Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ»

Кафедра информатики

Отчет по предмету:

«Машинное обучение»

По лабораторной работе №3

«Переобучение и регуляризация»

Выполнил: Сенькович Дмитрий Сергеевич

магистрант кафедры информатики

группы №858642

Проверил: Стержанов Максим Валерьевич

доцент, кандидат технических наук

Минск 2019

**Оглавление**

[1 Введение 2](#_Toc25694630)

[2 Предсказание объема воды, вытекающей из дамбы 4](#_Toc25694631)

[2.1 Постановка задачи 4](#_Toc25694632)

[2.2 Чтение данных 4](#_Toc25694633)

[2.3 График обучающей выборки 4](#_Toc25694634)

[2.4 Реализация функции стоимости потерь с L2 регуляризацией 5](#_Toc25694635)

[2.5 Реализация функции градиентного спуска с L2 регуляризацией 6](#_Toc25694636)

[2.6 Построение первой модели 7](#_Toc25694637)

[2.7 Построение графика процесса обучения 8](#_Toc25694638)

[2.8 Функция добавления p - 1 новых признаков 10](#_Toc25694639)

[2.9 Функция нормализации признаком 10](#_Toc25694640)

[2.10 Обучение первой модели с полиномиальными фичами. Переобучение. График данных и модели. График процесса обучения 11](#_Toc25694641)

[2.11 Обучение модели с полиномиальными фичами с различными параметрами 14](#_Toc25694642)

[2.12 Поиск лучшего коэффициента регуляризации 18](#_Toc25694643)

[2.13 Ошибка и график модели на контрольной выборке 20](#_Toc25694644)

# 1 Введение

В лабораторной работе рассматривается проблема переобучения и одно из решений этой проблемы - регуляризация.

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:

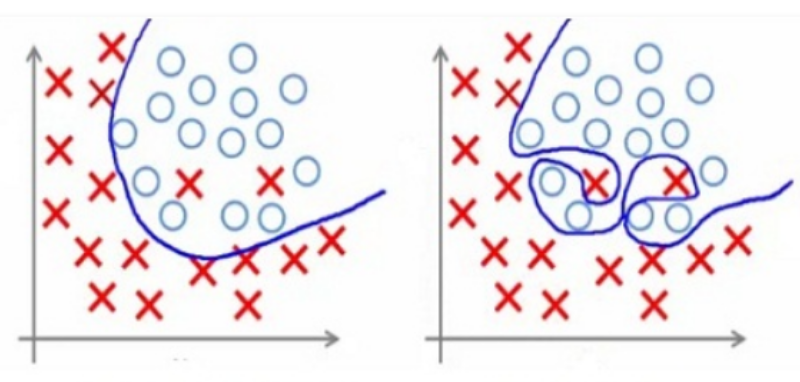


Рисунок 1.1. Пример хорошей и переобученной модели

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:

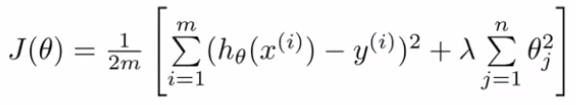


Рисунок 1.2. Функция стоимость с L2 регуляризацией

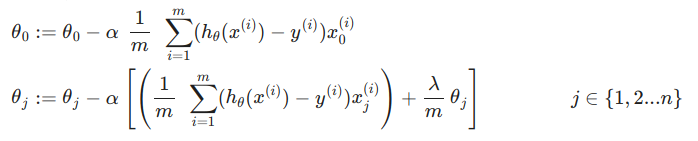


Рисунок 1.3. Оптимизация параметров модели с L2 регуляризацией

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).

# 2 Предсказание объема воды, вытекающей из дамбы

### 2.1 Постановка задачи

Даны изменения уровня воды и соответствующим этим изменениям объемы воды, вытекающие из дамбы. Требуется предсказать этот объем в зависимости от изменений воды.

### 2.2 Чтение данных

Считаем данные:

data = loadmat("ex3data1.mat")

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

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

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

y = y.transpose()

val\_x = np.array(data['Xval'])

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

val\_y = np.array(data['yval']).reshape(-1, 1)

val\_y = val\_y.transpose()

test\_x = np.array(data['Xtest'])

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

test\_y = np.array(data['ytest']).reshape(-1, 1)

test\_y = test\_y.transpose()

### 2.3 График обучающей выборки

Построим график зависимости объема воды от изменений воды:

plt.scatter(np.array(data['X']), np.array(data['y']))

plt.show()

Получившийся график представлен ниже:

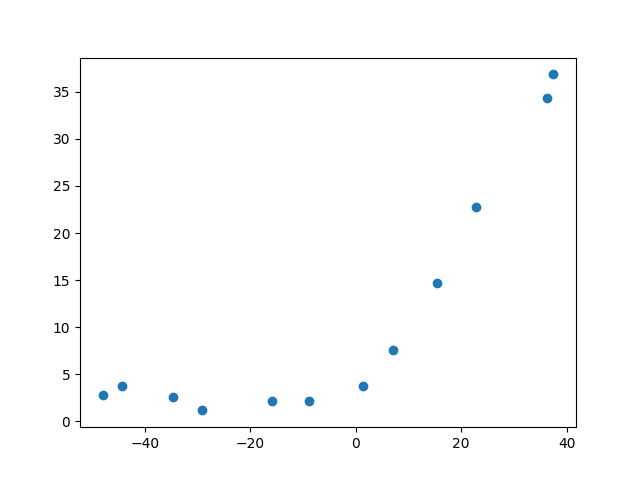


Рисунок 2.3.1. График обучающей выборки

### 2.4 Реализация функции стоимости потерь с L2 регуляризацией

Реализация функции стоимости аналогична реализации из первой лабораторной работы с добавлением слагаемого регуляризации, как во второй лабораторной работе:

def hypothesis(x, wt):

return x.dot(wt)

def cost(wt, x, y):

calculated\_hypothesis = hypothesis(x, wt)

m = x.shape[0]

hy\_diff = calculated\_hypothesis - y.transpose()

return (hy\_diff.transpose().dot(hy\_diff))[0].item(0) / (2 \* m) + np.sum(np.power(wt[1:], 2))\*l/(2\*m)

def cost\_scipy(w, x, y, l = 0):

calculated\_hypothesis = hypothesis(x, w.reshape(-1, 1))

m = y.shape[1]

hy\_diff = calculated\_hypothesis - y.transpose()

return (hy\_diff.transpose().dot(hy\_diff))[0].item(0) / (2 \* m) + np.sum(np.power(w[1:], 2))\*l/(2\*m)

Здесь приведены реализация для использования самописного градиентного спуска и библиотечного из библиотеки scipy.

### 2.5 Реализация функции градиентного спуска с L2 регуляризацией

Реализация функции градиентного спуска аналогична реализации из первой лабораторной работы с добавлением слагаемого регуляризации, как во второй лабораторной работе. Для начала приведем функции вычисления частных производных (для самописной и библиотечной реализации градиентного спуска соответственно):

def gradient(wt, x, y):

calculated\_hypothesis = hypothesis(x, wt)

wt\_restricted = np.copy(wt)

wt\_restricted[0][0] = 0

m = x.shape[0]

xt = x.transpose()

return ((xt.dot(calculated\_hypothesis - y.transpose())) / m) + wt\_restricted\*l/m

def gradient\_scipy(w, x, y, l = 0):

calculated\_hypothesis = hypothesis(x, w.reshape(-1, 1))

m = y.shape[1]

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

w\_restricted[0] = 0

xt = x.transpose()

return ((xt.dot(calculated\_hypothesis - y.reshape(-1, 1)) / m) + w\_restricted\*l/m).flatten()

Приведем также листинг кода, реализующий градиентный спуск по обычным правилам из предыдущих лабораторных работ:

def calculate\_new\_weights(wt, x, y):

calculated\_gradient = gradient(wt, x, y)

return wt - calculated\_gradient.dot(alpha)

def gradient\_descent(x, y):

wt = np.matrix([0.0] \* x.shape[1]).transpose()

its = 0

its\_hist = []

err\_hist = []

while true:

wt = calculate\_new\_weights(wt, x, y)

error = cost(wt, x, y)

its += 1

if its % 10000 == 0:

print('error %s' % error)

its\_hist.append(its)

err\_hist.append(error)

if its % 100000 == 0:

print('error %s' % error)

return wt, its, its\_hist, err\_hist

### 2.6 Построение первой модели

Первая модель линейной регрессии будет построена с использованием самописного градиентного спуска с параметром регуляризации 0. Также, построим график данных и модели:

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

print('w %s' % w)

plt.scatter(np.array(data['X']), np.array(data['y']))

x\_ax = np.linspace(min(np.array(data['X'])) - 5, max(np.array(data['y'])) + 5, 1000)

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

y\_ax = predict(x\_ax, w)

plt.plot(x\_ax[:, 1], y\_ax, linewidth=2)

plt.show()

Получим следующий график:

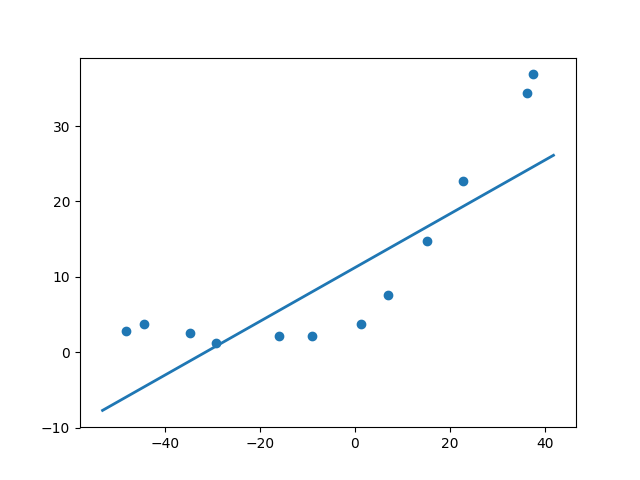


Рисунок 2.6.1. График обучающей выборки и первой модели с параметром регуляризации 0

Получили довольно плохую модель, в которой, к тому же, и не нет регуляризации, так как параметр регуляризации 0. Поэтому регуляризация здесь никак сработать не может.

### 2.7 Построение графика процесса обучения

Приведем листинг кода построения графика:

l = 0

m = x.shape[0]

x\_errors = []

val\_errors = []

for i in range(2, m+1):

xi = x[0:i]

yi = y[:, 0:i]

w0 = np.array([0.0] \* x.shape[1])

w = optimize.fmin\_cg(

f=cost\_scipy,

x0=w0,

fprime=gradient\_scipy,

args=(xi, yi, l)

)

x\_errors.append(cost\_scipy(w, xi, yi))

val\_errors.append(cost\_scipy(w, val\_x, val\_y))

x\_ax = [i for i in range(2, m+1)]

plt.plot(x\_ax, x\_errors, c="g")

plt.plot(x\_ax, val\_errors, c="r")

plt.legend(["Error on training set", "Error on validation set"], loc="best")

plt.show()

Здесь мы смотрим, как изменяется ошибка на обучающей выборке при увеличении количества экземпляров выборки. Также мы смотрим, улучшается ли прогноз на валидационной выборке. График позволяет увидеть, подвержена модель одной из проблем: underfitting или overfitting. Покажем, как выглядит график в нашем случае:

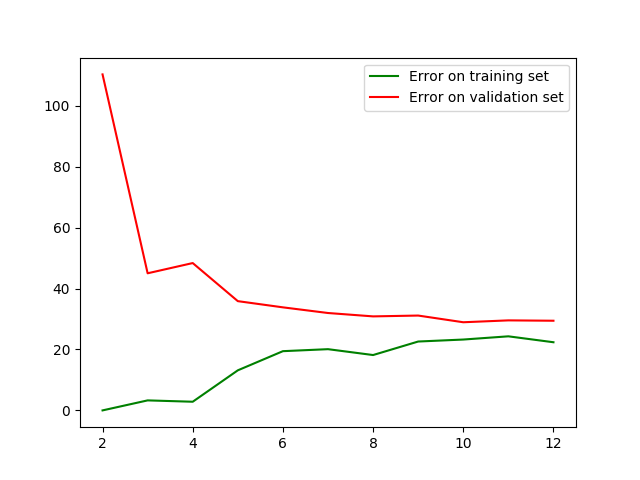


Рисунок 2.7.1. График процесса обучения первой модели с параметром регуляризации 0

На графике видно, что наш алгоритм подвержен проблеме underfitting. На графике это показывает близкое расстояние между кривыми. Это значит, что ошибки на обучающей и валидационной выборках при увеличении количества экземпляров в обучающей выборке очень близко, т.е. в такой модели ошибка на валидационной выборке не будет изменяться дальше: это все, на что способна данная модель, имеющая high bias.

Вторая проблема, которую можно увидеть на таком графике (не на этом) - overfitting. Это выглядит на графике как довольно большая разница между кривыми. В этом случае ошибка на обучающей выборке ожидается низкая - так как модель чересчур сложна и идеально подстраивается под обучающую выборку. В то же время, из-за того, что такая модель плохо обобщается, ошибка на валидационной выборке будет высока. Такую модель, возможно, можно исправить, например, добавляя больше данных, чтобы модели было сложнее под них подстроиться.

### 2.8 Функция добавления p - 1 новых признаков

Реализуем функцию добавления в выборку полиномиальных фич. Функция будет добавлять фичи во всей степенях включительно соответственно некоторому параметру:

def build\_polynomyial\_x(initial\_x, degree):

polynomial\_x = np.zeros(shape=(len(initial\_x), degree))

for i in range(0, degree):

polynomial\_x[:, i] = initial\_x.squeeze() \*\* (i + 1)

return polynomial\_x

В нашем случае мы будем строить полиномиальные фичи следующим образом:

build\_polynomyial\_x(np.array(data['X']), 8)

Это создаст новую выборку, в которой каждая фича - степень оригинальной выборки (от 1 до 8).

### 2.9 Функция нормализации признаком

Ввиду использования полинома высокой степени имеет смысл нормализовать выборку перед обучением (порядки данных чересчур разнятся):

def normalize\_x(x):

means = x.mean(axis=0)

stds = np.std(x, axis=0, ddof=1)

normalized\_x = (x - means) / stds

normalized\_x = np.hstack((np.ones((len(normalized\_x), 1)), normalized\_x))

return normalized\_x, means, stds

def normalize\_x\_with\_params(x, means, stds):

normalized\_x = (x - means) / stds

normalized\_x = np.hstack((np.ones((len(normalized\_x), 1)), normalized\_x))

return normalized\_x

В отличие от предыдущих лабораторных работ, здесь мы делаем feature scaling не по разнице между наибольшим и наименьшим элементов, а по среднеквадратическому отклонению, что чаще используется на практике.

### 2.10 Обучение первой модели с полиномиальными фичами. Переобучение. График данных и модели. График процесса обучения

Обучение модели будет проходить следующим образом:

def build\_polynomial\_approximation(data, y, l):

pol\_x = build\_polynomyial\_x(np.array(data['X']), 8)

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

w = optimize.fmin\_cg(

f=cost\_scipy,

x0=np.zeros(pol\_x.shape[1]),

fprime=gradient\_scipy,

args=(pol\_x, y, l),

maxiter=1000

).reshape(-1, 1)

plt.scatter(pol\_x[:, 1:2], np.array(data['y']))

x\_ax = np.linspace(min(np.array(data['X'])) - 5, max(np.array(data['X'])) + 5, 1000)

# normalize according to X

x\_ax\_polynomial = build\_polynomyial\_x(x\_ax, 8)

x\_ax\_polynomial = normalize\_x\_with\_params(x\_ax\_polynomial, mean\_res, diff\_res)

predictions = predict(x\_ax\_polynomial, w)

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

plt.plot(x\_ax, predictions, linewidth=2)

plt.show()

build\_polynomial\_approximation(data, y, 0)

Здесь мы добавляем новые полиномиальные фичи максимальной степени 8, нормализуем данные, обучаем модель библиотечной реализацией градиентного спуска для более быстрой сходимости и создаем график данных с получившейся моделью:

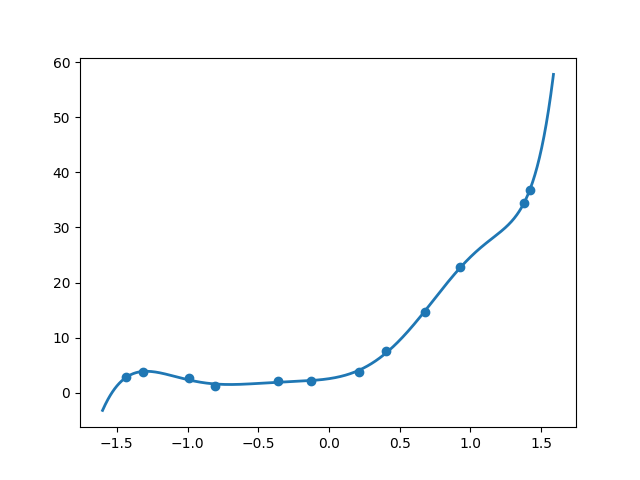


Рисунок 2.10.1. График обучающих данных и модели.

Очень похоже, что модель переобучилась и уже не сможет обобщаться на данные валидационной выборке. Проверим нашу гипотезу, построив график процесса обучения:

def build\_polynomial\_learning\_curves(data, y, val\_y, l):

m = x.shape[0]

pol\_x = build\_polynomyial\_x(np.array(data['X']), 8)

pol\_x, means, stds = normalize\_x(pol\_x)

pol\_val\_x = build\_polynomyial\_x(np.array(data['Xval']), 8)

pol\_val\_x = normalize\_x\_with\_params(pol\_val\_x, means, stds)

x\_errors = []

val\_errors = []

for i in range(2, m + 1):

pol\_xi = pol\_x[0:i]

yi = y[:, 0:i]

w0 = np.array([0.0] \* pol\_x.shape[1])

w = optimize.fmin\_cg(

f=cost\_scipy,

x0=w0,

fprime=gradient\_scipy,

args=(pol\_xi, yi, l)

)

x\_errors.append(cost\_scipy(w, pol\_xi, yi))

val\_errors.append(cost\_scipy(w, pol\_val\_x, val\_y))

x\_ax = [i for i in range(2, m + 1)]

plt.plot(x\_ax, x\_errors, c="g")

plt.plot(x\_ax, val\_errors, c="r")

plt.legend(["Error on training set", "Error on validation set"], loc="best")

plt.show()

build\_polynomial\_learning\_curves(data, y, val\_y, 0)

Код выше строит график процесса обучения: learning curves. Делает он это следующим образом: сначала из обучающей выборки берут 2 экземпляра, затем 3 и т.д. до целой выборки и модель обучается на этих данных. После, вычисляется ошибка и на обучающей, и на валидационной выборке. Далее, строится график с двумя кривыми: зависимости ошибки на обучающей и валидационной выборках от количества экземпляров обучающей выборки:

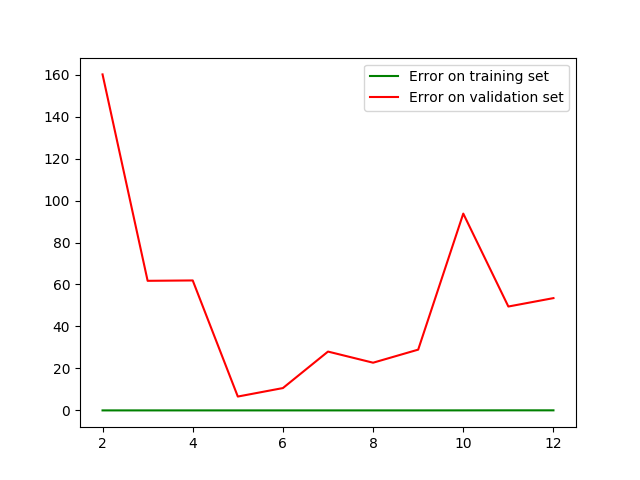


Рисунок 2.10.2. График процесса обучения модели.

Как видим, график подтверждает нашу гипотезу: имеет место переобучение, потому что кривая ошибки на обучающей выборке всегда 0, в то время как кривая ошибок на тестовой выборке всегда довольно высока и не имеет какой-либо закономерности.

### 2.11 Обучение модели с полиномиальными фичами с различными параметрами

Построим аналогичную модель с параметром регуляризации 1:

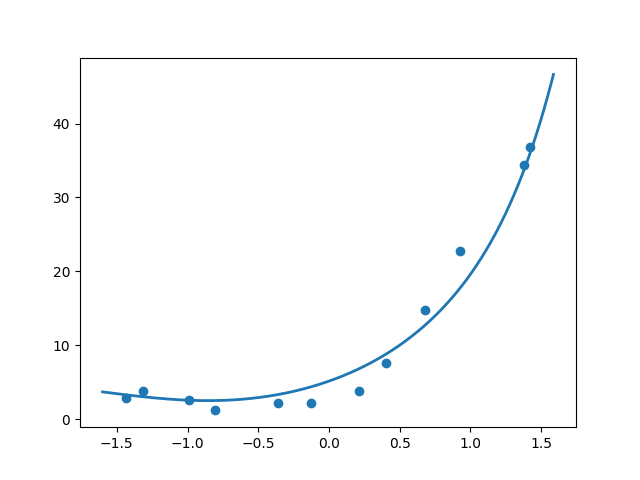


Рисунок 2.11.1. График модели с параметром регуляризации 1.

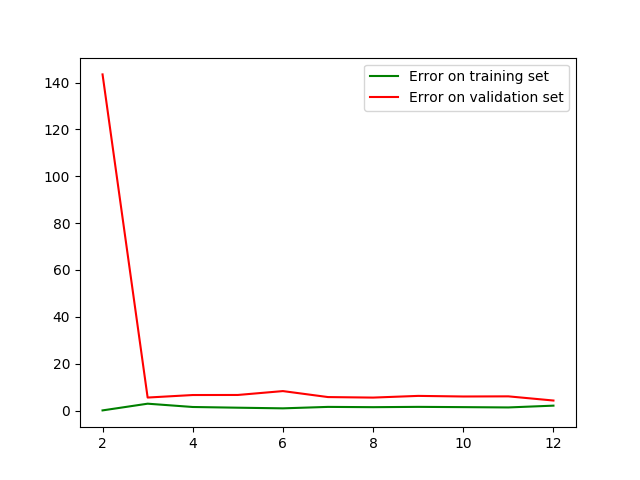


Рисунок 2.11.2. График процесса обучения модели с параметром регуляризации 1.

Из графиков модели и процесса обучения видно, что модель получилась гораздо лучше: конечно, она не так идеально подстраивается под обучающую выборку, но это нам и надо, ведь теперь модель обобщается на валидационную выборку, что видно на графике процесса обучения: с ходом обучения разница между ошибками на обучающей и валидационной выборках становится мала.

Теперь построим такую же модель с параметров регуляризации 100:

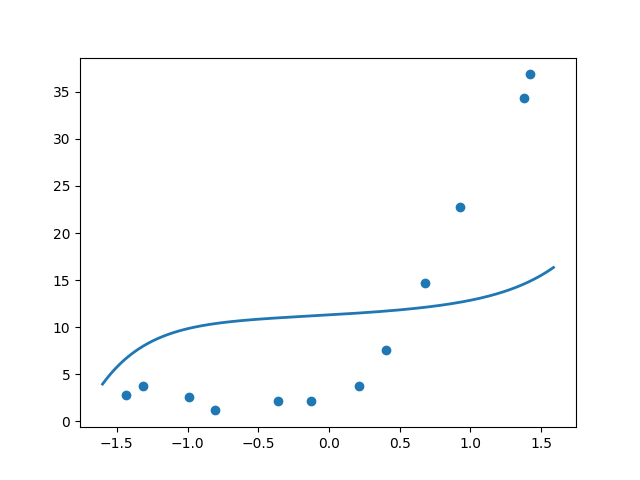


Рисунок 2.11.3. График модели с параметром регуляризации 100. Underfitting

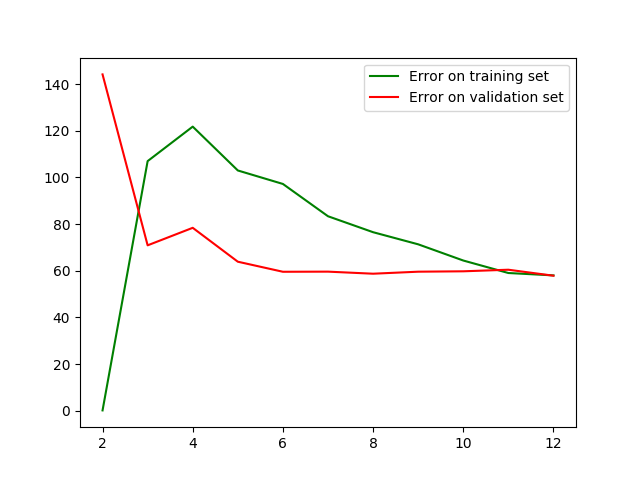


Рисунок 2.11.4. График процесса обучения модели с параметром регуляризации 100. Underfitting

Здесь уже по графику самой модели мы видим, что имеет место underfitting (high bias). Это значит, что из-за высоких штрафов мы настолько упростили модель, что ошибка на обучающей и валидационной выборках стала примерно одинаковая и очень высокая. Такая модель не сможет хорошо апроксимировать наши данные.

### 2.12 Поиск лучшего коэффициента регуляризации

Искать лучший коэффициент регуляризации будем следующим образом:

def hypothesis\_polynomial\_features(data, y, val\_y, l):

pol\_x = build\_polynomyial\_x(np.array(data['X']), 8)

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

w = optimize.fmin\_cg(

f=cost\_scipy,

x0=np.zeros(pol\_x.shape[1]),

fprime=gradient\_scipy,

args=(pol\_x, y, l),

maxiter=1000

).reshape(-1, 1)

pol\_val\_x = build\_polynomyial\_x(np.array(data['Xval']), 8)

# normalize according to X

pol\_val\_x = normalize\_x\_with\_params(pol\_val\_x, mean\_res, diff\_res)

return cost\_scipy(w, pol\_val\_x, val\_y)

lambdas = np.linspace(0.001, 10, 10000)

test\_errors = []

for l in lambdas:

err = hypothesis\_polynomial\_features(data, y, val\_y, l)

test\_errors.append(err)

best\_lambda\_index = np.argmin(test\_errors)

best\_lambda = lambdas[best\_lambda\_index]

min\_error = test\_errors[best\_lambda\_index]

print("best\_lambda %s, min\_error %s" % (best\_lambda, min\_error))

plt.plot(lambdas, test\_errors, c="r")

plt.show()

Здесь мы выбираем 10.000 значений параметра регуляризации в промежутке [0.001; 10], для каждого параметра обучаем аналогичную предыдущим пунктам модель, подсчитываем ошибку на валидационной выборке, выбираем параметр регуляризации с наименьшей ошибкой на валидационной выборке и создаем график ошибки на валидационной выборке в зависимости от параметра регуляризации:

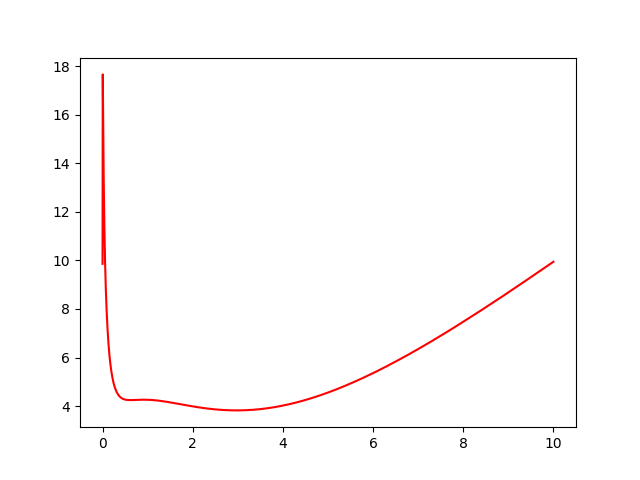


Рисунок 2.12.1. График зависимости ошибки на валидационной выборке от параметра регуляризации

В ходе эксперимента найдено лучше значение параметра: 2.972. В этом случае достигается минимальное значения ошибки: 3.822768364932697.

### 2.13 Ошибка и график модели на контрольной выборке

Построим график модели с параметров регуляризации 2.972 и подсчитаем ошибку на контрольной выборке:

def build\_polynomial\_approximation\_for\_test\_set(data, y, l):

pol\_x = build\_polynomyial\_x(np.array(data['X']), 8)

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

w = optimize.fmin\_cg(

f=cost\_scipy,

x0=np.zeros(pol\_x.shape[1]),

fprime=gradient\_scipy,

args=(pol\_x, y, l),

maxiter=1000

).reshape(-1, 1)

# test set plot

test\_x\_ax = (np.array(data['Xtest']) - mean\_res[0]) / diff\_res[0]

plt.scatter(test\_x\_ax, np.array(data['ytest']))

# predictions plot

x\_ax = np.linspace(min(np.array(data['Xtest'])) - 5, max(np.array(data['Xtest'])) + 5, 1000)

x\_ax\_polynomial = build\_polynomyial\_x(x\_ax, 8)

x\_ax\_polynomial = normalize\_x\_with\_params(x\_ax\_polynomial, mean\_res, diff\_res)

predictions = predict(x\_ax\_polynomial, w)

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

plt.plot(x\_ax, predictions, linewidth=2)

plt.show()

x\_test\_polynomial = build\_polynomyial\_x(np.array(data['Xtest']), 8)

x\_test\_polynomial = normalize\_x\_with\_params(x\_test\_polynomial, mean\_res, diff\_res)

print(cost\_scipy(w, x\_test\_polynomial, np.array(data['ytest'].T)))

График “лучшей” модели получился таким:

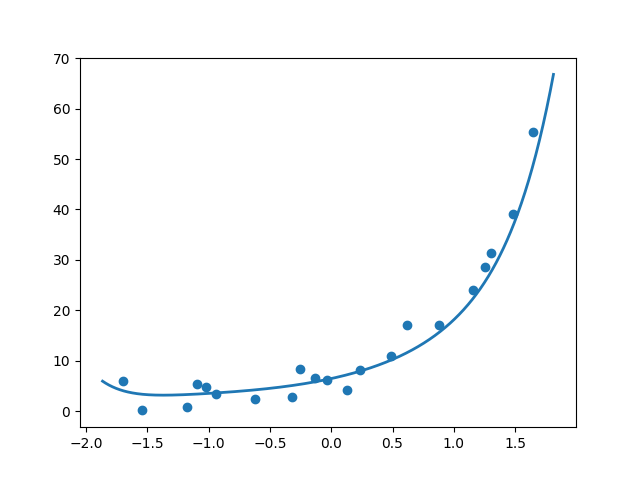


Рисунок 2.13.1. График модели с параметром регуляризации, результирующей в наименьшей ошибки на валидационной тесте

Ошибка на тестовой выборке составляет 3.826192129187544.