Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ»

Кафедра информатики

Отчет по предмету:

«Машинное обучение»

По лабораторной работе №2

«Логистическая регрессия. Многоклассовая классификация»

Выполнил: Сенькович Дмитрий Сергеевич

магистрант кафедры информатики

группы №858642

Проверил: Стержанов Максим Валерьевич

доцент, кандидат технических наук

Минск 2019

**Оглавление**

[1 Введение 3](#_Toc25617895)

[2 Предсказание поступления студента 4](#_Toc25617896)

[2.1 Постановка задачи 4](#_Toc25617897)

[2.2 Чтение данных 4](#_Toc25617898)

[2.3 График поступления студента 4](#_Toc25617899)

[2.4 Реализация функции потерь и градиентного спуска с векторизацией 5](#_Toc25617900)

[2.5 Реализация функции предсказания поступления студента 7](#_Toc25617901)

[2.6 Реализация предсказания другими методами 7](#_Toc25617902)

[2.7 Сравнение реализаций предсказания поступления и графики прямой предсказания, разделяющий данные 8](#_Toc25617903)

[3 Предсказание прохождения контроля изделия 12](#_Toc25617904)

[3.1 Постановка задачи 12](#_Toc25617905)

[3.2 Чтение данных 12](#_Toc25617906)

[3.3 График прохождения контроля 12](#_Toc25617907)

[3.4 Построение полиномиальных фич 13](#_Toc25617908)

[3.5 Добавление L2-регуляризации 14](#_Toc25617909)

[3.6 Реализация других методов оптимизации 16](#_Toc25617910)

[3.7 Реализация функции предсказания прохождения контроля изделием 17](#_Toc25617911)

[3.8 Построение разделяющей поверхности 17](#_Toc25617912)

[3.9 Различные значения параметра регуляризации 18](#_Toc25617913)

[4 Распознавание цифр 20](#_Toc25617914)

[4.1 Постановка задачи 20](#_Toc25617915)

[4.2 Чтение данных 20](#_Toc25617916)

[4.3 Визуализация картинок из выборки 20](#_Toc25617917)

[4.4 Реализация многоклассовой классификации по методу “один против всех” 21](#_Toc25617918)

[4.5 Функция предсказания классификатором 22](#_Toc25617919)

[4.6 Подсчет процента правильных классификаций на обучающей выборке 22](#_Toc25617920)

# 1 Введение

В лабораторной работе рассматривается задача классификации в машинном обучении и решение такой задачи - логистическая регрессия. Для решение этой задачи можно было бы использовать и линейную регрессию, но, к сожалению, влияние экземпляров выборки, не соответствующих общему прогнозу, сильно влияло бы на качество такой модели.

Поэтому простейшее решение задачи классификации - логистическая регрессия. Логистическая регрессия во многом схожа с линейной регрессией, более того, для оптимизации этой модели также можно использовать градиентный спуск.

# 2 Предсказание поступления студента

### 2.1 Постановка задачи

Даны оценки студента по двум экзаменам и информация о поступлении этого студента: 0 - не поступил, 1 - поступил.

### 2.2 Чтение данных

Считаем данные из файла ex2data1.txt:

x = []

y = []

with open('ex2data1.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])

### 2.3 График поступления студента

Построим график поступления студента в зависимости от оценок до двум экзаменам. По осям Ox и Oy будем откладывать значения оценок студента, а цвет соответствующей точки будет указывать поступил студент или нет. Зеленым цветом будут указаны поступившие студенты, а красным - нет:

x = np.matrix(x)

colors = ['green' if val else 'red' for val in y]

plt.scatter(x[:,1].A1, x[:,2].A1, c=colors)

plt.show()

Получившийся график представлен ниже:

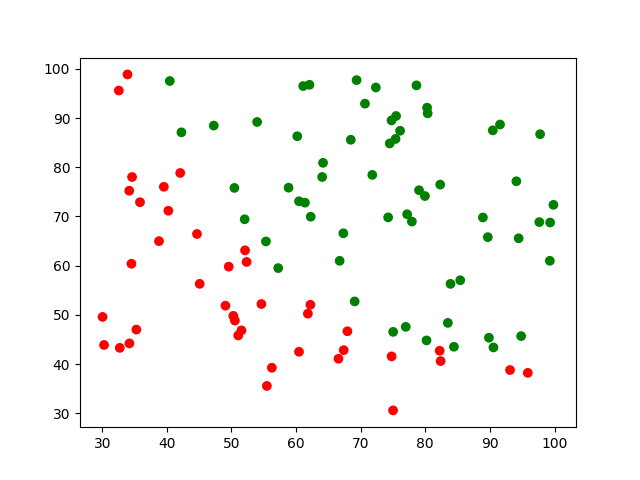


Рисунок 2.3.1. График поступления студента

### 2.4 Реализация функции потерь и градиентного спуска с векторизацией

Для начала, определимся с функцией гипотезы для логистической регрессии. Чтобы уйти от проблемы, которая была бы с линейной регрессией, а именно слишком резкое изменение модели при неожиданных экземпляров выборки, мы будем отображать значение произведения вектора параметров на значение вектора экземпляра выборки с помощью логистической функции следующим образом:

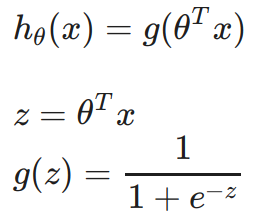


Рисунок 2.4.1. Функция гипотезы для логистической регрессии

Эта функция имеет следующий вид:

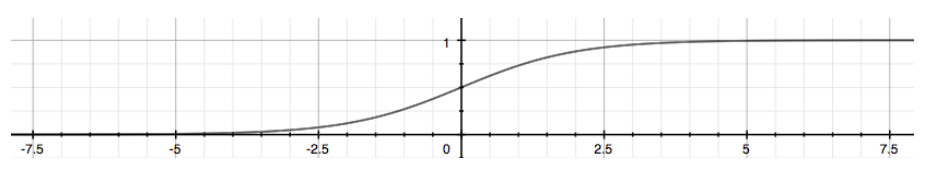


Рисунок 2.4.2. График логистической функции

Данный вид логистической функции обеспечивает “плавное” предсказывание функцией гипотезы принадлежности экземпляра к классу 0 или 1.

Далее, приведем новые формулы функций гипотезы, потерь и частных производных в векторном виде. Несколько замечаний: функция стоимости приобрела такой вид ввиду использования новой функции гипотезы. Для каждого экземпляра имеет значение лишь левая или правая часть слагаемых т.к. y принимает значения 0 либо 1. Итак, формулы для векторной реализации:



Рисунок 2.4.3. Функция гипотезы в векторном виде

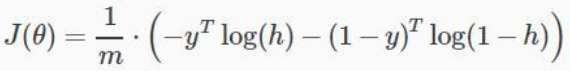


Рисунок 2.4.4. Функция потерь в векторном виде

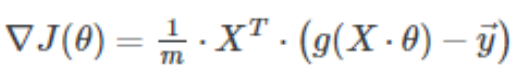


Рисунок 2.4.5. Вычисление частных производных в векторном виде

Ниже приведен код, реализующий данный векторный подход:

def sigmoid(z):

return 1.0 / (1 + np.exp(-z))

def h(ot, x):

return sigmoid(x.dot(ot))

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

h\_val = h(ot, x)

return (-y \* np.log(h\_val) - (1 - y) \* np.log(1 - h\_val))[0].item(0) / m

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

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

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

Заметим, что функция самого градиентного спуска не изменилась, потому и не включена в листинг кода выше.

### 2.5 Реализация функции предсказания поступления студента

Реализуем функцию следующим образом:

def admitted(o, x1, x2):

return sigmoid(o.item(0) + o.item(1) \* x1 + o.item(2) \* x2)

Функция возвращает вероятность поступления студента в зависимости от его оценок.

### 2.6 Реализация предсказания другими методами

Реализуем предсказание методами Нелдера — Мида и Бройдена — Флетчера — Гольдфарба — Шанно, реализованными в библиотеке scipy:

def optimize\_scipy(x, y, algorithm):

x = np.array(x)

y = np.array(y)

mean\_res, diff\_res = normalize\_x(x)

o = np.zeros((3, 1))

Result = optimize.minimize(fun=j\_scipy, x0=o, args=(x, y), method=algorithm, jac=j\_o\_der\_scipy)

o = Result.x

print("o %s " % o)

colors = ['green' if val else 'red' for val in y]

plt.scatter(x[:, 1], x[:, 2], c=colors)

line\_x = [-0.5, 0.5]

# because o1 + o2\*x1 + 03\*x2 = 0 for all the x1, x2 on such line

line\_y = - (o.item(0) + np.dot(o.item(1), line\_x)) / o.item(2)

plt.plot(line\_x, line\_y)

x1 = 78

x1 = (x1 - mean\_res[0]) / diff\_res[0]

x2 = 78

x2 = (x2 - mean\_res[0]) / diff\_res[0]

print(admitted(o, x1, x2))

plt.show()

### 2.7 Сравнение реализаций предсказания поступления и графики прямой предсказания, разделяющий данные

Запустим получившуюся реализацию градиентного спуска:

x = np.matrix(x)

y = np.matrix(y)

normalize\_x(x)

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)

colors = ['green' if val else 'red' for val in y.A1]

plt.scatter(x[:,1].A1, x[:,2].A1, c=colors)

line\_x = [-0.5, 0.5]

# because o1 + o2\*x1 + 03\*x2 = 0 for all the x1, x2 on such line

line\_y = - (o.item(0) + np.dot(o.item(1), line\_x)) / o.item(2)

plt.plot(line\_x, line\_y)

plt.show()

Ниже представлен график разделяющей прямой на наших данных:

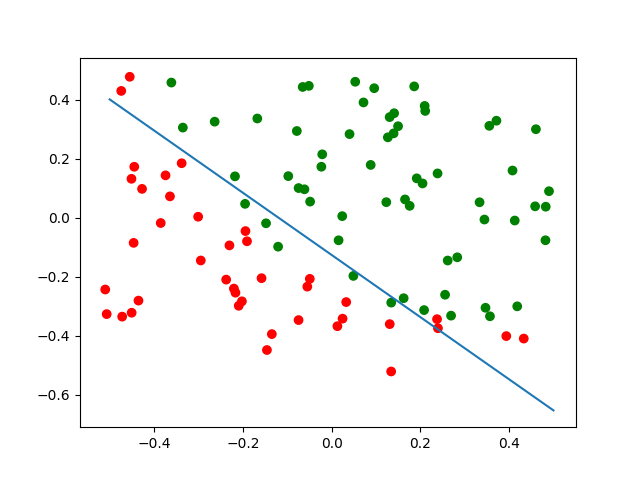


Рисунок 2.7.1. Данные и разделяющая прямая, полученная методом градиентного спуска

Для сравнения, приведем также и скорость работы нашего алгоритма: почти 35 секунд.

Теперь запустим библиотечные реализации алгоритмов, упомянутых в предыдущем пункте:

optimize\_scipy(x, y, 'BFGS')

optimize\_scipy(x, y, 'Nelder-Mead')

Результаты запуска такие:

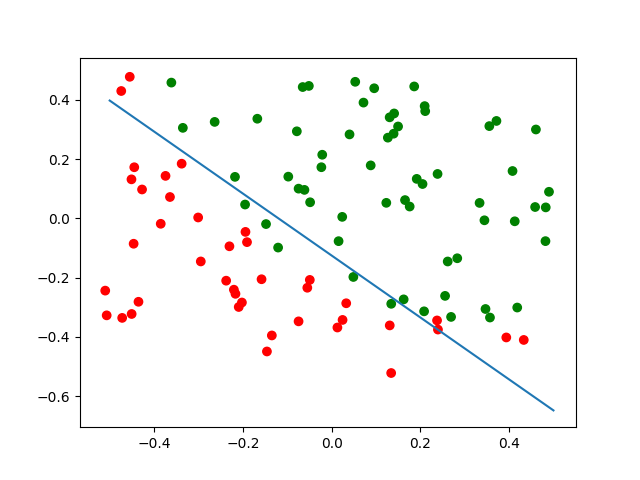


Рисунок 2.7.2. Данные и разделяющая прямая, полученная методом Бройдена — Флетчера — Гольдфарба — Шанно

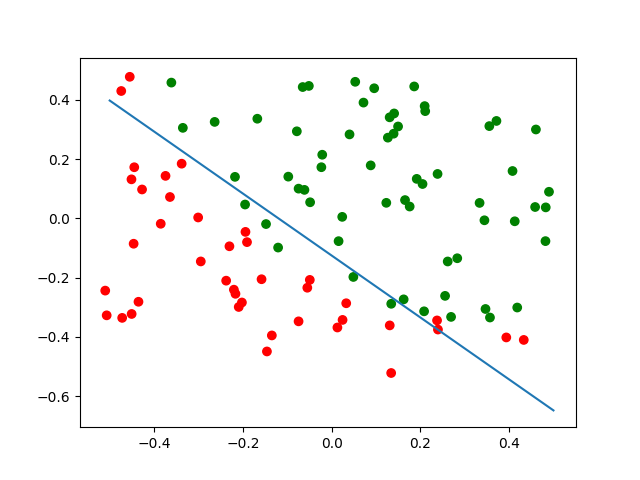


Рисунок 2.7.3. Данные и разделяющая прямая, полученная методом Нелдера — Мида

Как видим, и нашим, и библиотечными способами достигнуто примерно одинаковое качество предсказания. Но есть огромная разница во времени обучения. Если наш алгоритм исполнялся порядка 35 секунд, то библиотечные алгоритмы завершили свою работу за 2.6 и 12 миллисекунд соответственно.

# 3 Предсказание прохождения контроля изделия

### 3.1 Постановка задачи

Даны результаты теста изделия и соответствующее значение прохождения контроля: 0 - контроль не пройден, 1 - пройден.

### 3.2 Чтение данных

Считаем данные из файла ex2data2.txt:

x = []

y = []

with open('ex2data2.txt', 'r') as data\_file:

for line in data\_file:

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

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

y.append(data\_i[2])

### 3.3 График прохождения контроля

Построим график прохождения контроля изделиями из обучающей выборки. Так же, как и в предыдущей задаче, отметим зеленый цветом детали, прошедшие контроль, и красным - не прошедшие:

x = np.matrix(x)

colors = ['green' if val else 'red' for val in y]

plt.scatter(x[:,1].A1, x[:,2].A1, c=colors)

plt.show()

Соответствующий график приведен ниже:

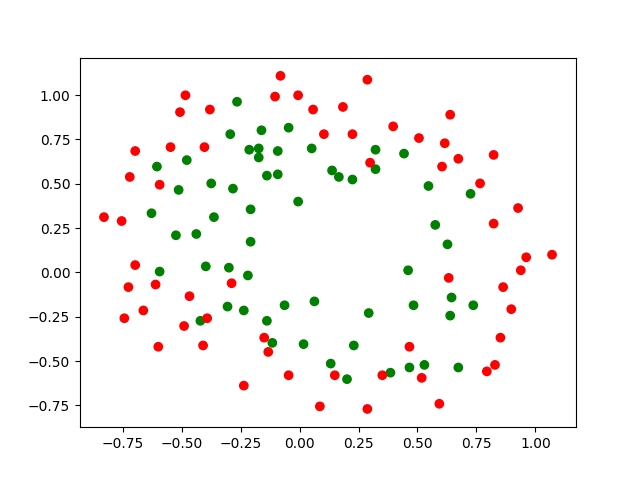


Рисунок 3.3.1. Результаты прохождения контроля изделиями

### 3.4 Построение полиномиальных фич

Линейная и логистическая регрессии способны апроксимировать как линейные, так и нелинейные зависимости. В нашем примере выше видно, что зависимость между результатами тестов и результатом прохождения контроля нелинейная. Поэтому далее бы добавим к нашим уже существующим фичам полиномиальные фичи, включающие в себя все комбинации фич x1 и x2 не выше 6 степени, то есть в итоге у нас получится 28 фич:

def calc\_o\_to\_pow():

for i in range(7):

for j in range(7 - i):

o\_to\_pow.append([i, j])

def calc\_hypothesis\_x(x):

m = x.shape[0]

for i in range(m):

xi = x[i]

hypothesis\_x.append(calc\_hypothesis\_x\_row(xi))

def calc\_hypothesis\_x\_row(xi):

row = []

for pow in o\_to\_pow:

i = pow[0]

j = pow[1]

row.append((xi[0]\*\*i)\*(xi[1]\*\*j))

return row

Здесь мы создаем новую выборку из 28 фич из изначально данных двух.

### 3.5 Добавление L2-регуляризации

Одна из главных проблем в машинном обучении - переобучение. Переобучение представляет из себя такое состояние модели, когда ошибка обучения практически или даже нулевая, что может показаться хорошим знаком, но, на самом деле, показывает, что данная модель не способна обобщаться на новые данные. Пример графиков хорошей и переобученной моделей представлен ниже:

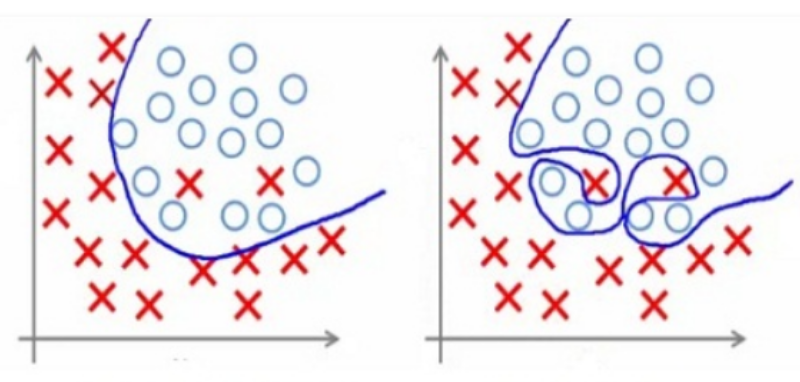


Рисунок 3.5.1. Пример хорошей и переобученной модели

Проблему переобучения можно решать по-разному: добавлять новые данные, упрощать модель, и добавлять регуляризацию, что и рассматривается в данной лабораторной работе.

Идея регуляризации заключается в следующем: если параметры модели будут меньше, то значение каждой части в слагаемом в функции потерь, соответствующее каждому из параметров, будет меньше, что в итоге результирует в построении более простой функции гипотезы, что, в свою очередь, менее подвержено переобучению.

Один из способов достижения такой цели - “штрафовать” функцию потерь за большие значения параметров модели. Один из вариантов - L2 регуляризация:

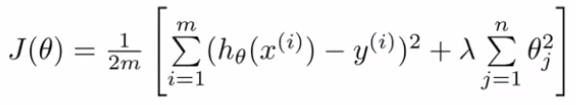


Рисунок 3.5.2. Функция стоимость с L2 регуляризацией

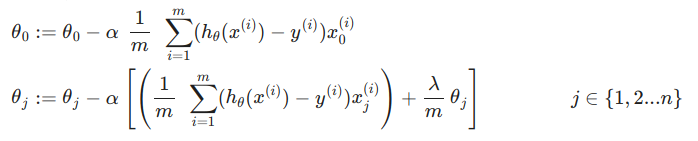


Рисунок 3.5.3. Оптимизация параметров модели с L2 регуляризацией

Как видим, к функции стоимости добавляется взвешенная сумма квадратов ее параметров (за исключением bias параметра), а к частным производным - взвешенная часть параметра (опять же, за исключением bias параметра).

Реализации функций потерь и частных производных с L2 регуляризацией представлены ниже (остальные детали реализации градиентного спуска остаются без изменений):

def j(m, ot):

h\_val = h(ot)

return ((-yt.dot(np.log(h\_val)) - (1 - yt).dot(np.log(1 - h\_val)))[0].item(0) / m) + np.sum(np.power(ot[1:], 2))\*l/(2\*m)

def j\_o\_der(m, ot):

h\_vec\_val = h(ot)

ot\_restricted = np.copy(ot)

ot\_restricted[0][0] = 0

return ((hypothesis\_x\_transposed.dot(h\_vec\_val - y)) / m) + ot\_restricted\*l/m

Обучим нашу модель с L2 регуляризацией на расширенном наборе фич градиентным бустингом:

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

### 3.6 Реализация других методов оптимизации

Реализуем также и два других метода оптимизации из предыдущей задачи помощью библиотеки scipy:

def j\_scipy(o, x, y):

h\_val = h(o.reshape(-1, 1))

m = hypothesis\_x.shape[0]

return ((-yt.dot(np.log(h\_val)) - (1 - yt).dot(np.log(1 - h\_val)))[0].item(0) / m) + np.sum(np.power(o[1:], 2))\*l/(2\*m)

def j\_o\_der\_scipy(o, x, y):

h\_vec\_val = h(o.reshape(-1, 1))

m = hypothesis\_x.shape[0]

o\_restricted = np.copy(o).reshape(-1, 1)

o\_restricted[0] = 0

return ((hypothesis\_x\_transposed.dot(h\_vec\_val - y.reshape(-1, 1)) / m) + o\_restricted\*l/m).flatten()

def optimize\_scipy(algorithm):

o = np.zeros((28, 1))

Result = optimize.minimize(fun=j\_scipy, x0=o, args=(hypothesis\_x, y), method=algorithm, jac=j\_o\_der\_scipy, options={'maxiter': 10000000})

o = Result.x

print("o %s " % o)

return o

optimize\_scipy('BFGS')

optimize\_scipy('Nelder-Mead')

### 3.7 Реализация функции предсказания прохождения контроля изделием

Функция предсказания прохождения контроля изделием в зависимости от результатов прохождения тестов реализована с выбором значения границы принятия решения 0.5 и выглядит следующим образом:

def predict(hypothesis\_x, o):

return sigmoid(hypothesis\_x.dot(o)) >= 0.5

### 3.8 Построение разделяющей поверхности

Запустим алгоритм градиентного спуска с шагом обучения 0.001, максимальным количеством итераций 160.000 и параметров регуляризации 0.1:

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

Изобразим разделяющую поверхность. Светло-красным цветом будет выделена часть плоскость результатов тестов, где алгоритм считает, что изделие не пройдет контроль, а светло-зеленым - пройдет:

t = np.linspace(-1, 1.25, 300)

temp = []

for tt in t:

for ttt in t:

temp.append([tt, ttt])

t = np.array(temp)

hypothesis\_x = []

calc\_hypothesis\_x(t)

hypothesis\_x = np.array(hypothesis\_x)

values = predict(hypothesis\_x, o)

colors\_v = ['#90ee90' if val else '#ff9185' for val in values]

plt.scatter(t[:,0], t[:,1], c=colors\_v, s = 3)

colors = ['green' if val else 'red' for val in y]

plt.scatter(x[:,0], x[:,1], c=colors)

plt.show()

Получившийся график выглядит вот так:

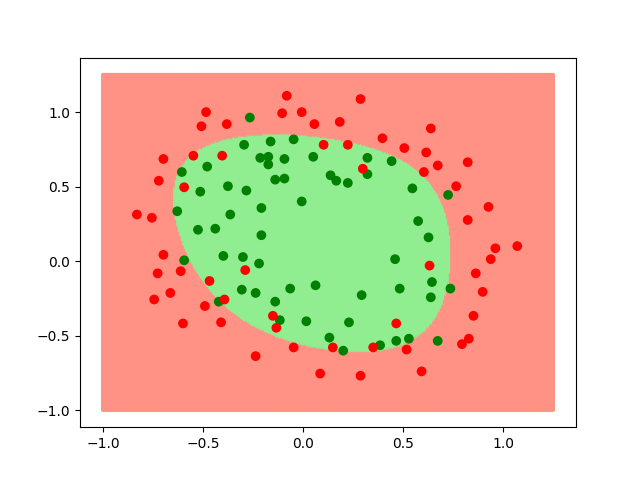


Рисунок 3.8.1. Разделяющая поверхность и данные. Градиентный спуск

### 3.9 Различные значения параметра регуляризации

Так как суть регуляризации заключается в избежании переобучения, но стоит предположить, что при маленьких значениях параметра регуляризации наша модель переобучится, так как у нас слишком много полиномиальных фич. В случае слишком большого параметра регуляризации мы должны столкнуться с другой важной проблемой машинного обучения: high bias - ситуация, обратная переобучению, в которой модель слишком проста и не способна давать адекватные ответы даже на обучающей выборке. Это происходит, потому что при слишком больших параметрах регуляризации оптимизация функции потерь приведет к чересчур большому уменьшению параметров модели, в результате чего гипотеза станет константой.

Проверим это на практике, запустив алгоритм Бройдена — Флетчера — Гольдфарба — Шанно (для более быстрой сходимости) на различных параметрах регуляризации: 0, 0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000. Ниже представлены результаты, т.е. разделяющие поверхности:

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

Рисунок 3.9.1. Разделяющие поверхности и данные. Различные параметры регуляризации

Графики подтверждают рассуждения выше: при параметре регуляризации 0, т.е. отсутствии регуляризации, мы получили модель, максимально подстроившуюся под обучающую выборку. С увеличение параметра регуляризации мы видим улучшения в обобщении данных моделью вплоть до значения параметра регуляризации 10. Начиная с этого значения видно, что модель начинает не “попадать” в значения выборки. Заканчивается это на значении параметра 1000, т.е. последним графиком, на котором видно, что модель стала настолько проста, что всегда относит изделия к непройденным контроль на обучающей выборке.

# 4 Распознавание цифр

### 4.1 Постановка задачи

Дана выборка изображений 20x20, каждое значение которой представляет собой картинку, изображенную значениями яркости в каждой из точек. Каждой картинке соответствует метка - цифра, которая соответствует картинка.

### 4.2 Чтение данных

Считаем данные:

data = loadmat("ex2data3.mat")

x = np.array(data['X'])

x = np.column\_stack([[1]\*(x.shape[0]), x])

xt = x.transpose()

target = np.array(data['y']).reshape(-1, 1)

Считываем данные, сразу добавляем bias.

### 4.3 Визуализация картинок из выборки

Визуализируем 10 различных картинок из набора (каждая цифра один раз):

fig, axs = plt.subplots(1, 10)

t = -1

i = 0

for j in range(target.shape[0]):

if target[j] != t:

axs[i].imshow(x[j].reshape(20, 20), cmap='hot')

axs[i].axis("off")

i += 1

t = target[j]

print(t)

plt.show()

Получившееся изображение 10-и цифр представлено ниже:



Рисунок 4.1.1. Цифры из обучающей выборки

### 4.4 Реализация многоклассовой классификации по методу “один против всех”

Для начала реализуем бинарный классификатор: в качестве метода выбираем логистическую регрессию с L2 регуляризацией и векторизацией. Реализация аналогична реализации из предыдущей задачи, поэтому листинг код опущен.

Многоклассовый классификатор по методу “один против всех”: это классификатор из 10-и бинарных классификаторов, каждый отделяет конкретный класс от всех остальных.

Покажем реализацию градиентным спуском:

classifiers = []

np.set\_printoptions(suppress=True)

index\_to\_number = [10, 1, 2, 3, 4, 5, 6, 7, 8, 9]

for i in range(10):

print('--------------')

print('number is %s' % (i+1))

y = [1 if val == index\_to\_number[i] else 0 for val in target]

y = np.array(y).reshape(-1, 1)

w, its, its\_hist, err\_hist = gradient\_descent(y)

classifiers.append(w.transpose())

print('--------------')

print(len(classifiers))

Для ускорения сходимости также реализуем многоклассовый классификатор, в котором каждый алгоритм обучается библиотечной функцией scipy optimize.fmin\_cg - оптимизированная версия градиентного спуска:

classifiers = []

index\_to\_number = [10, 1, 2, 3, 4, 5, 6, 7, 8, 9]

for i in range(10):

y = [1 if val == index\_to\_number[i] else 0 for val in target]

y = np.array(y).reshape(-1, 1)

w0 = np.array([0.0]\*401)

classifiers.append(optimize.fmin\_cg(

f=cost\_scipy,

x0=w0,

fprime=gradient\_scipy,

args=(x, y, l),

maxiter=100

))

### 4.5 Функция предсказания классификатором

Реализуем функцию предсказания получившимся классификатором цифр на картинках (код общий как для самописного градиентного спуска, так и для библиотечной версии):

def is\_n(classifier, xi):

return sigmoid(classifier.dot(xi.reshape(-1, 1)))[0].item(0)

def predict(x, classifiers):

m = x.shape[0]

predictions = [0] \* m

classifier\_to\_prediction = [10, 1, 2, 3, 4, 5, 6, 7, 8, 9]

for i in range(m):

xi = x[i]

probabilities = [0] \* 10

for k in range(10):

classifier = classifiers[k]

probabilities[k] = is\_n(classifier, xi)

prediction = classifier\_to\_prediction[np.argmax(probabilities)]

predictions[i] = prediction

return predictions

Здесь каждый бинарный классификатор многоклассового классификатора возвращает вероятность принадлежности экземпляра выборки классу этого бинарного классификатора, т.е. бинарный классификатор, обученный распознавать 1 выдаст вероятность того, что на картинке изображена цифра 1 и т.д.. Далее, для каждого экземпляра выбирается бинарный классификатор с наибольшей вероятностью. К цифре данного бинарного классификатора наш многоклассовый классификатор и относит экземпляр выборки.

### 4.6 Подсчет процента правильных классификаций на обучающей выборке

Подсчитывать процент правильных классификаций будем следующим образом:

predictions = predict(x, classifiers)

m = x.shape[0]

success\_count = 0

for i in range(x.shape[0]):

prediction\_i = predictions[i]

target\_i = target[i].item(0)

success\_count += 1 if target\_i == prediction\_i else 0

print("success rate: %s" % ((success\_count / m)\*100))

Как видим, здесь используется переменная classifiers - это массив обученных бинарных классификаторов. Сами бинарные классификаторы можно получить двумя способами: обучить с помощью самописной и библиотечной реализации градиентного спуска.

При использовании самописного градиентного спуска алгоритм сходится очень медленно, например, лучшая попытка достигнуть 95% правильно классифицированных картинок закончилась 2.000.000 итерациями и заняла порядка часа, в результате чего процент правильно классифицированных цифр составил 91.06%.

При использовании библиотечного градиента бинарные классификаторы обучаются очень быстро, достигая процента правильно классифицированных изображений 95.28% за 50 итераций.