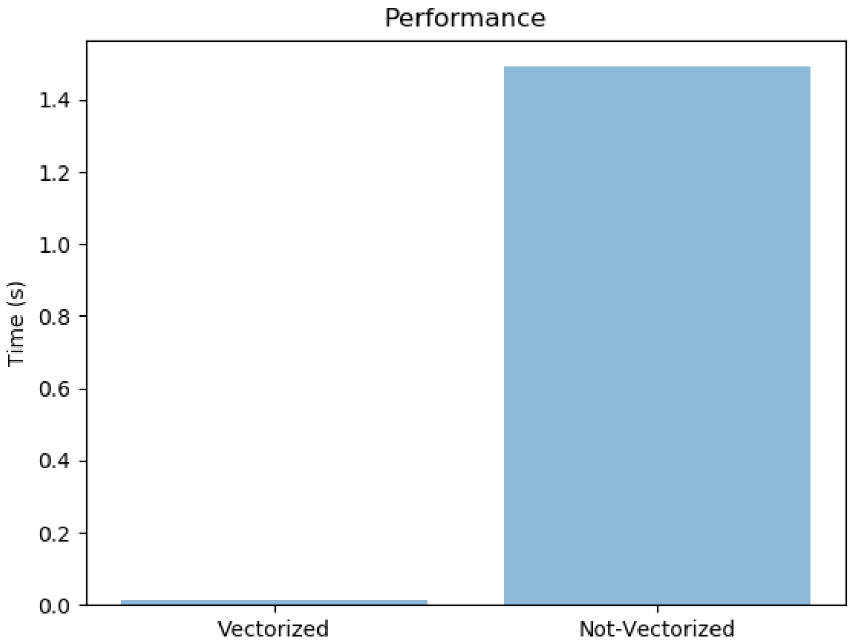


Рисунок 5 – разница между векторизованным и невекторизованным вычислением на одном и том же датасете

10. Графики приведены ниже:

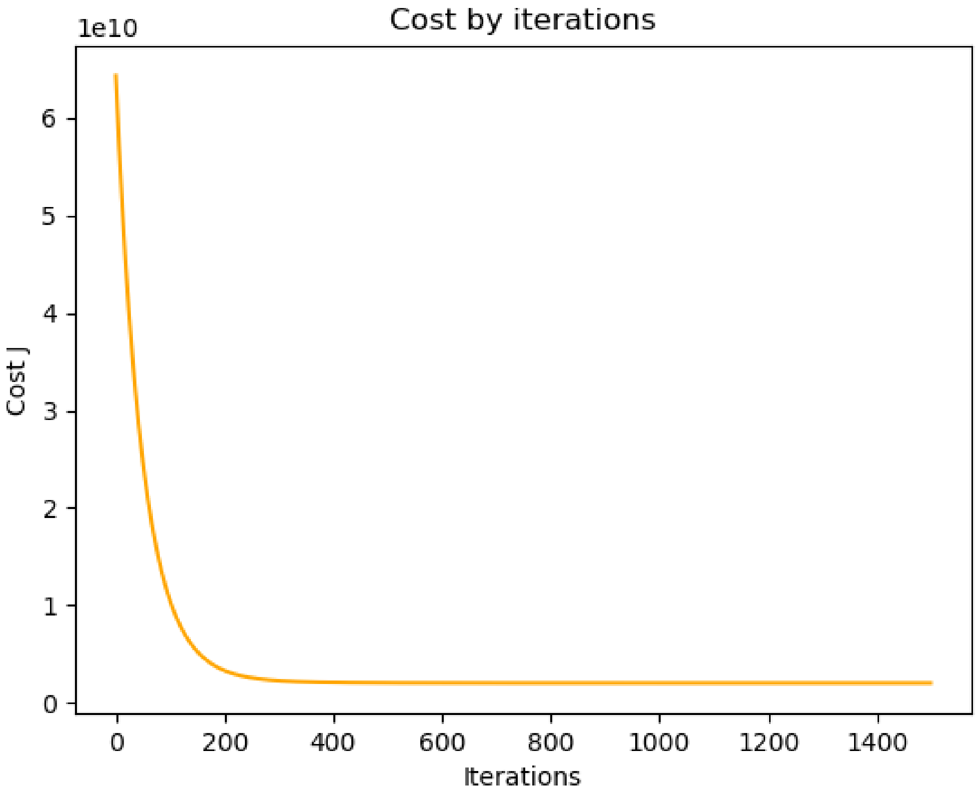


Рисунок 6 – график зависимости функции стоимости от количества итераций (alpha=0.01)

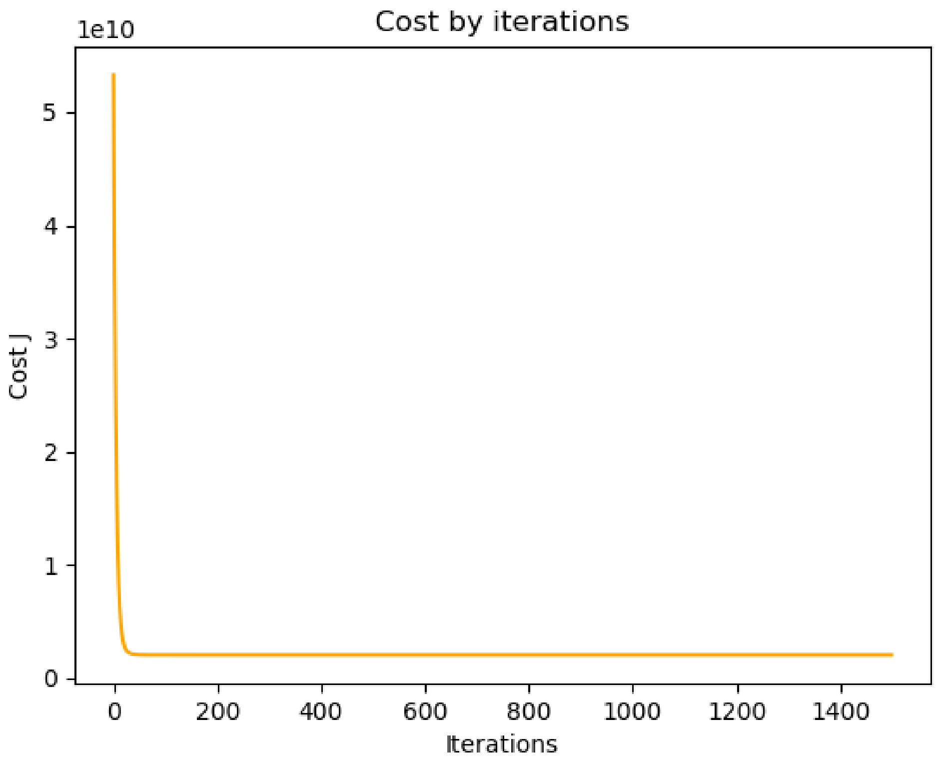


Рисунок 7 – график зависимости функции стоимости от количества итераций (alpha=0.1)

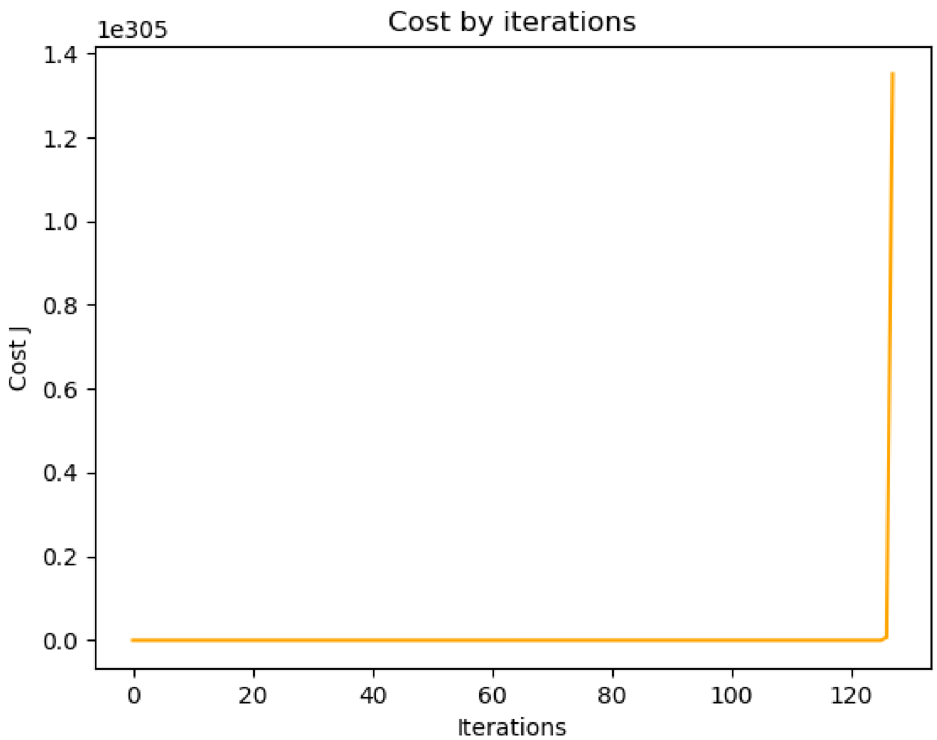


Рисунок 8 – график зависимости функции стоимости от количества итераций (alpha=10)

11. Решения получились абсолютно идентичными:

```
Solution:
[[340412.65957447]
 [110631.05027885]
 [ -6649.47427082]]

Normal equation:
[[340412.65957447]
 [110631.05027885]
 [ -6649.47427082]]
```

## Программный код:

```
from __future__ import division
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import os
import time

print("BSUIR: Machine Learning, L1")


def h0x(x_value, theta):
    return theta[0] + theta[1] * x_value


def compute_cost(X, Y, theta):
    m = len(X)
    diff = []
    for i in range(0, m):
        val = pow(h0x(X[i], theta) - Y[i], 2)
        diff.append(val)
    cost = (1 / (2 * m)) * sum(diff)
    return cost


def gradient_descent(X, Y, theta, iterations, alpha):
    """
    From Andrew Ng implementation: without ones vector in X
    """
    m = len(X)
    J = []
    for i in range(iterations):
        val = np.zeros(len(theta))
        for j in range(0, m):
            val[0] += h0x(X[j], theta) - Y[j]
            for k in range(1, len(theta)):
                val[k] += (h0x(X[j], theta) - Y[j]) * X[j]
        for z in range(0, len(theta)):
            theta[z] = theta[z] - (alpha / m) * val[z]
        J.append(compute_cost(X, Y, theta))
    return [theta, J]
```

```
def gradient_descent_not_vectorized(X, Y, theta, iterations, alpha):
    """
    For reducing impact of compute_cost_vectorized vs compute_cost
    """
    m = len(X)
    J = []
    for i in range(iterations):
        val = np.zeros(len(theta))
        for j in range(0, m):
            for k in range(0, len(theta)):
                val[k] += (h0x_vectorized(X[j], theta) - Y[j]) * X[j][k]
        for z in range(0, len(theta)):
            theta[z] = theta[z] - (alpha / m) * val[z]
        J.append(compute_cost_vectorized(X, Y, theta))
    return [theta, J]


def h0x_vectorized(X, theta):
    return X.dot(theta)


def compute_cost_vectorized(X, Y, theta):
    # J = (1 / (2 * m)) * (X * theta - y)' * (X * theta - y); % equally (sum(power(X, 2)))
    m = len(X)
    temp = (h0x_vectorized(X, theta) - Y)
    return (1 / (2 * m)) * np.dot(temp.T, temp)[0][0]


def gradient_descent_vectorized(X, Y, theta, iterations, alpha):
    m = len(Y)
    J_history = []
    for i in range(iterations):
        # theta = theta - alpha * (1/m) * (((X*theta) - y)' * X)'; % Vectorized
        h0x = (h0x_vectorized(X, theta) - Y).T
        dt = np.dot(h0x, X).T
        a = alpha * (1 / m) * dt
        theta = theta - a
        J_history.append(compute_cost_vectorized(X, Y, theta))
    return [theta, J_history]


def feature_normalization(X):
    X = X.T
    for i in range(1, len(X)):
        mu = np.mean(X[i])
        s = np.std(X[i], ddof=1)  # TODO: learn more about ddof
        X[i] = (X[i] - mu) / s
    return X.T


def normal_eqn(X, Y):
    # theta = pinv(X' * X) * (X' * y); % Vectorized
    return np.dot(np.linalg.inv(np.dot(X.T, X)), np.dot(X.T, Y))


if __name__ == '__main__':
    # 1
    file_path = os.path.join(os.path.dirname(__file__), 'data', 'ex1data1.csv')
```

```
data_frames = pd.read_csv(file_path)

x = data_frames['population']
y = data_frames['profit']

x = list(x)  # np.array(x)
y = list(y)

# 2
fig, ax = plt.subplots()
ax.scatter(x, y)

plt.show()

# 3
theta = [0, 0]
print('With theta = [0 ; 0]\nCost computed: ', compute_cost(x, y, theta))
print('Expected cost value (approx) 32.07\n')

theta = [-1, 2]
print('\nWith theta = [-1 ; 2]\nCost computed: ', compute_cost(x, y, theta))
print('Expected cost value (approx) 54.24\n')

# 4
iterations = 1500
alpha = 0.01

print('Running Gradient Descent ...\n')
# run gradient descent
theta = [0, 0]
[theta, J1] = gradient_descent(x, y, theta, iterations, alpha)

print('Theta found by gradient descent:', theta)
print("Cost: ", compute_cost(x, y, theta))
print('Expected theta values (approx):  -3.6303  1.1664\n\n')

ax.plot([4, 23], [h0x(0, theta), h0x(23, theta)], 'red')
plt.show()

# 5
u = np.arange(-5, 5, 0.1)
v = np.arange(-5, 5, 0.1)
z = np.zeros((len(u), len(v)))
for i in range(len(u)):
    for j in range(len(u)):
        z[i][j] = compute_cost(x, y, [u[i], v[j]])

u, v = np.meshgrid(u, v)

fig = plt.figure()
ax = fig.gca(projection='3d')
surf = ax.plot_surface(u, v, z, linewidth=0, antialiased=False)
plt.show()
fig, ax = plt.subplots()
plt.contour(u, v, z, np.logspace(-2, 3, 20))
plt.show()

# 6
file_path = os.path.join(os.path.dirname(__file__), 'data', 'ex1data2.csv')
```

```
data = pd.read_csv(file_path)

# 7-8
X = data.iloc[:, 0:2]  # read first two columns into X
Y = data.iloc[:, 2]  # read the third column into y
m = len(Y)
ones = np.ones((m, 1))
X = np.hstack((ones, X))  # [x1, x2] => [1, x1, x2]
theta = np.zeros((3, 1))
Y = Y[:, np.newaxis]  # convert to a matrix

print('With theta = [0; 0; 0]\nCost computed: ', compute_cost_vectorized(X, Y, theta))
print('Expected cost value (approx) 65591548106.45744\n')

print('Without normalization: \n')
# Can not be calculated since numbers are too large
# print(gradient_descent_vectorized(X, Y, theta, iterations, alpha)[0])
print('With normalization: \n')
X = feature_normalization(X)

start1 = time.time()
[theta, J1] = gradient_descent_not_vectorized(X, Y, theta, iterations, alpha)
end1 = time.time()
print('Time gradient not vectorized: ', end1 - start1, theta)

start2 = time.time()
[gdv, J] = gradient_descent_vectorized(X, Y, np.zeros((3, 1)), iterations, alpha)
end2 = time.time()

print('Vectorized time: ', end2 - start2)
print('Solution: ')
print(gdv)

# 9

objects = ('Vectorized', 'Not-Vectorized')
y_pos = np.arange(len(objects))
performance = [end2-start2, end1-start1]

plt.bar(y_pos, performance, align='center', alpha=0.5)
plt.xticks(y_pos, objects)
plt.ylabel('Time (s)')
plt.title('Performance')

plt.show()

# 10

year = range(0, iterations)

plt.plot(year, J, color='orange')
plt.xlabel('Iterations')
plt.ylabel('Cost J')
plt.title('Cost by iterations')
plt.show()

# 11

theta = normal_eqn(X, Y)
```

```
print('Normal equation: \n')
print(theta)
```