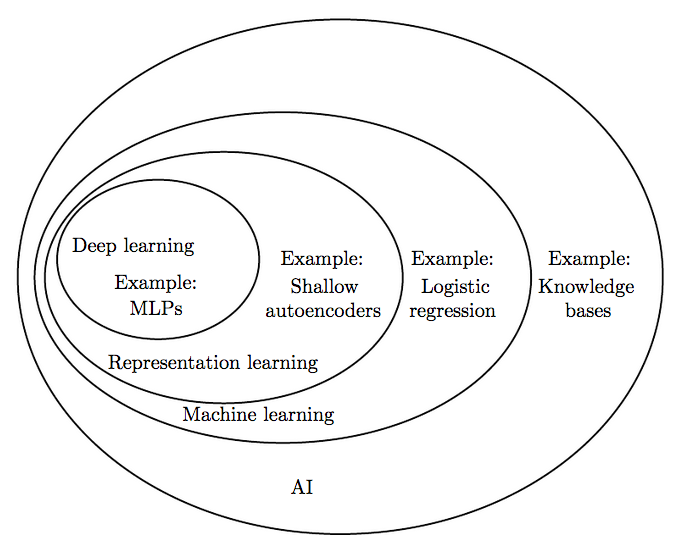
Базовые принципы машинного обучения на примере линейной регрессии

* [Машинное обучение](https://habrahabr.ru/hub/machine_learning/),
* [Математика](https://habrahabr.ru/hub/maths/),
* [Алгоритмы](https://habrahabr.ru/hub/algorithms/),
* [Python](https://habrahabr.ru/hub/python/),
* [Блог компании Open Data Science](https://habrahabr.ru/company/ods/)

Здравствуйте, коллеги! Это блог открытой русскоговорящей [дата саентологической ложи](http://ods.ai/). Нас уже легион, точнее 2500+ человек в слаке. За полтора года мы нагенерили 800к+ сообщений (ради этого слак выделил нам корпоративный аккаунт). Наши люди есть везде и, может, даже в вашей организации. Если вы интересуетесь машинным обучением, но по каким-то причинам не знаете про [Open Data Science](http://ods.ai/), то возможно вы в курсе мероприятий, которые организовывает сообщество. Самым масштабным из них является [DataFest](http://datafest.ru/), который проходил недавно в офисе [Mail.Ru Group](https://corp.mail.ru/ru/press/events/307/), за два дня его посетило 1700 человек. Мы растем, наши ложи открываются в городах России, а также в Нью-Йорке, Дубае и даже во Львове, да, мы не воюем, а иногда даже и употребляем горячительные напитки вместе. И да, мы некоммерческая организация, наша цель — просвещение. Мы делаем все ради искусства. (пс: на фотографии вы можете наблюдать заседание ложи в одном из тайных храмов в Москве).  
  
Мне выпала честь сделать первый пост, и я, пожалуй, отклонюсь от [своей привычной нейросетевой тематики](https://habrahabr.ru/users/mephistopheies/topics/) и сделаю пост о базовых понятиях машинного обучения на примере одной из самых простых и самых полезных моделей — линейной регрессии. Я буду использовать язык питон для демонстрации экспериментов и отрисовки графиков, все это вы с легкостью сможете повторить на своем компьютере. Поехали.

Формализмы



Машинное обучение — это подраздел искусственного интеллекта, в котором изучаются алгоритмы, способные обучаться без прямого программирования того, что нужно изучать. Линейная регрессия является типичным представителем алгоритмов машинного обучения. Для начала ответим на вопрос «а что вообще значит *обучаться*?». Ответ на этот вопрос мы возьмем из [книги 1997 года](http://www.cs.cmu.edu/~tom/mlbook.html) (стоит отметить, что оглавление этой книги не сильно отличается от современных книг по машинному обучению).

Говорят, что программа обучается на опыте E относительно класса задач T в смысле меры качества L, если при решении задачи T качество, измеряемое мерой L, возрастает при демонстрации нового опыта E.

Можно выделить следующие задачи T, решаемые машинным обучением: [обучение с учителем](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%83%D1%87%D0%B5%D0%BD%D0%B8%D0%B5_%D1%81_%D1%83%D1%87%D0%B8%D1%82%D0%B5%D0%BB%D0%B5%D0%BC), [обучение без учителя](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%83%D1%87%D0%B5%D0%BD%D0%B8%D0%B5_%D0%B1%D0%B5%D0%B7_%D1%83%D1%87%D0%B8%D1%82%D0%B5%D0%BB%D1%8F), [обучение с подкреплением](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%83%D1%87%D0%B5%D0%BD%D0%B8%D0%B5_%D1%81_%D0%BF%D0%BE%D0%B4%D0%BA%D1%80%D0%B5%D0%BF%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5%D0%BC), [активное обучение](https://en.wikipedia.org/wiki/Active_learning_(machine_learning)), [трансфер знаний](https://en.wikipedia.org/wiki/Inductive_transfer) и т.д. Регрессия (как и классификация) относится к классу задач обучения с учителем, когда по заданному набору признаков наблюдаемого объекта необходимо спрогнозировать некоторую целевую переменную. Как правило, в задачах обучения с учителем, опыт E представляется в виде множества пар признаков и целевых переменных: D={(xi,yi)}i=1,…,n. В случае линейной регрессии признаковое описание объекта — это действительный вектор x→∈Rm, а целевая переменная — это скаляр y∈R. Самой простой мерой качества L для задачи регрессии является L(y,y^)=(y−y^)2, где y^ — это наша оценка реального значения целевой переменной.   
  
У нас есть задача, данные и способ оценки программы/модели. Давайте определим, что такое *модель*, и что значит *обучить модель*. *Предиктивная модель* – это параметрическое семейство функций (семейство гипотез):

H={h(x,θ)|θ∈Θ}

где

* h:X×Θ→Y
* Θ — множество параметров

Таким образом, из большого семейства гипотез мы должны выбрать какую-то одну конкретную, которая с точки зрения меры L является лучшей. Процесс такого выбора назовем *алгоритмом обучения*:

M:(X×Y)n→H

Получается, что алгоритм обучения — это отображение из набора данных в пространство гипотез. Обычно процесс обучения с учителем состоит из двух шагов:

1. обучение: h=M(D);
2. применение: y^=h(x).

Часто для обучения модели пользуются *принципом минимизации эмпирического риска*. Риском гипотезы h называют ожидаемое значение функции стоимости L:

Q(h)=Ex,y∼P(x,y)[L(h(x),y)]=∫L(h(x),y)dP(x,y)

Но, к сожалению, такой интеграл не посчитать, т.к. распределение P(x,y) неизвестно, иначе и задачи не было бы. Но мы можем посчитать эмпирическую оценку риска, как среднее значение функции стоимости:

Qemp(h)=1n∑i=1nL(h(xi),yi)

Тогда, согласно принципу минимизации эмпирического риска, мы должны выбрать такую гипотезу h∈H, которая минимизирует Qemp:

h^=arg⁡minh∈HQemp(h)

У данного принципа есть существенный недостаток, решения найденные таким путем будут склонны к *переобучению*. Мы говорим, что модель обладает *обобщающей способностью*, тогда, когда ошибка на новом (тестовом) наборе данных (взятом из того же распределения P(x,y)) мала, или же предсказуема. Переобученная модель не обладает обобщающей способностью, т.е. на обучающем наборе данных ошибка мала, а на тестовом наборе данных ошибка существенно больше.

Линейная регрессия

Давайте ограничим пространство гипотез только линейными функциями от m+1 аргумента, будем считать, что нулевой признак для всех объектов равен единице x0=1:

∀h∈H,h(x→)=w0x0+w1x1+w2x2+⋯+wmxm=∑i=0mwixi=x→Tw→

Эмпирический риск (функция стоимости) принимает форму среднеквадратичной ошибки:

L(X,y→,w→)=12n∑i=1n(yi−x→iTw→i)2=12n‖y→−Xw→‖22=12n(y→−Xw→)T(y→−Xw→)

строки матрицы X — это признаковые описания наблюдаемых объектов. Один из алгоритмов обучения M такой модели — это метод наименьших квадратов. Вычислим производную функции стоимости:

∂L∂w→=∂∂w→12n(y→Ty→−2y→TXw→+w→TXTXw→)=12n(−2XTy→+2XTXw→)

приравняем к нулю и найдем решение в явном виде:

∂L∂w→=0⇔12n(−2XTy→+2XTXw→)=0⇔−XTy→+XTXw→=0⇔XTXw→=XTy→⇔w→=(XTX)−1XTy→

Поздравляю, дамы и господа, мы только что с вами вывели алгоритм машинного обучения. Реализуем же этот алгоритм. Начнем с датасета, состоящего всего из одного признака. Будем брать случайную точку на синусе и добавлять к ней шум — таким образом получим целевую переменную; признаком в этом случае будет координата x:

def generate\_wave\_set(n\_support=1000, n\_train=25, std=0.3):

data = {}

*# выберем некоторое количество точек из промежутка от 0 до 2\*pi*

data['support'] = np.linspace(0, 2\*np.pi, num=n\_support)

*# для каждой посчитаем значение sin(x) + 1*

*# это будет ground truth*

data['values'] = np.sin(data['support']) + 1

*# из support посемплируем некоторое количество точек с возвратом, это будут признаки*

data['x\_train'] = np.sort(np.random.choice(data['support'], size=n\_train, replace=True))

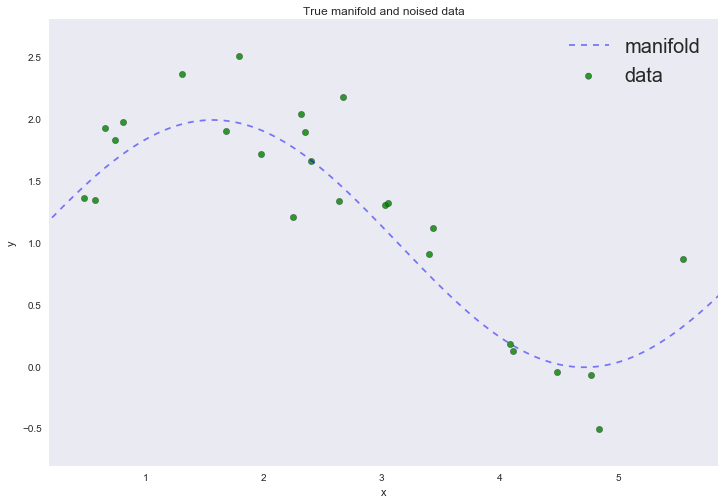
*# опять посчитаем sin(x) + 1 и добавим шум, получим целевую переменную*

data['y\_train'] = np.sin(data['x\_train']) + 1 + np.random.normal(0, std, size=data['x\_train'].shape[0])

return data

data = generate\_wave\_set(1000, 250)

Отрисовка графика

  
  
А теперь реализуем алгоритм обучения, используя магию NumPy:

*# добавим колонку единиц к единственному столбцу признаков*

X = np.array([np.ones(data['x\_train'].shape[0]), data['x\_train']]).T

*# перепишем, полученную выше формулу, используя numpy*

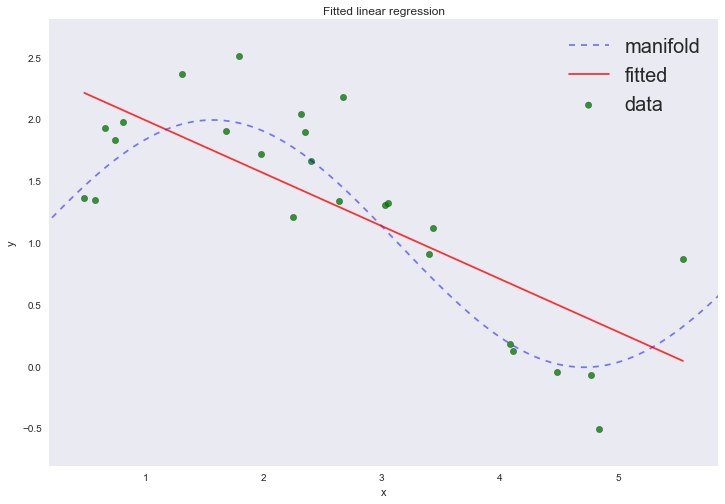
*# шаг обучения - в этом шаге мы ищем лучшую гипотезу h*

w = np.dot(np.dot(np.linalg.inv(np.dot(X.T, X)), X.T), data['y\_train'])

*# шаг применения: посчитаем прогноз*

y\_hat = np.dot(w, X.T)

Отрисовка графика

  
  
Как мы видим, линия не очень-то совпадает с настоящей кривой. Среднеквадратичная ошибка равна **0.26704** условных единиц. Очевидно, что если бы вместо линии мы использовали кривую третьего порядка, то результат был бы куда лучше. И, на самом деле, с помощью линейной регрессии мы можем обучать нелинейные модели.

Полиномиальная регрессия

В линейной регрессии мы ограничивали пространство гипотез только линейными функциями от признаков. Давайте теперь расширим пространство гипотез до всех полиномов степени p. Тогда в нашем случае, когда количество признаков равно одному m=1, пространство гипотез будет выглядеть следующим образом:

∀h∈H,h(x)=w0+w1x+w1x2+⋯+wnxp=∑i=0pwixi

Если заранее предрассчитать все степени признаков, то задача опять сводится к описанному выше алгоритму — методу наименьших квадратов. Попробуем отрисовать графики нескольких полиномов разных степеней.

*# список степеней p полиномов, который мы протестируем*

degree\_list = [1, 2, 3, 5, 7, 10, 13]

cmap = plt.get\_cmap('jet')

colors = [cmap(i) for i in np.linspace(0, 1, len(degree\_list))]

margin = 0.3

plt.plot(data['support'], data['values'], 'b--', alpha=0.5, label='manifold')

plt.scatter(data['x\_train'], data['y\_train'], 40, 'g', 'o', alpha=0.8, label='data')

w\_list = []

err = []

for ix, degree in enumerate(degree\_list):

*# список с предрасчитанными степенями признака*

dlist = [np.ones(data['x\_train'].shape[0])] + \

map(lambda n: data['x\_train']\*\*n, range(1, degree + 1))

X = np.array(dlist).T

w = np.dot(np.dot(np.linalg.inv(np.dot(X.T, X)), X.T), data['y\_train'])

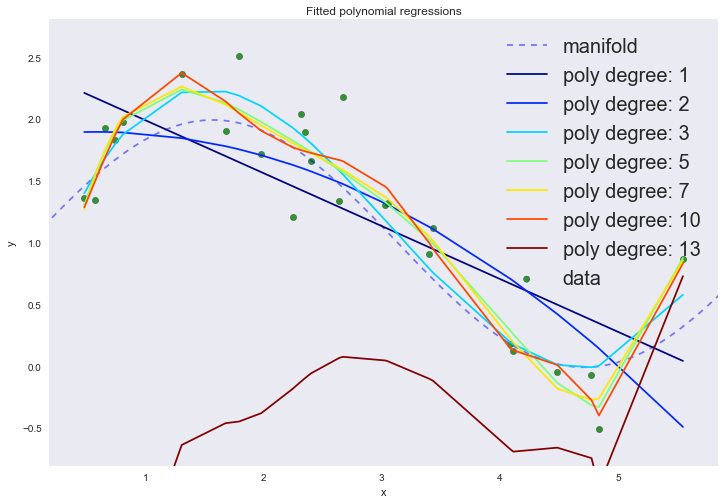
w\_list.append((degree, w))

y\_hat = np.dot(w, X.T)

err.append(np.mean((data['y\_train'] - y\_hat)\*\*2))

plt.plot(data['x\_train'], y\_hat, color=colors[ix], label='poly degree: %i' % degree)

Отрисовка графика

  
  
На графике мы можем наблюдать сразу два феномена. Пока не обращайте внимание на 13-ую степень полинома. При увеличении степени полинома, средняя ошибка продолжает уменьшаться, хотя мы вроде были уверены, что именно кубический полином должен лучше всего описывать наши данные.

|  |  |
| --- | --- |
| **p** | **error** |
| 1 | 0.26704 |
| 2 | 0.22495 |
| 3 | 0.08217 |
| 5 | 0.05862 |
| 7 | 0.05749 |
| 10 | 0.0532 |
| 13 | 5.76155 |

Это явный признак переобучения, который можно заметить по визуализации даже не используя тестовый набор данных: при увеличении степени полинома выше третьей модель начинает *интерполировать* данные, вместо *экстраполяции*. Другими словами, график функции проходит точно через точки из тренировочного набора данных, причем чем выше степень полинома, тем через большее количество точек он проходит. Степень полинома отражает *сложность* модели. Таким образом, сложные модели, у которых степеней свободы достаточно много, могут попросту запомнить весь тренировочный набор, полностью теряя обобщающую способность. Это и есть проявление негативной стороны принципа минимизации эмпирического риска.  
  
Вернемся к полиному 13-ой степени, с ним явно что-то не так. По идее, мы ожидаем, что полином 13-ой степени будет описывать тренировочный набор данных еще лучше, но результат показывает, что это не так. Из курса линейной алгебры мы помним, что обратная матрица существует только для несингулярных матриц, т.е. тех, у которых нет линейной зависимости колонок или строк. В методе наименьших квадратов нам необходимо инвертировать следующую матрицу: (XTX)−1. Для тестирования на линейную зависимость или *мультиколлинеарность* можно использовать [*число обусловленности матрицы*](https://ru.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D0%BB%D0%BE_%D0%BE%D0%B1%D1%83%D1%81%D0%BB%D0%BE%D0%B2%D0%BB%D0%B5%D0%BD%D0%BD%D0%BE%D1%81%D1%82%D0%B8). Один из способов оценки этого числа для матриц — это отношение модуля максимального собственного числа матрицы к модулю минимального собственного числа. Большое число обусловленности матрицы, или же наличие одного или нескольких собственных чисел близких к нулю свидетельствует о наличии мультиколлинеарности (или нечеткой мультиколлиниарности, когда ci≈kcj+b). Такие матрицы называются слабо обусловленными, а задача — некорректно поставленной. При инвертировании такой матрицы, решения имеют большую дисперсию. Это проявляется в том, что при небольшом изменении начальной матрицы, инвертированные будут сильно отличаться друг от друга. На практике это всплывет тогда, когда к 1000 семплов, вы добавите всего один, а решение МНК будет совсем другим. Посмотрим на собственные числа полученной матрицы, нас там ждет сюрприз:

np.linalg.eigvals(np.cov(X[:, 1:].T))

Out[10]:

array([

9.29965299e+17+0.j , 4.04567033e+13+0.j ,

5.44657111e+09+0.j , 3.54104756e+06+0.j ,

8.36745166e+03+0.j , 6.82745279e+01+0.j ,

8.88434986e-01+0.j , 2.42827315e-02+0.00830052j,

2.42827315e-02-0.00830052j, 1.17621840e-03+0.j ,

1.72254789e-04+0.j , -5.68384880e-06+0.j ,

2.39611454e-07+0.j ])



Все так, numpy вернул два комплекснозначных собственных значения, что идет вразрез с теорией. Для симметричных и положительно определенных матриц (каковой и является матрица XTX) все собственные значения должны быть действительные. Возможно, это произошло из-за того, что при работе с большими числами матрица стала слегка несимметричной, *но это не точно* ¯\\_(ツ)\_/¯. Если вы вдруг найдете причину такого поведения нумпая, пожалуйста, напишите в комменте.  
  
**UPDATE** (один из членов ложи по имени Андрей Оськин, с ником в слаке skoffer, без аккаунта на хабре, подсказывает):

Есть только одно замечание — не надо пользоваться формулой `(X^T X^{-1}) X^T` для вычисления коэффициентов линейной регрессии. Проблема с расходящимися значениями хорошо известна и на практике используют `QR` или `SVD`.  
  
Ну, то есть вот такой кусок кода даст вполне приличный результат:

degree = 13

dlist = [np.ones(data['x\_train'].shape[0])] + \

list(map(lambda n: data['x\_train']\*\*n, range(1, degree + 1)))

X = np.array(dlist).T

q, r = np.linalg.qr(X)

y\_hat = np.dot(np.dot(q, q.T), data['y\_train'])

plt.plot(data['x\_train'], y\_hat, label='poly degree: %i' % degree)

Перед тем как перейти к следующему разделу, давайте посмотрим на амплитуду параметров полиномиальной регрессии. Мы увидим, что при увеличении степени полинома, размах значений коэффициентов растет чуть ли не экспоненциально. Да, они еще и скачут в разные стороны.

Визуализация коэффициентов

L2 Регуляризация

Регуляризация — это способ уменьшить сложность модели чтобы предотвратить переобучение или исправить некорректно поставленную задачу. Обычно это достигается добавлением некоторой априорной информации к условию задачи. Например так:

Lreg(X,y→,w→)=L(X,y→,w→)+λR(w→)

* λ — это коэффициент регуляризации, то, насколько сильно мы хотим учитывать условие R

На графиках мы увидели, что амплитуда значений коэффициентов слишком большая, попробуем ее уменьшить, добавив ограничение на L2 норму вектора параметров. 

R(w→)=12‖w→‖22=12∑j=1mwj2=12w→Tw→

Новая функция стоимости примет вид:

L(X,y→,w→)=12(y→−Xw→)T(y→−Xw→)+λ2w→Tw→

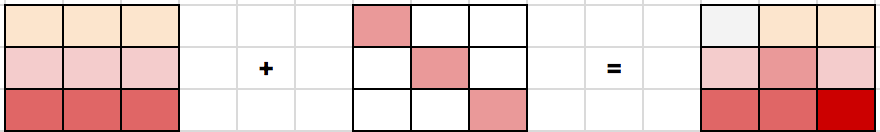
Вычислим производную по параметрам:

∂L∂w→=∂∂w→(12(y→−Xw→)T(y→−Xw→)+λ2w→Tw→)=∂∂w→(12(y→Ty→−2y→TXw→+w→TXTXw→)+λ2w→Tw→)=−XTy→+XTXw→+λw→

И найдем решение в явном виде:

∂L∂w→=0⇔−XTy→+XTXw→+λw→=0⇔XTXw→+λw→=XTy→⇔(XTX+λE)w→=XTy→⇔w→=(XTX+λE)−1XTy→

* E — единичная диагональна матрица

Такая регрессия называется гребневой регрессией (ridge regression). А гребнем является как раз диагональная матрица которую мы прибавляем к матрицеXTX с линейнозависимыми колонками, в результате получаемая матрица не сингулярна.  
  
  
  
Для такой матрицы число обусловленности будет равно: emax+λemin+λ, где ex — это собственные числа матрицы. Таким образом, увеличивая параметр регуляризации мы уменьшаем число обусловленности, а обусловленность задачи улучшается.

*# define regularization parameter*

lmbd = 0.1

degree\_list = [1, 2, 3, 10, 12, 13]

cmap = plt.get\_cmap('jet')

colors = [cmap(i) for i in np.linspace(0, 1, len(degree\_list))]

margin = 0.3

plt.plot(data['support'], data['values'], 'b--', alpha=0.5, label='manifold')

plt.scatter(data['x\_train'], data['y\_train'], 40, 'g', 'o', alpha=0.8, label='data')

w\_list\_l2 = []

err = []

for ix, degree in enumerate(degree\_list):

dlist = [[1]\*data['x\_train'].shape[0]] + map(lambda n: data['x\_train']\*\*n, range(1, degree + 1))

X = np.array(dlist).T

w = np.dot(np.dot(np.linalg.inv(np.dot(X.T, X) + lmbd\*np.eye(X.shape[1])), X.T), data['y\_train'])

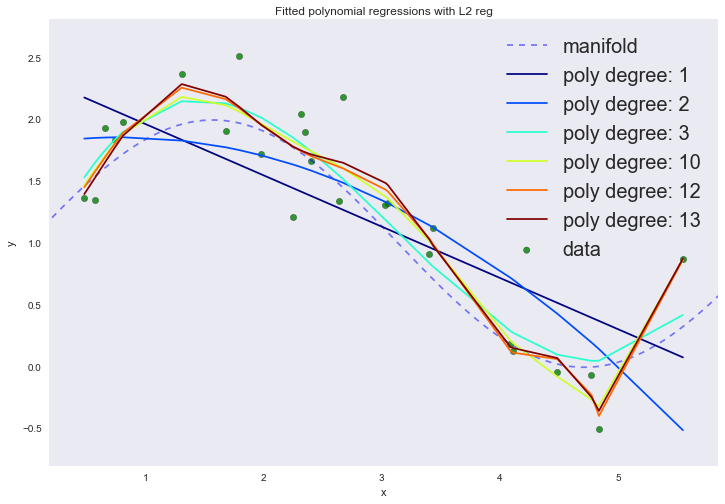
w\_list\_l2.append((degree, w))

y\_hat = np.dot(w, X.T)

plt.plot(data['x\_train'], y\_hat, color=colors[ix], label='poly degree: %i' % degree)

err.append(np.mean((data['y\_train'] - y\_hat)\*\*2))

Отрисовка графика



|  |  |
| --- | --- |
| **p** | **error** |
| 1 | 0.26748 |
| 2 | 0.22546 |
| 3 | 0.08803 |
| 10 | 0.05833 |
| 12 | 0.05585 |
| 13 | 0.05638 |

В результате даже 13-ая степень ведет себя так, как мы ожидаем. Графики немного сгладились, хотя мы все равно наблюдаем небольшое переобучение на степенях выше третьей, что выражается в интерполяции данных в правой части графика. 

Визуализация коэффициентов

Амплитуда коэффициентов также изменилась, хотя скакать в разные стороны они не перестали. Мы помним, что полином третьей степени должен лучше всего описывать наши данные, хотелось бы, чтобы в результате регуляризации все коэффициенты при полиномиальных признаках степени выше третьей были равны нулю. И, оказывается, есть и такой регуляризатор.

L1 регуляризация

Попробуем теперь ограничить вектор параметров модели, используя L1 норму:

R(w→)=‖w→‖1=∑j=1m|wj|

Тогда задача примет вид:

L(X,y→,w→)=12n∑i=1n(xi→Tw→−yi)2+λ∑j=1m|wj|

Посчитаем производную по параметрам модели (надеюсь уважаемые господа не будут пинать меня, за то, что я вжух и взял производную по модулю):

∂L∂wj=1n∑i=1n(xi→Tw→−yi)xi→+λsign(wj)

К сожалению, такая задача не имеет решения в явном виде. Для поиска хорошего приближенного решения мы воспользуемся методом [градиентного спуска](https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D0%B4%D0%B8%D0%B5%D0%BD%D1%82%D0%BD%D1%8B%D0%B9_%D1%81%D0%BF%D1%83%D1%81%D0%BA), тогда формула обновления весов примет вид:

w→new:=w→−α∂L∂w→

а в задаче появляется еще один гиперпараметр α, отвечающий за скорость спуска, его в машинном обучении называют скоростью обучения (learning rate).   
Запрограммировать такой алгоритм не составит труда, но нас ждет еще один сюрприз:

lmbd = 1

degree = 13

dlist = [[1]\*data['x\_train'].shape[0]] + map(lambda n: data['x\_train']\*\*n, range(1, degree + 1))

X = np.array(dlist).T

*# функция для вычисления среднеквадратичное ошибки*

def mse(u, v):

return ((u - v)\*\*2).sum()/u.shape[0]

*# начальное приближение*

w = np.array([-1.0] \* X.shape[1])

*# максимальное количество итераций*

n\_iter = 20

*# сделаем скорость обучения очень маленькой, на всякий случай*

lr = 0.00000001

loss = []

for ix in range(n\_iter):

w -= lr\*(np.dot(np.dot(X, w) - data['y\_train'], X)/X.shape[0] + lmbd\*np.sign(w))

y\_hat = np.dot(X, w)

loss.append(mse(data['y\_train'], y\_hat))

print loss[-1]

Получим такую вот эволюцию ошибки:

1.3051230958e+38

1.21979102398e+58

1.14003816725e+78

1.06549974318e+98

9.95834819687e+117

9.30724755635e+137

8.69871743413e+157

8.12997446782e+177

7.59841727794e+197

7.10161456943e+217

6.63729401109e+237

6.20333184222e+257

5.79774315864e+277

5.41867283397e+297

inf

inf

inf

inf

inf

inf

Даже при такой небольшой скорости обучения, ошибка все равно растет и очень даже стремительно. Причина в том, что каждый признак измеряется в разных масштабах, от небольших чисел у полиномиальных признаков 1-2 степени, до огромных при 12-13 степени. Для того чтобы итеративный процесс сошелся, необходимо либо выбрать экстремально мелкую скорость обучения, либо каким-то образом нормализовать признаки. Применим следующее преобразование к признакам и попробуем запустить процесс еще раз:

μ¯⋅j=1n∑i=1nxijσ¯⋅j=1n∑i=1n(xij−μ¯⋅j)2

x→new=x→−μ¯σ¯

Такое преобразование называется стандартизацией, распределение каждого признака теперь имеет нулевое матожидание и единичную дисперсию.

lmbd = 1

degree = 13

dlist = [[1]\*data['x\_train'].shape[0]] + map(lambda n: data['x\_train']\*\*n, range(1, degree + 1))

X = np.array(dlist).T

*# вычислим выборочное среднее каждого признака*

x\_mean = X.mean(axis=0)

*# вычислим выборочное стандартное отклонение признаков*

x\_std = X.std(axis=0)

*# применим преобразование*

X = (X - x\_mean)/x\_std

X[:, 0] = 1.0

w = np.array([-1.0] \* X.shape[1])

n\_iter = 100

lr = 0.1

loss = []

for ix in range(n\_iter):

w -= lr\*(np.dot(np.dot(X, w) - data['y\_train'], X)/X.shape[0] + lmbd\*np.sign(w))

y\_hat = np.dot(X, w)

loss.append(mse(data['y\_train'], y\_hat))

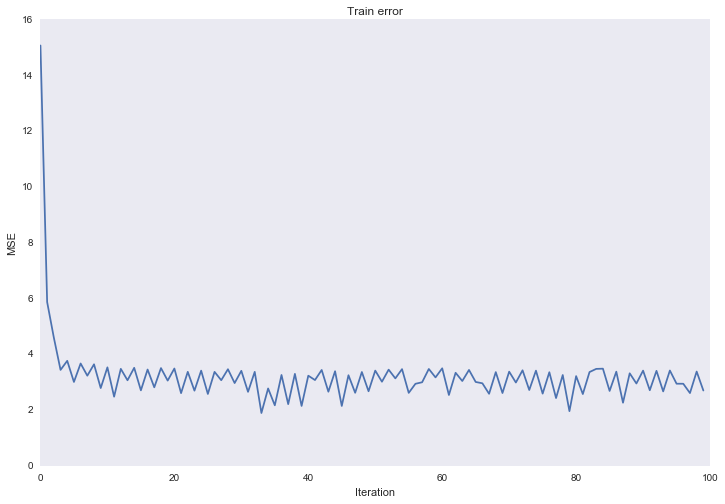
plt.plot(loss)

plt.title('Train error')

plt.xlabel('Iteration')

plt.ylabel('MSE')

plt.show()

Все стало сильно лучше.  
  
  
Нарисуем теперь все графики:

degree\_list = [1, 2, 3, 10, 12, 13]

cmap = plt.get\_cmap('jet')

colors = [cmap(i) for i in np.linspace(0, 1, len(degree\_list))]

margin = 0.3

plt.plot(data['support'], data['values'], 'b--', alpha=0.5, label='manifold')

plt.scatter(data['x\_train'], data['y\_train'], 40, 'g', 'o', alpha=0.8, label='data')

def mse(u, v):

return ((u - v)\*\*2).sum()/u.shape[0]

def fit\_lr\_l1(X, y, lmbd, n\_iter=100, lr=0.1):

w = np.array([-1.0] \* X.shape[1])

loss = []

for ix\_iter in range(n\_iter):

w -= lr\*(np.dot(np.dot(X, w) - y, X)/X.shape[0] +lmbd\*np.sign(w))

y\_hat = np.dot(X, w)

loss.append(mse(y, y\_hat))

return w, y\_hat, loss

w\_list\_l1 = []

for ix, degree in enumerate(degree\_list):

dlist = [[1]\*data['x\_train'].shape[0]] + map(lambda n: data['x\_train']\*\*n, range(1, degree + 1))

X = np.array(dlist).T

x\_mean = X.mean(axis=0)

x\_std = X.std(axis=0)

X = (X - x\_mean)/x\_std

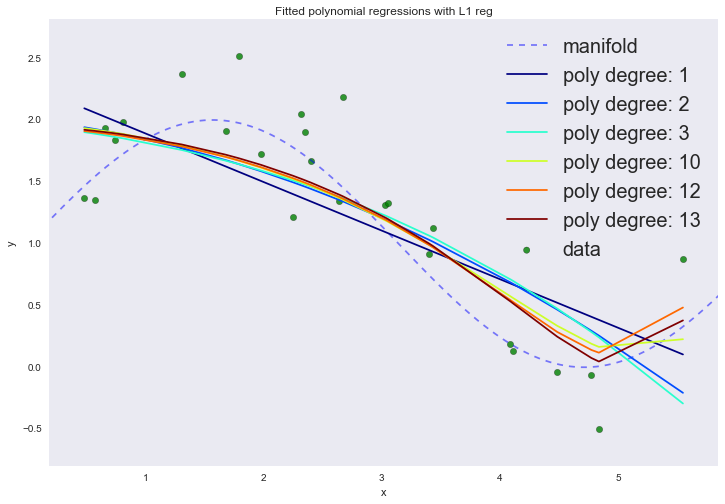
X[:, 0] = 1.0

w, y\_hat, loss = fit\_lr\_l1(X, data['y\_train'], lmbd=0.05)

w\_list\_l1.append((degree, w))

plt.plot(data['x\_train'], y\_hat, color=colors[ix], label='poly degree: %i' % degree)

Отрисовка графика



|  |  |
| --- | --- |
| **p** | **error** |
| 1 | 0.27204 |
| 2 | 0.23794 |
| 3 | 0.24118 |
| 10 | 0.18083 |
| 12 | 0.16069 |
| 13 | 0.15425 |

Если посмотреть на коэффициенты, мы увидим, что большая часть из них близка к нулю (то, что у 13-ой степени коэффициент совсем не нулевой, можно списать на шум и малое количество примеров в обучающей выборке; так же стоит помнить, что теперь все признаки измеряются в одинаковых шкалах).

Визуализация коэффициентов

Описанный способ построения регрессии называется LASSO регрессия. Очень хотелось бы думать, что дядька на коне бросает веревку и ворует коэффициенты, а на их месте остается нуль. Но нет, LASSO = least absolute shrinkage and selection operator.

Байесовская интерпретация линейной регрессии

Две вышеописанные регуляризации, да и сама лининейная регрессия с квадратичной функцией ошибки, могут показаться какими-то грязными эмпирическими трюками. Но, оказывается, если взглянуть на эту модель с другой точки зрения, с точки зрения байесовой статистики, то все становится по местам. Грязные эмпирические трюки станут априорными предположениями. В основе байесовой статистики находится формула Байеса:

p(y∣x)=p(x∣y)p(y)p(x)

* p(y) — априорные ожидания (prior): насколько правдоподобна гипотеза перед наблюдением данных;
* p(x∣y) — правдоподобие (likelihood): насколько правдоподобны данные при условии того, что гипотеза верна;
* p(x)=∑zp(x∣z)p(z) — маргинальная вероятность (marginal probability или evidence): вероятность данных, усредненная по всевозможным гипотезам;
* p(y∣x) — апостериорное распределение (posterior): насколько правдоподобна гипотеза при наблюдаемых данных.

В статистике обычно ищут точечную оценку максимума правдоподобия (ML = maximum likelihood):

θML=arg⁡maxθp(D∣θ)

В то время как в байесовом подходе интересуются апостериорным распределением:

p(θ∣D)∝p(D∣θ)p(θ)

Часто получается так, что интеграл, полученный в результате байесового вывода, крайне нетривиален (в случае линейной регрессии это, к счастью, не так), и тогда нужна точечная оценка. Тогда мы интересуемся максимумом апостериорного распределения (MAP = maximum a posteriori):

θMAP=arg⁡maxθp(θ∣D)=arg⁡maxθp(D∣θ)p(θ)

Давайте сравним ML и MAP гипотезы для линейной регрессии, это даст нам четкое понимание смысла регуляризаций. Будем считать, что все объекты из обучающей выборки были взяты из общей популяции независимо и равномерно распределенно. Это позволит нам записать совместную вероятность данных (правдоподобие) в виде:

p(D)=∏i=1np(xi)

А также будем считать, что целевая переменная подчиняется следующему закону:

y=w→Tx→+ϵ,ϵ∼N(0,σ2)

или

p(y∣x→,w→,σ2)=N(y∣w→Tx→,σ2)

Т.е. верное значение целевой переменной складывается из значения детерминированной линейной функции и некоторой непрогнозируемой случайной ошибки, с нулевым матожиданием и некоторой дисперсией. Тогда, мы можем записать правдоподобие данных как:

p(y→∣X,w→,σ2)=∏i=1nN(yi∣w→Tx→i,σ2)

удобнее будет прологарифмировать это выражение:

L=ln⁡p(y→∣X,w→,σ2)=ln⁡∏i=1nN(yi∣w→Tx→i,σ2)=ln⁡1(σ2π)ne−12σ2∑i=1n(yi−w→Tx→i)2=−n2ln⁡2πσ2−12σ2∑i=1n(yi−w→Tx→i)2

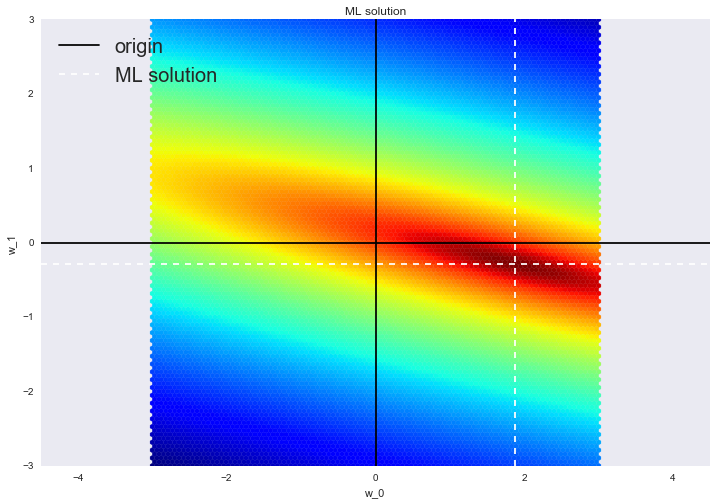
И внезапно мы увидим, что оценка, полученная методом максимального правдоподобия, – это то же самое, что и оценка, полученная методом наименьших квадратов. Сгенерируем новый набор данных большего размера, найдем ML решение и визуализируем его.

data = generate\_wave\_set(1000, 100)

X = np.vstack((np.ones(data['x\_train'].shape[0]), data['x\_train'])).T

w = np.dot(np.dot(np.linalg.inv(np.dot(X.T, X)), X.T), data['y\_train'])

Отрисовка графика

  
  
По оси абсцисс и ординат отложены различные значения всех двух параметров модели (решаем именно линейную регрессию, а не полиномиальную), цвет фона пропорционален значению правдоподобия в соответствующей точке значений параметров. ML решение находится на самом пике, где правдоподобие максимально.  
  
Найдем MAP оценку параметров линейной регрессии, для этого придется задать какое-нибудь априорное распределение на параметры модели. Пусть для начала это будет опять нормальное распределение: p(w→)=N(w→∣0,σ02E).

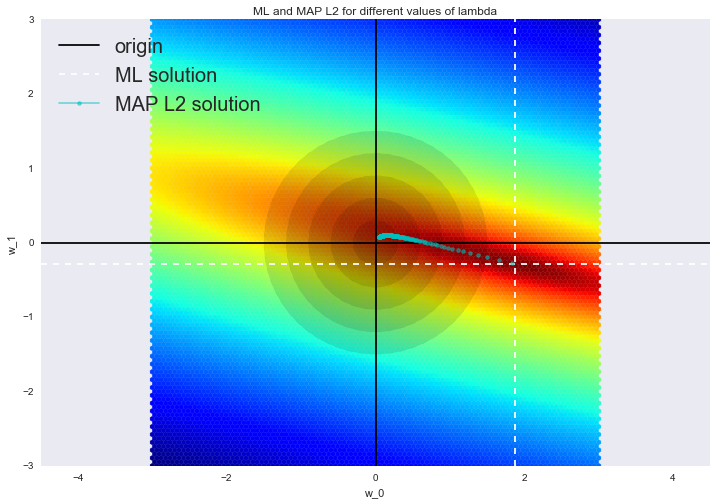
Нормальное распределение

Тогда апостериорное распределение примет вид:

p(w→∣y→,X,σ2)∝N(w→∣0,σ02E)∏i=1nN(yi∣w→Tx→i,σ2)

Если расписать логарифм этого выражения, то вы легко увидите, что добавление нормального априорного распределения — это то же самое, что и добавление L2 нормы к функции стоимости. Попробуйте сделать это сами. Также станет ясно, что варьируя регуляризационный параметр, мы изменяем дисперсию априорного распределения: λ=12σ02. 

Отрисовка графика

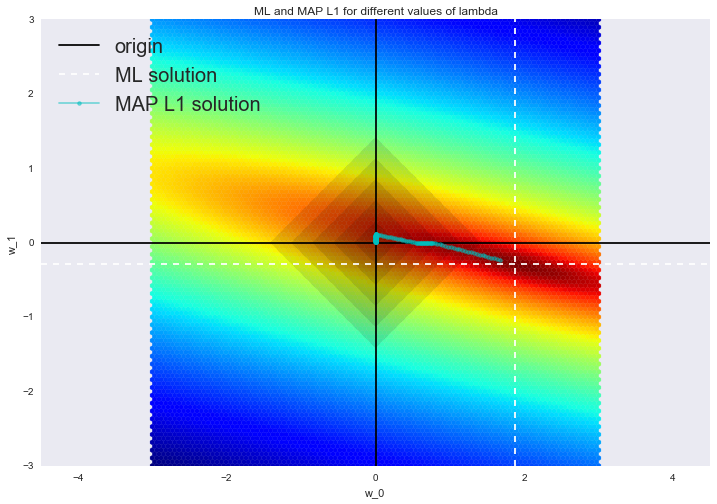
  
  
Теперь на график добавились круги, исходящие от центра — это плотность априорного распределения (круги, а не эллипсы из-за того, что матрица ковариации данного нормального распределения диагональна, а на диагонали находится одно и то же число). Точками обозначены различные решения MAP задачи. При увеличении параметра регуляризации (что эквивалентно уменьшению дисперсии), мы заставляем решение отдаляться от ML оценки и приближаться к центру априорного распределения. При большом значении параметра регуляризации, все параметры будут близки к нулю.  
  
Естественно мы можем наложить и другое априорное распределение на параметры модели, например распределение Лапласа, тогда получим то же самое, что и при L1 регуляризации. 

Распределение Лапласа

Тогда апостериорное распределение примет вид:

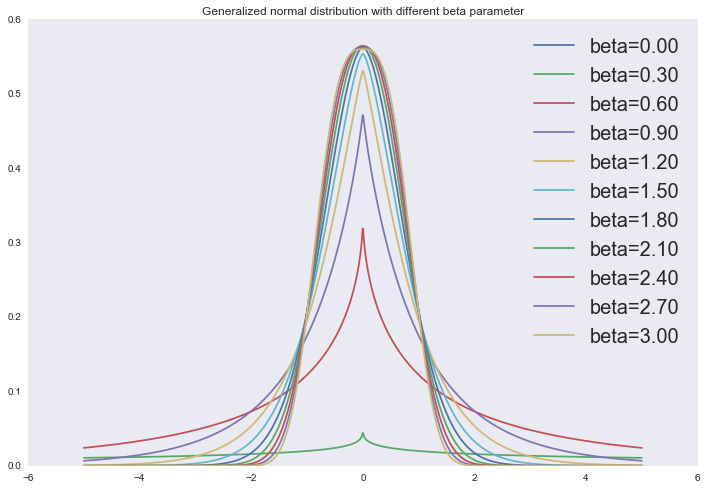
p(w→∣y→,X,σ2)∝Laplace(w→∣0,β)∏i=1nN(yi∣w→Tx→i,σ2)

Отрисовка графика

  
  
Глобальная динамика не изменилась: увеличиваем параметр регуляризации — решение приближается к центру априорного распределения. Также мы можем наблюдать, что такая регуляризация способствует нахождению разреженных решений: вы можете видеть два участка, на которых сначала один параметр равен нулю, затем второй параметр (в конце оба равны нулю).  
  
И на самом деле два описанных регуляризатора — это частные случаи наложения [обобщенного нормального распределения](https://en.wikipedia.org/wiki/Generalized_normal_distribution) в качестве априорного распределения на параметры линейной регрессии:

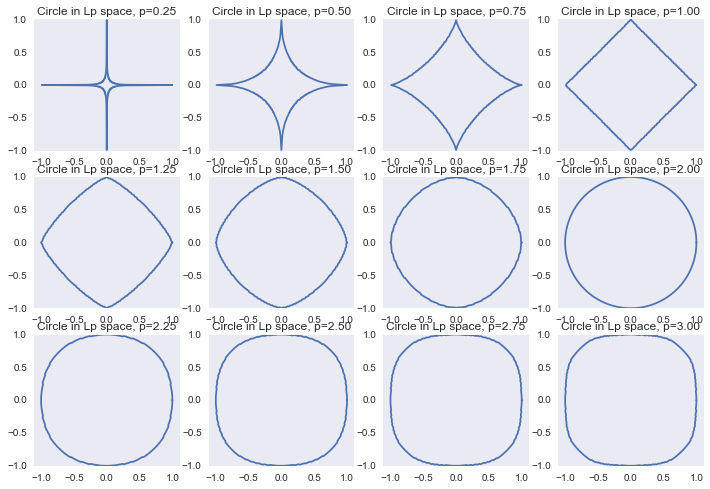
p(x∣α,β,μ)=β2αΓ(1β)e−(|x−μ|α)β

Отрисовка графика

  
  
Или же мы можем смотреть на эти регуляризаторы с точки зрения ограничения Lp нормы, как в предыдущей части:

‖x‖p=(∑i=1∞|x|p)1p

Отрисовка графика



Заключение

[Здесь](https://github.com/mephistopheies/dds/blob/master/lr_040117/ipy/lecture.ipynb) вы найдете jupyter notebook со всем вышеописанным и несколькими бонусами. Отдельное спасибо тем, кто осилил этот текст до конца.   
  
Желающим копнуть эту тему глубже, рекомендую:

* [лекции Сергея Николенко](http://logic.pdmi.ras.ru/~sergey/teaching/mlkfu2014.html), откуда позаимствована идея этого jupyter notebook'a;
* [лекции Бориса Демешева](https://www.youtube.com/playlist?list=PL-U1Z5tJ1i-YA-GdNXmQ8TEB0jjeDrDb_) по эконометрике (со 146ого видео), и его же [курс на курсере](https://ru.coursera.org/learn/ekonometrika).

Понимание линейной регрессии является ключом к пониманию более сложных моделей, вплоть до глубоких нейронных сетей. Если мы возьмем сигмойд от линейной функции — получим логистическую регрессию. Состекаем несколько логрегрессоров в один слой — получим softmax regression/max entropy regression. А если состекать несколько слоев — будет неронная сеть. Такие дела.