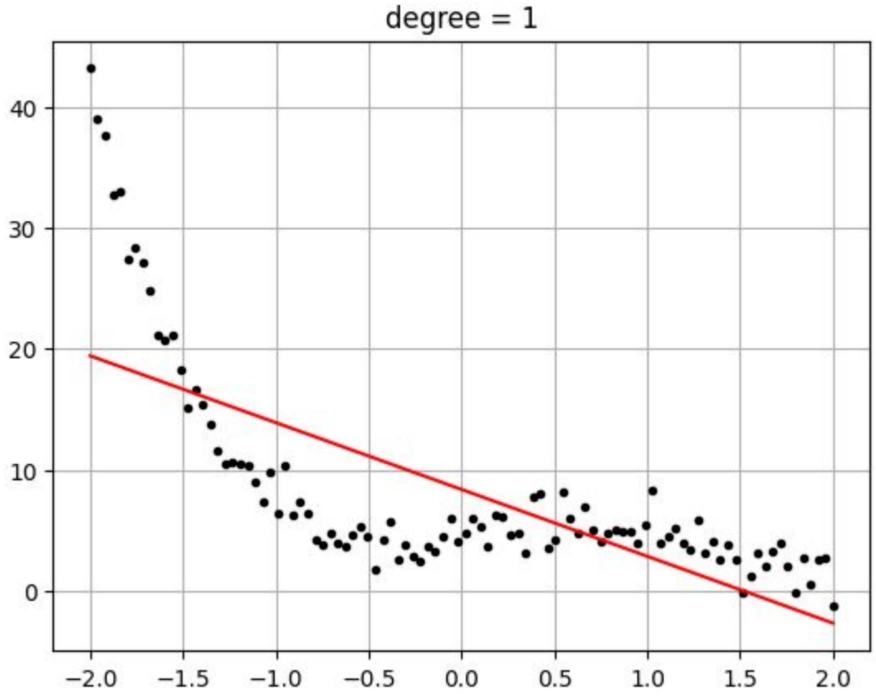
Лекция 7

Регрессия

Полиномиальная регрессия

- В реальных задачах данные далеко не всегда имеют линейную зависимость.
- При наличии нелинейной зависимости, можно попытаться подобрать полиномиальную зависимость
- Коэффициенты полинома можно найти при помощи обычной линейной регрессии



Конструирование полиномиальных признаков

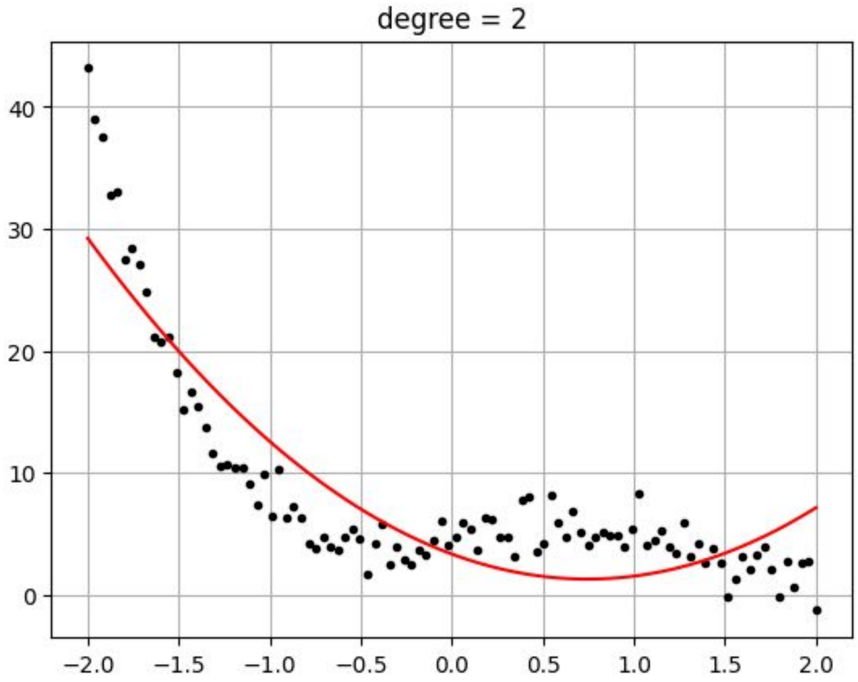
- Линейная регрессия имеет вид $y = f(b,x') = b_0 + b_1x'_1 + b_2x'_2 + \ldots$
- Главное, чтобы линейная регрессия была линейной комбинацией параметров
- Поэтому, уравнение регрессии записать как $y = f(b,x) = b_0 + b_1c_1(x) + b_2c_2(x) + \ldots$
- То есть, коэффициент возле параметра можно представить как функцию, зависящую от предиктора
- Если предположить, что $c_1(x) = x$, $c_2(x) = x^2$ и так далее ($c_n(x) = x^n$, то уравнение линейной регрессии от одного предиктора можно представить как $y = f(b,x) = b_0 + b_1x + b_2x^2 + \ldots + b_nx^n$.
- Для нескольких предикторов можно представить как $y = f(b, [x_1,x_2]) = b_0 + b_1x_1^2 + b_2x_1x_2 + b_3x_2^2$
- Таким образом, полиномиальная регрессия может быть решена как линейная

Полиномиальные признаки в Sklearn

- В Sklearn для генерации полиномиальных признаков есть sklearn.preprocessing.PolynomialFeatures. Признаки создаются по увеличению степени
- Пар. degree - задает степень полинома. Можно указать нижнюю и верхнюю границу признаков.
- Пар. include_bias - если True, создаст признак равный 1
- Пар. interaction_only - если True, создаст только комбинации признаков. Например, для x_1, x_2, x_3 будут получены только 1, x_1, x_2, x_3, x_1x_2, x_1x_3, x_2x_3, $x_1x_2x_3$

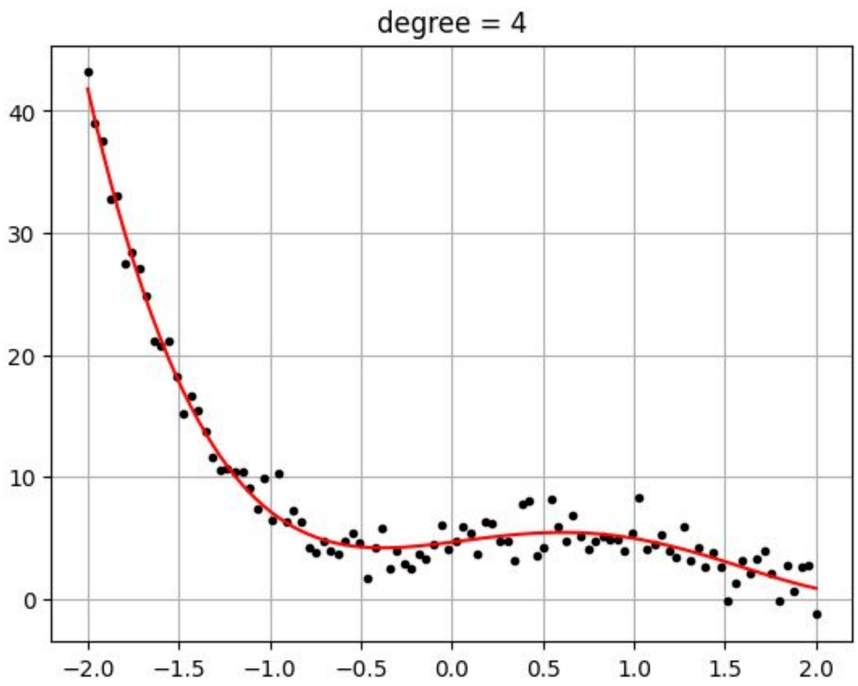
Полиномиальная регрессия пример (1)

```
x2 = PolynomialFeatures(2).fit_transform(x)
lin2 = LinearRegression(fit_intercept = False)
lin2.fit(x2, y)

c2 = lin2.coef_[0]
y2_pred = c2[0] + c2[1] * x + c2[2] * x * x

plt.plot(x,y,'k.')
plt.plot(x,y2_pred,'r-')
plt.grid()
plt.title('degree = 2')
plt.show()
```



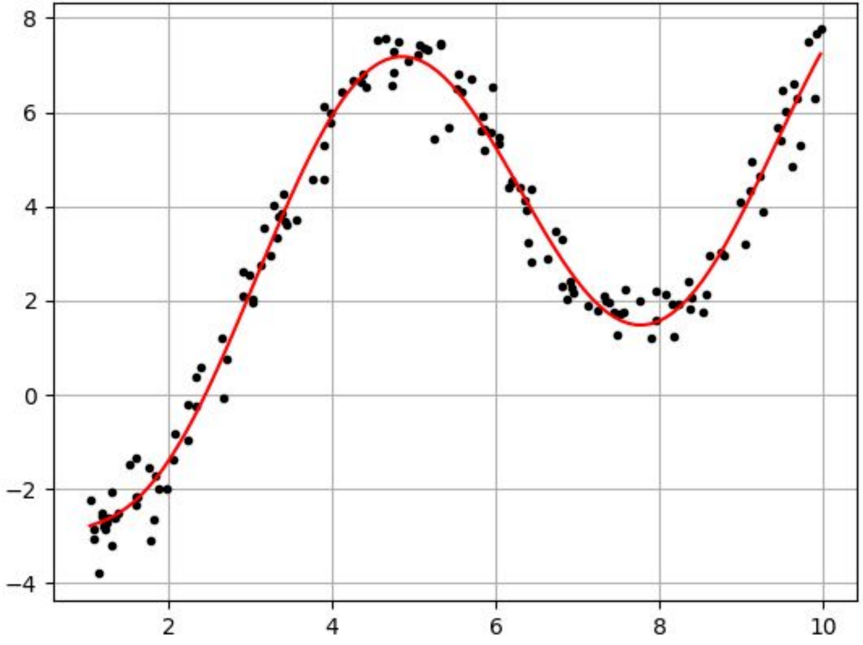
Полиномиальная регрессия пример (2)

```
x4 = PolynomialFeatures(4,
                include_bias=False).fit_transform(x)
lin4 = LinearRegression(fit_intercept = True)
lin4.fit(x4, y)

c4 = lin4.coef_[0]
y4_pred =
lin4.intercept_+c4[0]*x+c4[1]*x*x+c4[2]*x*x*x+
                      c4[3]*x*x*x*x

plt.plot(x,y,'k.')
plt.plot(x,y4_pred,'r-')
plt.grid()
plt.title('degree = 4')
plt.show()
```



Нелинейная регрессия с помощью линейной

- По аналогии можно конструировать нелинейные признаки.
- Такое конструирование очень затруднительно, если заранее не знать вид зависимости.
- Лучше делать аппроксимацию полиномиальной функцией. По сути представление функции в виде ряда Тейлора.
- Линейной регрессией нельзя подобрать параметры уравнения по типу $y = b_1 b_2 \exp(b_3 x)$



y = 2.44*ln(x) - 3.5*sin(x)

Не информативные признаки

- Заранее не известно, какие признаки значимые, а какие нет.
- Линейная регрессия по умолчанию использует все признаки, и тем самым может находить зависимости в шуме.
- Можно наложить штраф на параметры - регуляризировать.
- l1 - регуляризация модифицирует SSE следующим образом:

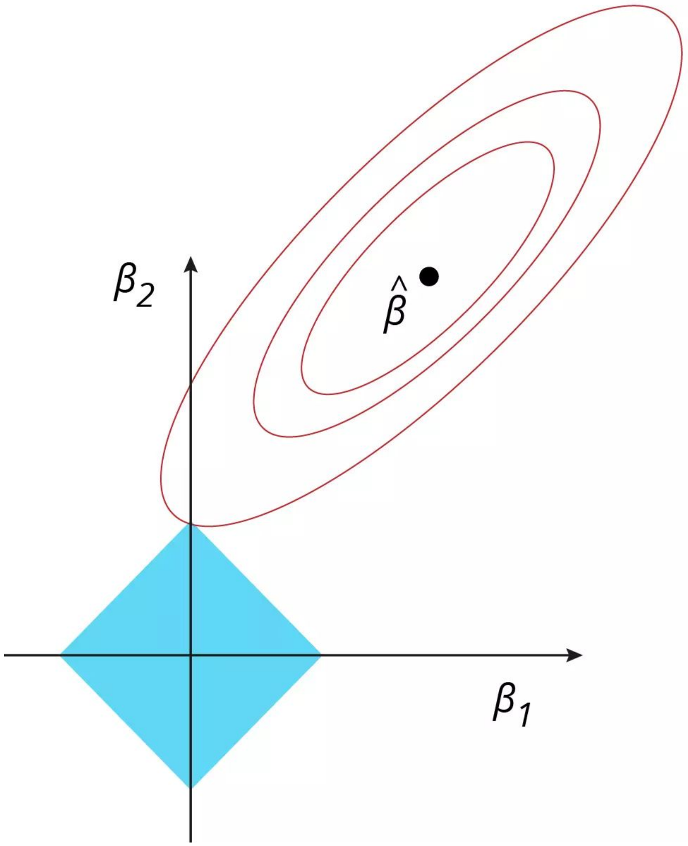
$$\sum_{i=1}^{n} (y - \hat{y})^2 + \alpha \sum_{j=1}^{m} |b_m|$$

- Такая регрессия называется Lasso (*Least Absolute Shrinkage and Selection Operator*)
- В Sklearn sklearn.linear_model.Lasso

Особенность Лассо регрессии

- В Лассо регрессии также минимизируется сумма модулей параметров.
- Таким образом, Лассо регрессия позволяет не учитывать не информативные признаки путем, путем обнуления параметров.
- Значение α задает силу штрафа к параметрам. Чем больше значение, тем сильнее штраф и больше параметров обнуляются

$$\sum_{i=1}^{n}(y-\hat{y})^2+\alpha\sum_{j=1}^{m}|b_m|$$



Мультиколлинеарность признаков

- Когда признаки имеют зависимости между собой, то происходит дублирование информации, вследствие чего регрессор может обучиться неправильно и иметь высокую дисперсию.
- Часто проявляется в виде того, что одни параметров гораздо больше других.
- l2 - регуляризация модифицирует SSE следующим образом:

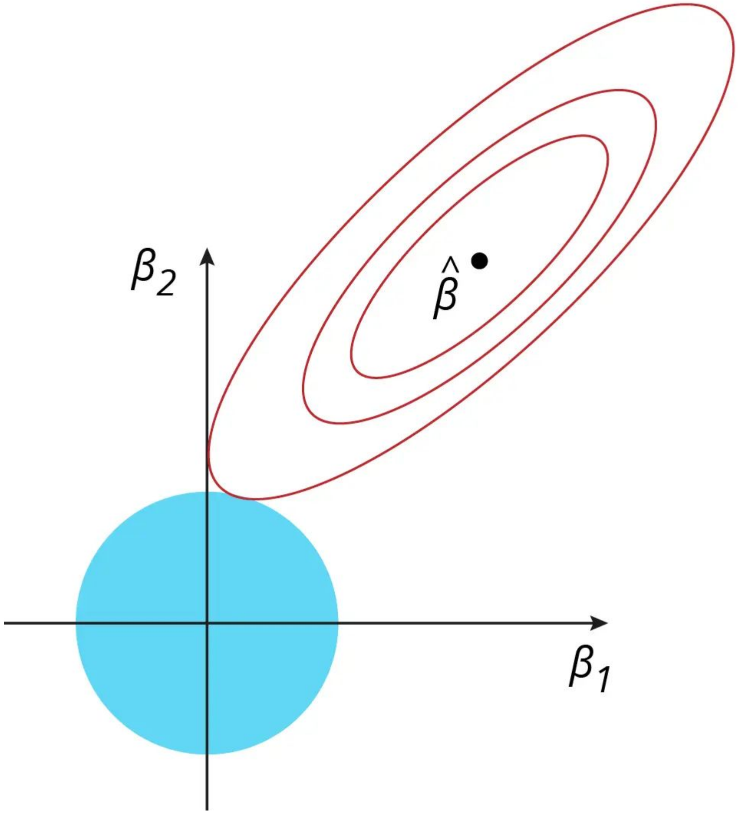
$$\sum_{i=1}^{n} (y - \hat{y})^2 + \alpha \sum_{j=1}^{m} (b_m)^2$$

- Такая регрессия называется Ridge (Гребневая)
- В Sklearn sklearn.linear_model.Ridge

Особенность Гребновой регрессии

- В Гребневой регрессии также минимизируется сумма квадратов параметров.

- Таким образом, Гребневая регрессия позволяет ограничить рост коэффициентов, пытаясь сделать их как можно меньше.



Метод эластичной сети (Elastic net)

- l1 и l2 регуляризации можно объединить в одной модели.
- В такой модели будут отбрасывать неинформативные признаки, а также будет происходить минимизация коэффициентов

$$\sum_{i=1}^{N}(y-\widehat{y})^2+\lambda_1\sum_{j=1}^{m}|b_m|+\lambda_2\sum_{j=1}^{m}(b_m)^2$$

$$\sum_{i=1}^{n}(y-\hat{y})^2\,(1-\alpha)\sum_{j=1}^{m}|b_m|+\alpha\sum_{j=1}^{m}(b_m)^2\,,\alpha=\frac{\lambda_2}{\lambda_1+\lambda_2}$$

- В Sklearn sklearn.linear_model.ElasticNet

Градиентные методы

Методы оптимизации

- Методы 0-го порядка анализируют только значение целевой функции. Для поиска максимума или минимума можно использовать координатный спуск или случайный поиск. Плохо работают в случае множества мин./макс. и большой размерности.
- **Методы 1-го порядка учитывают градиент (1-я производная) целевой функции. Градиент указывает направление возрастания целевой функции.**
- Методы 2-го порядка в расчетах используют матрицу Гессе (2-я производная) целевой функции. Матрица Гессе показывает выпуклость/вогнутость, что позволяет быстро находить глобальные мин./макс.. Данные методы эффективные, но используются редко из-за дороговизны вычислений.

Градиентный спуск

- Когда целевая дифференцируема, то можно найти точку минимума или максимума в которой для градиента соблюдается условие оптимальности:

$$\begin{cases} \frac{\partial f(x,w)}{\partial w_1} = 0 \\ \frac{\partial f(x,w)}{\partial w_2} = 0 \\ \frac{\partial f(x,w)}{\partial w_n} = 0 \end{cases}$$

- Не гарантируется, что оптимальная точка является глобальным мин./макс.
- Идея градиентного спуска заключается в том, что задается случайный набор параметров w, а затем итеративно идет минимизации функция путем изменением параметров w по направлению антиградиента:

$$w_{n+1} = w_n - \alpha \nabla f(x, w_n)$$

- Параметр α - скорость обучения. Определяет силу изменения параметров.

Линейная регрессия через градиентный спуск

- Хоть для решения задачи линейной регрессии есть аналитическое решение через систему уравнения, но для большого количества наблюдений >10 000 требуется большое количество ресурсов. Поэтому, в таких ситуациях рекомендуется применять градиентный спуск для решения задачи.
- В Sklearn для решения таким путем есть sklearn.linear_model.SGDRegressor. Обладающий следующими параметрами:
  - penalty - добавления l1/l2 регуляризации
  - eta0 - скорость обучения
  - learning_rate - корректировка скорости обучения по ходу алгоритма
  - max_iter - максимальное кол-во итераций
  - tol - критерий ранней остановки

Как искать градиент?

1. Аналитически - самостоятельно вычислить формулу градиента целевой функции, а затем ее запрограммировать. Самый быстрый способ вычисления, но удобен только для простых целевых функций.

2. Численно - расчет приближенного значения градиента на основе расчета функции для небольшого изменения входных значений. Позволяет находить градиент любой функции без информации о ее производных, но дает лишь приближенное значение.

3. Автоматическое дифференцирование - использует цепное правило, представляя сложную функцию как композицию более простых, для которых известно аналитическое решение. Зная дерево вычислений функции, можно вывести аналитическую форму для сложной функции.

Автоматическое дифференцирование лежит в основе таких библиотек как TensorFlow и PyTorch

Цепное правило

- Имеется сложная функция вида $y = f(g(h(x)))$
- Ее можно представить как $y = f(g(h(w_0))) = f(g(w_1)) = f(w_2) = w_3$
- То цепное правило для вычисления частных производных выглядит как:

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial w_2}\frac{\partial w_2}{\partial w_1}\frac{\partial w_1}{\partial x} = \frac{\partial f(w_2)}{\partial w_2}\frac{\partial g(w_1)}{\partial w_1}\frac{\partial h(w_0)}{\partial w_0}$$

- Зная дерево вычислений и выполняя замену переменных, можно найти частные производные.

Типы автоматического дифференцирования

- Прямое вычисление, рекурсивно вычисляет выражение

$$\frac{\partial w_i}{\partial x} = \frac{\partial w_i}{\partial w_{i-1}} \frac{\partial w_{i-1}}{\partial x}, w_i = y$$

- Обратное вычисление, рекурсивно вычисляет выражение

$$\frac{\partial y}{\partial w_i} = \frac{\partial y}{\partial w_{i+1}} \frac{\partial w_{i+1}}{\partial w_i}, w_i = x$$

- Затраты на вычисления частных производных примерно такое же как и у вычисления самой функции
- Если есть функция $f : \mathbb{R}^n \to \mathbb{R}^m$, то:
  - прямое вычисление эффективно если n << m
  - обратное вычисление эффективно если n >> m (исп. в машинном обуч.)

Прямое вычисление

- При прямом вычислении расчет начинается с независимых переменных
- Если переменная напрямую зависит от независимой переменной, то

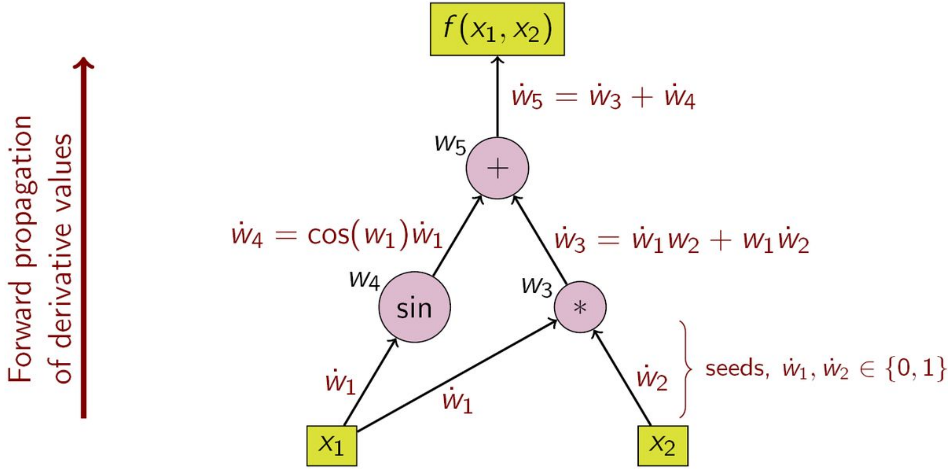
$$\dot{w}_i = \frac{\partial w_i}{\partial x}$$

- Если переменная напрямую не зависит от независимой переменной, то

$$\dot{w}_i = \sum_{j \in predictor} \frac{\partial w_i}{\partial w_j} \dot{w}_j$$

- Например, есть ф-ция

$$y = f(x_1, x_2) = x_1 x_2 + sin(x_1) = w_1 w_2 + sin(w_1) = w_3 + w_4 = w_5$$



$w_1 = x_1$	$w'_1 = 1$ (ядро)
$w_2 = x_2$	$w'_2 = 0$ (ядро)
$w_3 = w_1 w_2$	$w'_3 = w_2 w'_1 + w_1 w'_2$
$w_4 = \sin(w_1)$	$w'_4 = \cos(w_1) w'_1$
$w_5 = w_3 + w_4$	$w'_5 = w'_3 + w'_4$

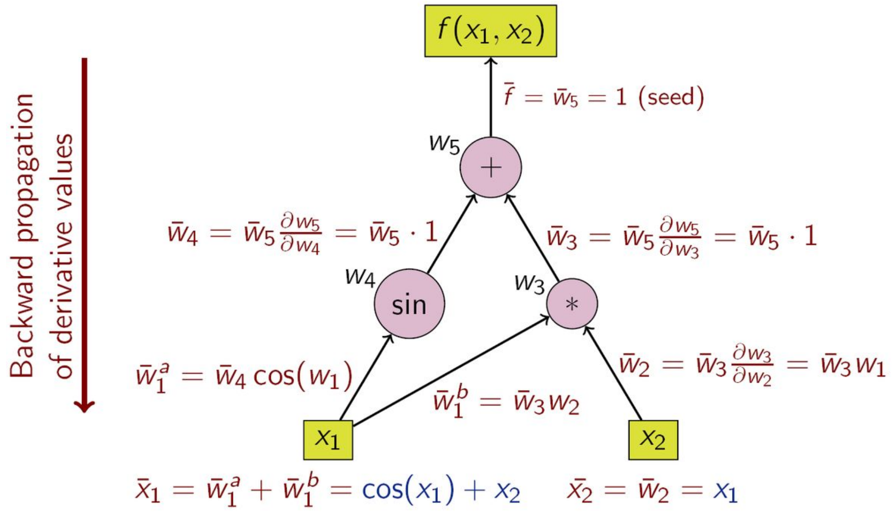
Обратное вычисление

- При обратном вычислении расчет начинается с зависимой переменной
- Если переменная напрямую влияет на зависимую переменную, то

$$\overline{w}_i = \frac{\partial y}{\partial w_i}$$

- Если переменная напрямую не влияет на зависимую переменную, то

$$\overline{w}_i = \sum_{j \in depended} \overline{w}_j \frac{\partial w_j}{\partial w_i}$$



$w_5 = f(x_1, x_2) = w_3 + w_4$	$w'_{5,5} = 1$ (ядро)
$w_4 = \sin(w_1)$	$w'_{5,4} = w'_5 * 1$
$w_3 = w_1 w_2$	$w'_{5,3} = w'_5 * 1$
$w_2 = x_2$	$w'_{5,2} = w'_3\ w_1 = x_1$
$w_1 = x_1$	$w'_{5,1} = w'_3 \cos(w_1) + w'_3 w_2 = \cos(x_1) + x_2$

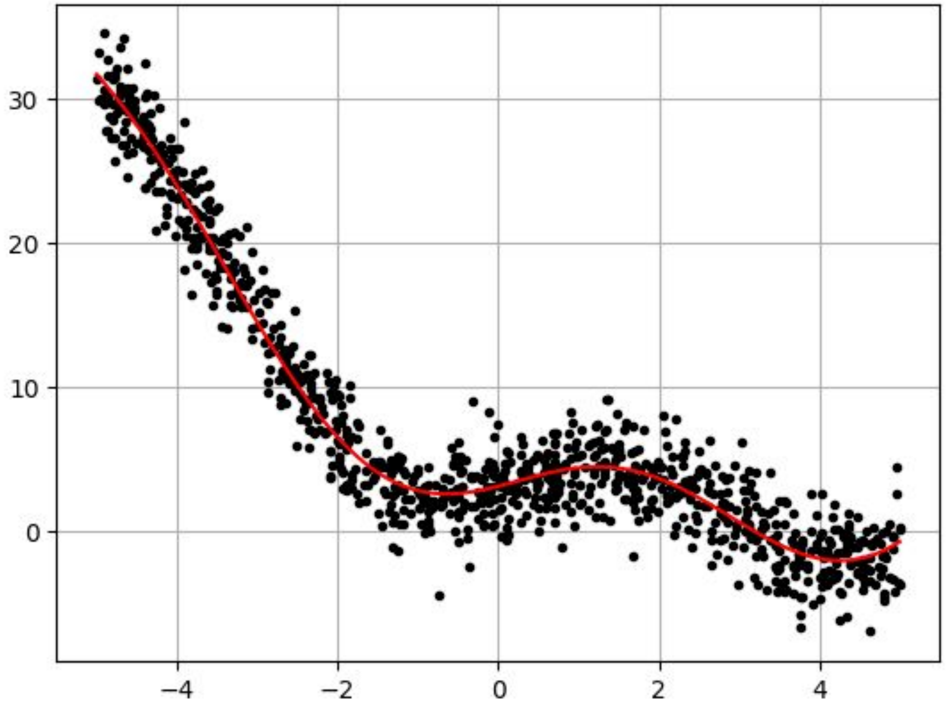
Расчет градиента в TensorFlow

- TensorFlow позволяет автоматически строить графы вычисления и дифференцирования
- tf.Variable - переменная, которую можно менять
- tf.constant - константная переменная
- tf.GradientTape - лента для хранения всех вычислений. Используется для вычисления градиента

```
x = tf.Variable(4.0)
b = tf.constant(2.0)
c = tf.constant(1.0)
with tf.GradientTape() as tape:
    #запись вычислений
    y = x * x - b * x + c
dy_dx = tape.gradient(y,x)
x.assign_add(-(dy_dx * 0.333))
```

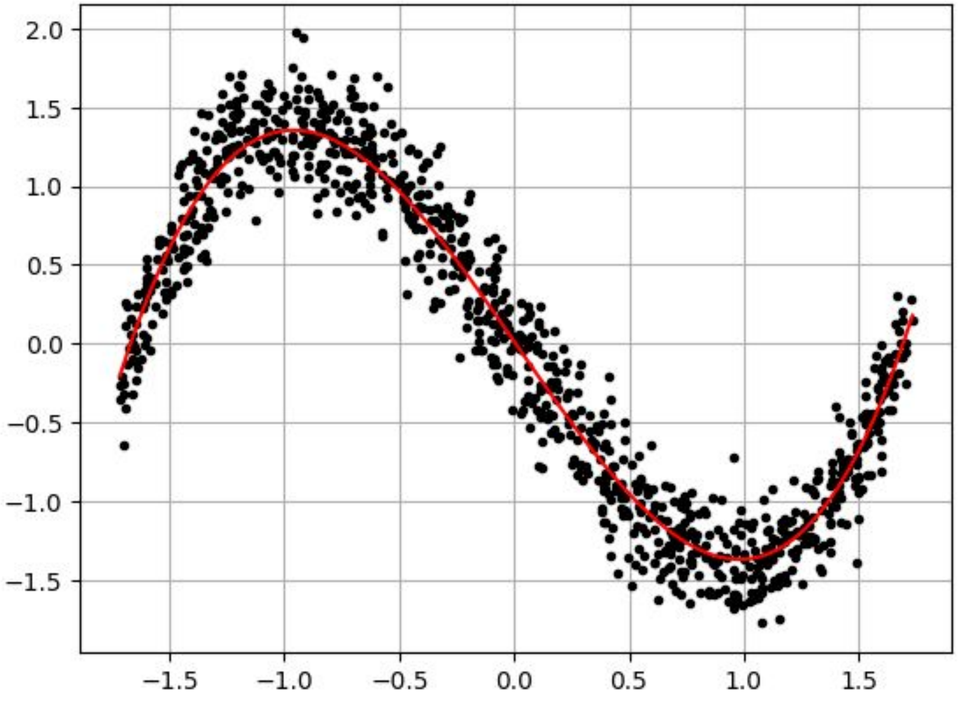
Нелинейная регрессия в TensorFlow

```
#данные константа, так как в ходе обучения не меняются
x_tf = tf.constant(X, dtype = tf.float32)
y_tf = tf.constant(Y, dtype = tf.float32)
#генерируем случайные параметры модели
w = [tf.Variable(np.random.randn()) for _ in range(4)]
#скорость обучения
alpha = tf.constant(0.001, dtype = tf.float32)
#количество итераций (эпох)
epoch_n = 3000
#цикл обучения
for epoch in range(epoch_n):
    with tf.GradientTape() as tape:
        y_pred = w[0] * (x_tf - w[1]) * (x_tf - w[2])
                    + w[3] * tf.math.sin(x_tf)
        loss = tf.reduce_mean(tf.square(y_tf - y_pred))
    grad = tape.gradient(loss, w)
    for i in range(4):
        w[i].assign_add(-(alpha * grad[i]))
    if(epoch+1) % 250 == 0:
        print(f"E: {epoch+1}, L: {loss.numpy()}")
```



Полиномиальная регрессия в TensorFlow

```
x2_tf = tf.constant(X2, dtype = tf.float32)
y2_tf = tf.constant(Y2, dtype = tf.float32)
names = ['A','B','C','D']
coefs = [tf.Variable(np.random.rand(), name = sym)
                    for sym in names]
epoch2_n = 1500
#используем встроенные оптимизатор
optimizer = tf.keras.optimizers.SGD(0.01)
for epoch in range(epoch2_n):
    with tf.GradientTape() as tape:
        y_pred2 = tf.math.polyval(coefs, x2_tf)
        loss = tf.reduce_mean(tf.square(y2_tf - y_pred2))
    grads = tape.gradient(loss, coefs)
    #применение градиентов
    optimizer.apply_gradients(zip(grads, coefs))
    if(epoch + 1) % 50 == 0:
        print(f"E: {epoch+1}, L: {loss.numpy()}")
```



Библиотек Autograd

- Autograd - простая библиотека для расчета градиента функции https://autograd.readthedocs.io/
- Позволяет устанавливать режим расчета

```
ad.set_mode('forward') #'reverse'
x = Variable(-3)
b1 = x * x
b2 = -2 * x
y = b1 + b2 + 1
print(y)
```