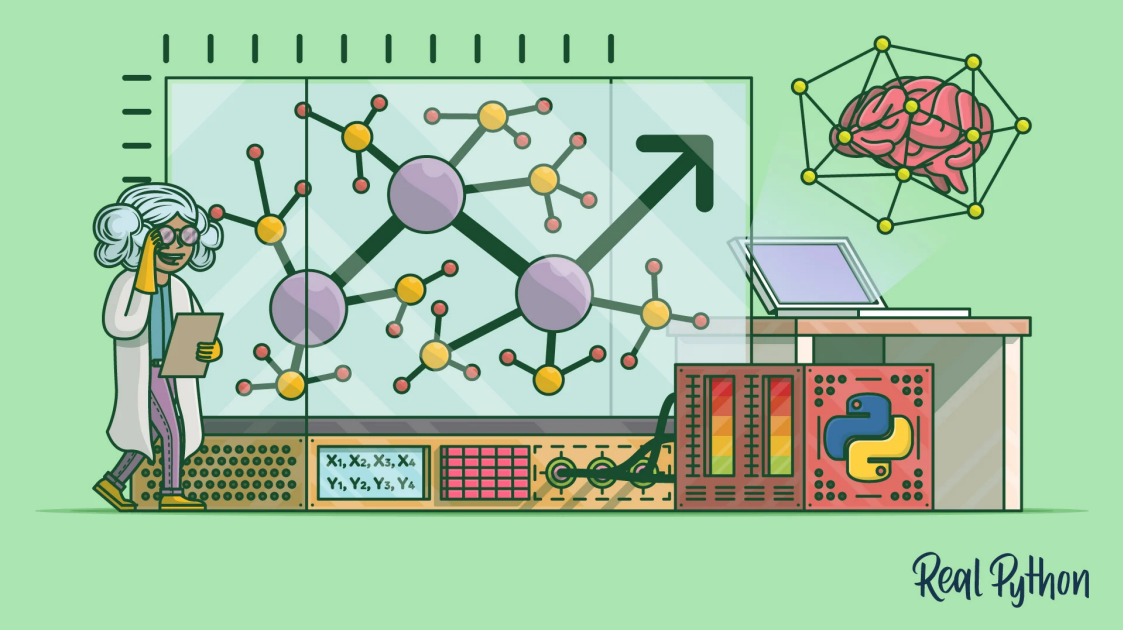
Linear-Regression

3 июня 2022 г.



1 Линейная регрессия

Начнем с обычных импортов:

[1]:
```
%matplotlib inline
%matplotlib notebook
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
import numpy as np
```

1.1 Простая линейная регрессия

Имеем некоторую неслучайную переменную x, значение которой меняющиеся от опыта к опыту при проведении эксперимента. Для каждого значения x_k $(k = \overline{1,m})$ наблюдается некоторый эффект η_k (измерения). Также известно, что измерения получаются с ошибкой и связь между значениями неслучайной переменной x_k, случайными измерениями η_k и случайной ошибкой ε_k имеет вид $\eta_k = ax_k + b + \varepsilon_k$ $(k = \overline{1,m};\ a, b \in \mathbb{R})$. Требуется найти значения $\widehat{a}$, $\widehat{b}$ наилучшим, в

некотором смысле, образом согласующие наблюдаемые значения y_k случайной величины η_k со значениями переменной x_k.

Рассмотрим сумму квадратов наблюдаемых ошибок e_k случайной величины ε_k:

$$S(a,b) = \sum_{k=1}^{m} e_k^2 = \sum_{k=1}^{m} (y_k - ax_k - b)^2 .$$

Значения $\widehat{a}$, $\widehat{b}$, удовлетворяющие равенству

$$S\left(\widehat{a},\widehat{b}\right) = \min_{(a,b)\in\mathbb{R}^2} S(a,b) = \min_{(a,b)\in\mathbb{R}^2} \sum_{k=1}^{m} (y_k - ax_k - b)^2 ,$$

называют оценкой наименьших квадратов (ОНК) параметров.

$\widehat{y} = \widehat{a}x + \widehat{b}$ – предсказанное значение y для данного x.

$$\operatorname{grad} S(a,b) = \begin{pmatrix} \dfrac{\partial S}{\partial a} \\ \dfrac{\partial S}{\partial b} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \Leftrightarrow \begin{cases} \sum\limits_{k=1}^{m} 2(y_k - ax_k - b)(-x_k) = 0, \\ \sum\limits_{k=1}^{m} 2(y_k - ax_k - b)(-1) = 0. \end{cases} \Leftrightarrow$$

Минимум функционала $S(a,b)$ достигается при

$$\widehat{a} = \frac{m\sum\limits_{k=1}^{m} x_k y_k - \left(\sum\limits_{k=1}^{m} x_k\right)\left(\sum\limits_{k=1}^{m} y_k\right)}{m\sum\limits_{k=1}^{m} x_k^2 - \left(\sum\limits_{k=1}^{m} x_k\right)^2}, \quad \widehat{b} = \frac{1}{m}\left(\sum_{k=1}^{m} y_k - \widehat{a}\sum_{k=1}^{m} x_k\right).$$

Эквивалентные формулы

$$\widehat{a} = \frac{\sum\limits_{k=1}^{m} (x_k - \overline{X})(y_k - \overline{Y})}{\sum\limits_{k=1}^{m} (x_k - \overline{X})^2}, \ \widehat{b} = \overline{Y} - \widehat{a}\overline{X}, \text{ где } \overline{X} = \frac{1}{m}\sum_{k=1}^{n} x_k, \ \overline{Y} = \frac{1}{m}\sum_{k=1}^{n} y_k.$$

Тогда предсказанное значение $\widehat{y} = \widehat{a}(x - \overline{X}) + \overline{Y}$.

1.1.1 Пример

Рассмотрим следующие данные, распределенные около прямой $y = 2x - 5$:

[2]:
```
m=50

rng = np.random.RandomState(81)
x = 10 * rng.rand(m)
y = 2 * x - 5 + rng.randn(m)
plt.figure()
plt.scatter(x, y)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[2]: <matplotlib.collections.PathCollection at 0x1e6d0b004f0>

[3]:
```
C=np.cov(x,y)

a=C[0,1]/C[0,0]
b=np.mean(y)-a*np.mean(x)

xfit = np.linspace(0, 10, 10)
yfit = a*xfit + b

print(f"Model coef:     a={a}")
print(f"Model intercept: b={b}")

plt.figure()
plt.scatter(x, y, color='green')
plt.plot(xfit, yfit)
```

```
Model coef:     a=1.9681862451970724
Model intercept: b=-4.7583874782991815
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[3]: [<matplotlib.lines.Line2D at 0x1e6d0b62340>]

В `sklearn` есть несколько классов, реализующих линейную регрессию: * `LinearRegression` — "классическая" линейная регрессия с оптимизацией MSE. * `Ridge` — линейная регрессия с оптимизацией MSE и ℓ_2-регуляризацией * `Lasso` — линейная регрессия с оптимизацией MSE и ℓ_1-регуляризацией

У моделей из `sklearn` есть методы `fit` и `predict`. Первый принимает на вход обучающую выборку и вектор целевых переменных и обучает модель, второй, будучи вызванным после обучения модели, возвращает предсказание на выборке.

[4]:
```
from sklearn.linear_model import LinearRegression
model = LinearRegression(fit_intercept=True)

model.fit(x[:, np.newaxis], y)

xfit = np.linspace(0, 10, 10)
yfit = model.predict(xfit[:, np.newaxis])

plt.figure()
plt.scatter(x, y, color='green')
plt.plot(xfit, yfit)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[4]:
```
[<matplotlib.lines.Line2D at 0x1e6d2272d60>]
```

Подбираемые параметры модели (в библиотеке `Scikit-Learn` всегда содержат в конце знак подчеркивания) включают угловой коэффициент a и точку пересечения с осью ординат b. В данном случае соответствующие параметры – `coef_` и `intercept_`:

[5]:
```
print(f"Model coef:      a={model.coef_[0]}")
print(f"Model intercept: b={model.intercept_}")
```

```
Model coef:      a=1.9681862451970729
Model intercept: b=-4.758387478299183
```

Полученные результаты очень близки к исходной зависимости, как и должно быть.

1.2 Многомерная линейная регрессия

Дано: $\eta_k = x_{1k}w_1 + \cdots + x_{nk}w_n + \varepsilon_k = \overrightarrow{x}_k^T \overrightarrow{w} + \varepsilon_k \ (k = \overline{1,m}) \Leftrightarrow \overrightarrow{\eta} = X^T \overrightarrow{w} + \overrightarrow{\varepsilon}$

1. $\overrightarrow{\eta} = (\eta_1, \ldots, \eta_m)^T$ – случайный вектор, состоящий из $m \in \mathbb{N}$;
2. $X = \left(\overrightarrow{x}_1, \ldots, \overrightarrow{x}_m\right)$ – матрица из m столбцов $\overrightarrow{x}_k \ (k = \overline{1,m})$, где $\overrightarrow{x}_k$ – последовательность неслучайных векторных переменных (факторов);
3. $\overrightarrow{\varepsilon} = (\varepsilon_1, \ldots, \varepsilon_m)^T$ – вектор, состоящий из m случайных ошибок $\varepsilon_k \ (k = \overline{1,m})$; $M\varepsilon_k = 0$; $D\varepsilon_k = \sigma^2 > 0$; $M\left(\varepsilon_i \varepsilon_j\right) = 0 \ (i = \overline{1,m}, \ j = \overline{i+1,m})$ (случайные ошибки некоррелированы);
4. $\overrightarrow{w} = (w_1, \ldots, w_n)^T$ – вектор неизвестных неслучайных параметров (весов).

Если выполняются условия $1-4$, то говорят, что имеет место модель линейной регрессии; $w_1, \ldots, w_n$ – коэффициенты регрессии; σ^2 – остаточная дисперсия.

1.2.1 Пример

В простой регрессии: - $\overrightarrow{\eta} = (\eta_1, \ldots, \eta_m)^T$ – наблюдения; - $\overrightarrow{x}_k = (x_k \quad 1)^T$ – факторы; - $\overrightarrow{w} = (a \quad b)^T$ – неизвестные параметры.

$$\eta_k = \overrightarrow{x}_k^T \overrightarrow{w} + \varepsilon_k = \begin{pmatrix} x_k & 1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} + \varepsilon_k = ax_k + b + \varepsilon_k$$

1.2.2 Свойства

1. $M\eta_k = M\left\{\overrightarrow{x}_k^T \overrightarrow{w}\right\} + M\varepsilon_k = \overrightarrow{x}_k^T \overrightarrow{w} \ \left(k = \overline{1,m}\right)$
2. $D\eta_k = M\left[(\eta_k - M\eta_k)^2\right] = M\left[\left(\overrightarrow{x}_k^T \overrightarrow{w} + \varepsilon_k - \overrightarrow{x}_k^T \overrightarrow{w}\right)^2\right] = M\varepsilon_k^2 = D\varepsilon_k = \sigma^2 \ \left(k = \overline{1,m}\right) \Rightarrow$ $D\overrightarrow{\eta} = D\overrightarrow{\varepsilon} = M\left[\overrightarrow{\varepsilon}\,\overrightarrow{\varepsilon}^T\right] = \sigma^2 \cdot E_m$

1.2.3 Замечание

В более общем случае возможны корреляции между ошибками, а следовательно и наблюдениями, т. о. $D\overrightarrow{\varepsilon} = M\left[\overrightarrow{\varepsilon}\,\overrightarrow{\varepsilon}^T\right] = \sigma^2 \cdot G$.

Если ввести замену переменных $\overrightarrow{\zeta} = G^{-1/2}\overrightarrow{\eta} \Rightarrow$ - Математическое ожидание $\overrightarrow{\zeta}$: $M\overrightarrow{\zeta} = M\left[G^{-1/2}\overrightarrow{\eta}\right] = G^{-1/2}M\overrightarrow{\eta} = G^{-1/2}X^T\overrightarrow{w}$, $XG^{-1/2}$ – новая матрица факторов; - Матрица ковариаций $\overrightarrow{\zeta}$:

$$\begin{aligned} D\overrightarrow{\zeta} =& M\left[\left(\overrightarrow{\zeta} - M\overrightarrow{\zeta}\right)\left(\overrightarrow{\zeta} - M\overrightarrow{\zeta}\right)^T\right] = \\ =& M\left[G^{-1/2}\left(\overrightarrow{\eta} - M\overrightarrow{\eta}\right)\left(\overrightarrow{\eta} - M\overrightarrow{\eta}\right)^T G^{-1/2}\right] = \\ =& G^{-1/2}\left(D\overrightarrow{\eta}\right)G^{-1/2} = G^{-1/2}\left(\sigma^2 \cdot G\right)G^{-1/2} = \sigma^2 \cdot E_m \end{aligned}$$

$\Rightarrow$ Заменой переменных $\overrightarrow{\zeta} = G^{-1/2}\overrightarrow{\eta}$ свели задачу к задаче с некоррелированными ошибкам, а значит модель $1-4$ достаточно общая и заслуживает подробного рассмотрения.

1.2.4 Метод наименьших квадратов (МНК)

Разработан К. Гауссом (1809 г.) и получил теоретико-вероятностное обоснование А. Марковым (1900 г.). Метод применяется для нахождения весов $\overrightarrow{w} = (w_1, \ldots, w_n)^T$ в задаче линейной регрессии.

Рассмотрим квадратичную форму

$$S(\overrightarrow{w}) = \overrightarrow{e}^T\overrightarrow{e} = (\overrightarrow{y} - X^T\overrightarrow{w})^T(\overrightarrow{y} - X^T\overrightarrow{w}),$$

которую называют функционалом МНК и $\widehat{\overrightarrow{w}} = \arg\min_{\overrightarrow{w}} S(\overrightarrow{w})$.

Необходимые условия экстремума:

$$\operatorname{grad} S(\overrightarrow{w}) = 0, \quad S(\overrightarrow{w}) = \overrightarrow{y}^T\overrightarrow{y} - \overrightarrow{y}^T X^T\overrightarrow{w} - \overrightarrow{w}^T X\overrightarrow{y} + \overrightarrow{w}^T XX^T\overrightarrow{w}.$$

Величина $\left(\overrightarrow{y}^T X^T\overrightarrow{w}\right)^T = \overrightarrow{w}^T X\overrightarrow{y}$ – скалярная $\Rightarrow$

$$\overrightarrow{y}^T X^T\overrightarrow{w} = \left(\overrightarrow{y}^T X^T\overrightarrow{w}\right)^T = \overrightarrow{w}^T X\overrightarrow{y}$$

Тогда $S(\overrightarrow{w}) = \overrightarrow{y}^T\overrightarrow{y} - 2\overrightarrow{y}^T X^T\overrightarrow{w} + \overrightarrow{w}^T XX^T\overrightarrow{w} \Rightarrow \operatorname{grad} S(\overrightarrow{w}) = -2X\overrightarrow{y} + 2XX^T\overrightarrow{w} = 0 \Leftrightarrow$ уравнению $XX^T\overrightarrow{w} = X\overrightarrow{y}$, называемого системой нормальных уравнений МНК.

Теорема. let $\widehat{\overrightarrow{w}}$ – произвольное решение системы нормальных уравнений $\Rightarrow \min_{\overrightarrow{w}} S(\overrightarrow{w}) = S(\widehat{\overrightarrow{w}})$ и этот минимум одинаков для всех решений системы нормальных уравнений.

if $det(XX^T) \neq 0 \Rightarrow$ оценка МНК единственная и определяется равенством $\widehat{\overrightarrow{w}} = (XX^T)^{-1}X\overrightarrow{y}$.

$\triangleleft$

$$\begin{aligned} S(\overrightarrow{w}) =& (\overrightarrow{y} - X^T\overrightarrow{w})^T(\overrightarrow{y} - X^T\overrightarrow{w}) = \\ =& \left((\overrightarrow{y} - X^T\widehat{\overrightarrow{w}}) + X^T(\widehat{\overrightarrow{w}} - \overrightarrow{w})\right)^T\left((\overrightarrow{y} - X^T\widehat{\overrightarrow{w}}) + X^T(\widehat{\overrightarrow{w}} - \overrightarrow{w})\right) = \\ =& (\overrightarrow{y} - X^T\widehat{\overrightarrow{w}})^T(\overrightarrow{y} - X^T\widehat{\overrightarrow{w}}) + 2(\widehat{\overrightarrow{w}} - \overrightarrow{w})^T X(\overrightarrow{y} - X^T\widehat{\overrightarrow{w}}) + \\ &+ (\widehat{\overrightarrow{w}} - \overrightarrow{w})^T XX^T(\widehat{\overrightarrow{w}} - \overrightarrow{w}) = \\ =& S(\widehat{\overrightarrow{w}}) + 2(\widehat{\overrightarrow{w}} - \overrightarrow{w})^T(X\overrightarrow{y} - XX^T\widehat{\overrightarrow{w}}) + (\widehat{\overrightarrow{w}} - \overrightarrow{w})^T XX^T(\widehat{\overrightarrow{w}} - \overrightarrow{w}) = \\ =& S(\widehat{\overrightarrow{w}}) + (\widehat{\overrightarrow{w}} - \overrightarrow{w})^T XX^T(\widehat{\overrightarrow{w}} - \overrightarrow{w}). \end{aligned}$$

Поскольку XX^T неотрицательно определена $\Rightarrow S(\overrightarrow{w}) \geq S(\widehat{\overrightarrow{w}}) \Rightarrow$ минимум $S(\overrightarrow{w})$ достигается в $\widehat{\overrightarrow{w}}$ и одинаков для $\forall$ решения нормальных уравнений $\Rightarrow$ любое решение системы нормальных уравнений является оценкой МНК. Для невырожденной XX^T система нормальных уравнений имеет единственное решение $\widehat{\overrightarrow{w}} = (XX^T)^{-1}X\overrightarrow{y}$. ▷

1.2.5 Пример

$$\eta_k = w_0 + w_1 x_{1k} + w_2 x_{2k} + \varepsilon_k$$

с несколькими величинами x_k. Геометрически это подобно подбору плоскости для точек в трех измерениях.

[6]:
```
np.random.seed(32)
X = np.random.uniform(0, 10, size=(50,2))

y = 0.5 + np.dot(X, [-2.0, 1.5]) + np.random.normal(scale=1.0, size=X.shape[0])
```

Возможности `LinearRegression` позволяют работать с многомерными линейными моделями.

[7]:
```
from sklearn.linear_model import LinearRegression
model = LinearRegression(fit_intercept=True)
model.fit(X, y)
print(model.intercept_)
print(model.coef_)
```

```
0.5790960288296627
[-1.91567403  1.41453722]
```

[8]:
```
from mpl_toolkits.mplot3d import Axes3D

t = np.linspace(0, 10, 50)
x_fit_1, x_fit_2 = np.meshgrid(t, t)

x_fit = np.column_stack((x_fit_1.ravel(),x_fit_2.ravel()))

y_fit = model.predict(x_fit)

fig = plt.figure(figsize=(4,4))
ax = Axes3D(fig, auto_add_to_figure=False)
#ax.plot_trisurf(x_fit[:,0], x_fit[:,1], y_fit, alpha=0.3, cmap='inferno')
ax.plot_surface(x_fit_1, x_fit_2, y_fit.reshape(x_fit_1.shape), linewidth=0,␣
 ↪rstride=10, cstride=10, alpha=0.3, cmap='rainbow')
ax.scatter(X[:,0], X[:,1], y, color='black', marker='*')
fig.add_axes(ax)

ax.elev = 15
ax.azim = 45
```

```
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Линейная регрессия может использоваться для аппроксимации наших данных прямыми, плоскостями и гиперплоскостями. Складывается впечатление, что этот подход ограничивается лишь строго линейными отношениями между переменными, но, если проявить изобретательность, возможности оказываются гораздо шире.

1.3 Регрессия в спрямляющем пространстве

Один из трюков, позволяющих приспособить линейную регрессию к нелинейным отношениям между переменными, – функциональное преобразование данных. Идея состоит в том, чтобы взять многомерную линейную модель:

$$y = w_1 z_1 + \ldots + w_N z_N$$

и подставить в нее $z_j = \varphi_j(\overrightarrow{x})$ $(j = \overline{1,N})$, где $\varphi_j : \mathbb{R}^n \to \mathbb{R}$ – некоторая функция, выполняющая преобразование факторов $\overrightarrow{x} \Rightarrow$

$$y = w_1 \varphi_1(\overrightarrow{x}) + \ldots + w_N \varphi_N(\overrightarrow{x}).$$

В итоге получается модель в N-мерном пространстве, которое называют спрямляющим.

Соответствующую задачу регрессии теперь можно переформулировать следующим образом: $\eta_k = \varphi_1(\overrightarrow{x}_k)w_1 + \cdots + \varphi_N(\overrightarrow{x}_k)w_N + \varepsilon_k$ $(k = \overline{1,m}) \Leftrightarrow \overrightarrow{\eta} = \Phi(X)\overrightarrow{w} + \overrightarrow{\varepsilon}$

В частности, для одномерных факторов x, если $\varphi_k(x) = x^{k-1}$, получаем полиномиальную регрессию:

$$y = a_0 + a_1 x + \ldots + a_{N-1} x^{N-1}$$

Обратим внимание, что модель по-прежнему остается линейной — линейность относится к тому, что коэффициенты w_k никогда не умножаются и не делятся друг на друга. Фактически мы взяли наши одномерные значения x и выполнили проекцию их на более многомерное пространство, так что с помощью линейной аппроксимации мы можем теперь отражать более сложные зависимости между x и y.

1.3.1 Полиномиальные базисные функции

Данное полиномиальное преобразование настолько удобно, что было встроено в библиотеку `Scikit-Learn` в виде преобразователя `PolynomialFeatures`:

[9]:
```
from sklearn.preprocessing import PolynomialFeatures
x = np.array([2, 3, 4])
poly = PolynomialFeatures(2, include_bias=True)
display(x[:, np.newaxis])
display(poly.fit_transform(x[:, np.newaxis]))
```

```
array([[2],
       [3],
       [4]])
```

```
array([[ 1.,  2.,  4.],
       [ 1.,  3.,  9.],
       [ 1.,  4., 16.]])
```

Преобразователь превратил наш одномерный массив в трехмерный путем возведения каждого из значений в степень. Далее можно будет пользоваться этим преобразователем для регрессионной модели.

После такого преобразования можно воспользоваться линейной моделью для подбора намного более сложных зависимостей между величинами x и y. Например, рассмотрим зашумленную синусоиду

[10]:
```
from sklearn.linear_model import LinearRegression

np.random.seed(36)
x = np.linspace(0, 1, 100)
y = np.cos(1.5 * np.pi * x)

x_train = np.random.uniform(0, 1, size=30)
y_train = np.cos(1.5 * np.pi * x_train) + np.random.normal(scale=0.1,␣
 ↪size=x_train.shape)

from sklearn.preprocessing import PolynomialFeatures
fig, axs = plt.subplots(figsize=(16, 4), ncols=3)
for i, degree in enumerate([1, 4, 30]):
    X_train = PolynomialFeatures(degree, include_bias=True).
 ↪fit_transform(x_train[:, np.newaxis])
    X = PolynomialFeatures(degree, include_bias=True).fit_transform(x[:, np.
 ↪newaxis])
    w=np.dot(np.linalg.inv(np.matmul(X_train.T,X_train)),np.dot(X_train.
 ↪T,y_train))
    y_pred=np.dot(X,w)

    axs[i].plot(x, y, label="Real function")
    axs[i].scatter(x_train, y_train, label="Data")
    axs[i].plot(x, y_pred, label="Prediction")
    if i == 0: axs[i].legend()
    axs[i].set_title("Degree = %d" % degree)
    axs[i].set_xlabel("$x$")
    axs[i].set_ylabel("$f(x)$")
    axs[i].set_ylim(-2, 2)

    print(f"Degree = {degree}")
    print(f"Coeffs = {w}")
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
Degree = 1
Coeffs = [ 0.77571792 -1.96635465]
Degree = 4
Coeffs = [  0.96001923   0.0697848  -14.40575795  18.10888186  -4.68598787]
Degree = 30
Coeffs = [ 8.63945778e-01  1.68288699e+01 -3.54481232e+02  2.68226575e+03
 -1.02001221e+04  2.26756875e+04 -4.03703594e+04  5.36838125e+04
  4.94457656e+04 -3.26019375e+05  3.26033844e+05  1.95884750e+05
 -2.39219250e+05 -5.84798500e+05  7.02292875e+05  1.27272875e+05
 -1.28943375e+05  3.26927375e+05 -1.87051919e+06  1.20632300e+06
  7.47456500e+05  2.55077250e+05 -1.44563600e+06  2.41653600e+06
 -5.15512250e+06  6.22653350e+06 -4.04926425e+06  1.44199575e+06
 -1.55871844e+06  2.39541267e+06 -1.08729412e+06]
```

Оформим функцию для дальнейшего использования с моделями `sklearn`

[11]:
```
def fit_and_draw(model):
    np.random.seed(36)
    x = np.linspace(0, 1, 100)
    y = np.cos(1.5 * np.pi * x)

    x_train = np.random.uniform(0, 1, size=30)
    y_train = np.cos(1.5 * np.pi * x_train) + np.random.normal(scale=0.1,␣
 ↪size=x_train.shape)

    from sklearn.preprocessing import PolynomialFeatures
    fig, axs = plt.subplots(figsize=(16, 4), ncols=3)
    for i, degree in enumerate([1, 4, 30]):
        X_train = PolynomialFeatures(degree, include_bias=False).
 ↪fit_transform(x_train[:, np.newaxis])
        X = PolynomialFeatures(degree, include_bias=False).fit_transform(x[:, np.
 ↪newaxis])

        regr = model.fit(X_train, y_train)
        y_pred = regr.predict(X)

        axs[i].plot(x, y, label="Real function")
        axs[i].scatter(x_train, y_train, label="Data")
        axs[i].plot(x, y_pred, label="Prediction")
        if i == 0:
            axs[i].legend()
        axs[i].set_title("Degree = %d" % degree)
        axs[i].set_xlabel("$x$")
        axs[i].set_ylabel("$f(x)$")
        axs[i].set_ylim(-2, 2)
```

```
        print(f"Degree = {degree}")
        print(f"Coeffs = {model.coef_}")
        print(f"Intercept = {model.intercept_}")
```

[12]:
```
from sklearn.linear_model import LinearRegression

poly_model = LinearRegression()
fit_and_draw(poly_model)
```

```
<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

Degree = 1
Coeffs = [-1.96635465]
Intercept = 0.7757179200527622
Degree = 4
Coeffs = [  0.0697848  -14.40575795  18.10888186  -4.68598787]
Intercept = 0.960019225440941
Degree = 30
Coeffs = [-4.52840244e+03  4.57451069e+05 -1.96388216e+07  4.52137630e+08
 -6.41571458e+09  6.07784252e+10 -4.02177363e+11  1.90049292e+12
 -6.42961320e+12  1.52385081e+13 -2.36149011e+13  1.88907611e+13
  3.53909083e+12 -1.96426490e+13  3.36012973e+12  1.80752089e+13
 -1.78125292e+12 -1.79277897e+13 -4.30295114e+12  1.56381708e+13
  1.19344013e+13 -7.79130765e+12 -1.77282172e+13 -2.62017013e+12
  1.64792282e+13  1.26777276e+13 -1.29722474e+13 -1.80761176e+13
  2.16364418e+13 -6.13557594e+12]
Intercept = 9.196001896042334
```

С помощью линейной модели, используя полиномиальные базисные функции седьмого порядка, мы получили прекрасную аппроксимацию этих нелинейных данных. Применение базисных функций в нашей линейной модели делает ее намного гибче, но также и быстро приводит к переобучению. Например, если выбрать слишком высокую степень полинома, то мы получаем неудачное приближение.

1.4 Оптимальность оценок МНК

Теорема. let XX^T – невырождена $\Rightarrow$ оценка МНК $\widehat{\vec{w}} = (XX^T)^{-1}X\vec{\eta}$: 1. несмещенная; 2. эффективная в классе линейных несмещенных оценок $\vec{w}$; 3. Матрица ковариаций оценки МНК: $D(\widehat{\vec{w}}) = \sigma^2(XX^T)^{-1}$.

◁ 1. Несмещенность:

$$M\widehat{\vec{w}} = M\left[(XX^T)^{-1}X\vec{\eta}\right] = M\left[(XX^T)^{-1}X\left(X^T\vec{w} + \vec{\varepsilon}\right)\right] =$$
$$= (XX^T)^{-1}XX^T\vec{w} + (XX^T)^{-1}XX^TM\vec{\varepsilon} = \vec{w}\,.$$

2. Эффективная оценка в классе линейных несмещенных оценок означает следующее. let $\overrightarrow{\zeta} = L\overrightarrow{\eta}$ – произвольная линейная несмещенная оценка $\overrightarrow{w}$. Т. е. $M\overrightarrow{\zeta} = LM\overrightarrow{\eta} = LM\left[X^T\overrightarrow{w} + \overrightarrow{\varepsilon}\right] = LX^T\overrightarrow{w} = \overrightarrow{w}$ и $D\zeta_i \geqslant D\widehat{w}_i$ $(i = \overline{1,n})$. Значит, несмещенность $\overrightarrow{\zeta} \Leftrightarrow LX^T = E$.

Для доказательства эффективности, рассмотрим $D\overrightarrow{\zeta} = LD\overrightarrow{\eta}L^T = \sigma^2 LL^T$.

$$\begin{aligned}
LL^T =&(L-(XX^T)^{-1}X+(XX^T)^{-1}X)(L-(XX^T)^{-1}X+(XX^T)^{-1}X)^T = \\
=&(L-(XX^T)^{-1}X)(L-(XX^T)^{-1}X)^T+(L-(XX^T)^{-1}X)X^T(XX^T)^{-1}+ \\
&+(XX^T)^{-1}X(L^T-X^T(XX^T)^{-1})+(XX^T)^{-1}X\left[(XX^T)^{-1}X\right]^T = \\
=&(L-(XX^T)^{-1}X)(L-(XX^T)^{-1}X)^T+(LX^T-E)(XX^T)^{-1}+ \\
&+(XX^T)^{-1}(XL^T-E)+(XX^T)^{-1}XX^T(XX^T)^{-1} = \\
=&\left\{\ LX^T=E,\quad XL^T=E\ \right\} = \\
=&(L-(XX^T)^{-1}X)(L-(XX^T)^{-1}X)^T+(XX^T)^{-1}.
\end{aligned}$$

Оба слагаемых имеют вид $AA^T \Rightarrow$ у каждого слагаемого неотрицательные диагональные элементы, сумма которых и представляет собой $D\zeta_i$ $(i = \overline{1,n})$ и имеют наименьшие значения при $L = (XX^T)^{-1}X$. Соответствующая эффективная оценка $\widehat{\overrightarrow{\zeta}} = (XX^T)^{-1}X\overrightarrow{\eta}$, совпадающая с оценкой МНК.

3. Матрица ковариаций оценки МНК:

$$\begin{aligned}
D\widehat{\overrightarrow{w}} =&(XX^T)^{-1}XD(\overrightarrow{\eta})X^T(XX^T)^{-1} = (XX^T)^{-1}X(\sigma^2E)X^T(XX^T)^{-1} = \\
=&\sigma^2(XX^T)^{-1}.\triangleright
\end{aligned}$$

1.5 Оценка остаточной дисперсии

1.5.1 Запись квадратичной формы через след

let $z = (z_1,\dots,z_n)^T$; $A = \begin{pmatrix} \dots & \dots & \dots \\ \dots & a_{ij} & \dots \\ \dots & \dots & \dots \end{pmatrix}$, $a_{ij} = a_{ji}$ $(i = \overline{1,n},\ j = \overline{1,n})$

$$\begin{aligned}
z^TAz =& \sum_{i=1}^n\sum_{j=1}^n z_ia_{ij}z_j = \sum_{i=1}^n\left[\sum_{j=1}^n a_{ij}(z_jz_i)\right] = \\
=&\left\{A(z\cdot z^T) = \begin{pmatrix} \dots & \dots & \dots \\ \dots & \sum\limits_{j=1}^n a_{ij}z_jz_k & \dots \\ \dots & \dots & \dots \end{pmatrix}\right\} = \\
=&\mathrm{tr}(Azz^T)
\end{aligned}$$

Из доказательства теоремы о минимизации квадратичной формы МНК $S(\overrightarrow{w})$ решением системы нормальных уравнений:

$$S(\overrightarrow{w}) = S(\widehat{\overrightarrow{w}}) + (\widehat{\overrightarrow{w}} - \overrightarrow{w})^T XX^T(\widehat{\overrightarrow{w}} - \overrightarrow{w}) \Rightarrow$$

$$MS(\overrightarrow{w}) = MS(\widehat{\overrightarrow{w}}) + M\left[(\widehat{\overrightarrow{w}} - \overrightarrow{w})^T XX^T(\widehat{\overrightarrow{w}} - \overrightarrow{w})\right]$$

С другой стороны, по опредению

$$MS(\overrightarrow{w}) = M\left[\overrightarrow{\varepsilon}^T\overrightarrow{\varepsilon}\right] = \left\{\begin{array}{c} \overrightarrow{\varepsilon}^T\overrightarrow{\varepsilon} = \sum\limits_{i=1}^{m}\varepsilon_i^2; M[\varepsilon_i] = 0; D[\varepsilon_i] = \sigma^2 \\ \Rightarrow M\left[\overrightarrow{\varepsilon}^T\overrightarrow{\varepsilon}\right] = \sum\limits_{i=1}^{m} M\left[\varepsilon_i^2\right] = \sum\limits_{i=1}^{m} D[\varepsilon_i] = m\sigma^2 \end{array}\right\} = m\sigma^2$$

Рассмотрим

$$\begin{aligned} M\left[(\widehat{\overrightarrow{w}} - \overrightarrow{w})^T XX^T(\widehat{\overrightarrow{w}} - \overrightarrow{w})\right] =& M\left[tr\left(XX^T(\widehat{\overrightarrow{w}} - \overrightarrow{w})(\widehat{\overrightarrow{w}} - \overrightarrow{w})^T\right)\right] = \\ =& tr\left(XX^T M\left[(\widehat{\overrightarrow{w}} - \overrightarrow{w})(\widehat{\overrightarrow{w}} - \overrightarrow{w})^T\right]\right) \end{aligned}$$

Оценка $\widehat{\overrightarrow{w}}$ – несмещенная $\Rightarrow M\left[\widehat{\overrightarrow{w}}\right] = \overrightarrow{w} \Rightarrow D\left[\widehat{\overrightarrow{w}}\right] = M\left[(\widehat{\overrightarrow{w}} - \overrightarrow{w})(\widehat{\overrightarrow{w}} - \overrightarrow{w})^T\right] = \sigma^2(XX^T)^{-1}$ $\Rightarrow$

$$M\left[(\widehat{\overrightarrow{w}} - \overrightarrow{w})^T XX^T(\widehat{\overrightarrow{w}} - \overrightarrow{w})\right] = \sigma^2 tr\left(XX^T(XX^T)^{-1}\right) = trE \cdot \sigma^2 = n\sigma^2\,.$$

В итоге, имеем

$$m\sigma^2 = MS(\widehat{\overrightarrow{w}}) + n\sigma^2 \Rightarrow MS(\widehat{\overrightarrow{w}}) = (m-n)\sigma^2\,.$$

Т. о. if в качестве оценки σ^2 выбрать $\widehat{\sigma}^2 = \frac{S(\widehat{\overrightarrow{w}})}{m-n}$, то она будет несмещенной оценкой для σ^2:

$$M\left[\widehat{\sigma}^2\right] = \frac{MS(\widehat{\overrightarrow{w}})}{m-n} = \frac{(m-n)\sigma^2}{m-n} = \sigma^2\,.$$

1.6 Регуляризация

1.6.1 Гребневая регрессия ($L2$-регуляризация)

Вероятно, самый часто встречающийся вид регуляризации — гребневая регрессия (ridge regression), или $L2$-регуляризация ($L2$-regularization), также иногда называемая регуляризацией Тихонова (Tikhonov regularization). Она заключается в наложении штрафа на сумму квадратов (евклидовой нормы) коэффициентов модели. В данном случае штраф для модели будет равен:

$$P = \alpha \sum_{k=1}^{n} w_k^2\,,$$

где α — свободный параметр, служащий для управления уровнем штрафа. Этот тип модели со штрафом встроен в библиотеку `Scikit-Learn` в виде оценивателя `Ridge`:

[13]:
```
from sklearn.linear_model import Ridge

poly_model = Ridge(alpha=0.5e-4)
fit_and_draw(poly_model)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
Degree = 1
Coeffs = [-1.96631031]
Intercept = 0.7756999136786734
Degree = 4
Coeffs = [ -0.64203823 -10.7073901   11.8185108   -1.3476647 ]
Intercept = 0.986796508792788
Degree = 30
Coeffs = [-0.82610999 -8.3193884   4.56116154  4.05002638  1.76531704
0.24447651
 -0.46523757 -0.70111183 -0.71581948 -0.64050122 -0.53181246 -0.41248169
 -0.29289016 -0.17964427 -0.07787892  0.00865097  0.07785544  0.12937172
  0.16434367  0.1850294   0.19436066  0.19554015  0.1917205   0.18577774
  0.18017253  0.17688405  0.17739829  0.18273379  0.19349055  0.20991089]
Intercept = 0.9820313158728272
```

Параметр α служит для управления сложностью получаемой в итоге модели. В предельном случае $\alpha \to 0$ мы получаем результат, соответствующий стандартной линейной регрессии; в предельном случае $\alpha \to \infty$ будет происходить подавление любого отклика модели. Достоинства гребневой регрессии включают, помимо прочего, возможность ее эффективного расчета – вычислительные затраты практически не превышают затрат на расчет исходной линейной регрессионной модели.

1.6.2 Лассо-регрессия (L_1 регуляризация)

Еще один распространенный тип регуляризации – так называемая лассо-регуляризация, включающая штрафование на сумму абсолютных значений (L1-норма) коэффициентов регрессии:

$$P = \alpha \sum_{k=1}^{n} |w_n| .$$

Хотя концептуально эта регрессия очень близка к гребневой, результаты их могут очень сильно различаться. Например, по геометрическим причинам лассо-регрессия любит разреженные модели, то есть она по возможности делает коэффициенты модели равными нулю.

Посмотреть на поведение этой регрессии мы можем, воспроизведя график, но с использованием коэффициентов, нормализованных с помощью нормы $L1$:

[14]:
```
from sklearn.linear_model import Lasso

poly_model = Lasso(alpha=1.0e-4, max_iter=10000)
fit_and_draw(poly_model)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
Degree = 1
Coeffs = [-1.9650018]
```

```
Intercept = 0.7751684734880687
Degree = 4
Coeffs = [-1.92815017 -4.16751937  0.88096454  4.3820449 ]
Intercept = 1.036684161709409
Degree = 30
Coeffs = [-1.72864786 -4.51859608  0.          5.91868208  0.          0.
 -0.         -0.         -0.         -0.03492571 -0.44330999 -0.19981369
 -0.         -0.         -0.         -0.         -0.         -0.
 -0.         -0.         -0.          0.          0.          0.
  0.          0.          0.          0.          0.          0.        ]
Intercept = 1.0230776740430572
```

При использовании штрафа лассо-регрессии большинство коэффициентов в точности равны нулю, а функциональное поведение моделируется небольшим подмножеством из имеющихся базисных функций. Как и в случае гребневой регуляризации, параметр α управляет уровнем штрафа и его следует определять путем перекрестной проверки.

1.6.3 Гауссовы базисные функции

Можно использовать и другие базисные функции. Например, один из часто применяющихся примемов – обучение модели, представляющей собой сумму не полиномиальных, а Гауссовых базисных функций.

Эти Гауссовы базисные функции не встроены в библиотеку `Scikit-Learn`, но мы можем написать для их создания пользовательский преобразователь, как показано ниже (преобразователи библиотеки `Scikit-Learn` реализованы как классы языка Python; разработка пользовательского преобразователя – отличный способ разобраться с их созданием):

$$\Phi(x) = e^{-\frac{1}{2}\left(\frac{x-m}{\sigma}\right)^2}$$

[15]:
```
rng = np.random.RandomState(8)
x = 10 * rng.rand(50)
y = np.sin(x) + 0.1 * rng.randn(50)
```

[16]:
```
from sklearn.base import TransformerMixin

class GaussianFeatures(TransformerMixin):
    """Uniformly spaced Gaussian features for one-dimensional input"""

    def __init__(self, N, width_factor=2.0):
        self.N = N
        self.width_factor = width_factor

    @staticmethod
    def _gauss_basis(x, y, width):
        arg = (x - y) / width
        return np.exp(-0.5 * arg ** 2)
```

```
    def fit(self, X, y=None):
        # create N centers spread along the data range
        self.centers_ = np.linspace(X.min(), X.max(), self.N)
        self.width_ = self.width_factor * (self.centers_[1] - self.centers_[0])
        return self

    def transform(self, X):
        return self._gauss_basis(X, self.centers_, self.width_)

X_train = GaussianFeatures(20).fit_transform(x[:,np.newaxis])
gauss_model = LinearRegression()
gauss_model.fit(X_train, y)

xfit = np.linspace(0, 10, 1000)
X_fit=GaussianFeatures(20).fit_transform(xfit[:,np.newaxis])
yfit = gauss_model.predict(X_fit)

plt.figure()
plt.scatter(x, y)
plt.plot(xfit, yfit)
plt.xlim(0, 10)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[16]: (0.0, 10.0)

Мы рассмотрели данный пример лишь для того, чтобы подчеркнуть, что в полиномиальных базисных функциях нет никакого колдунства. Если у вас есть какие-то априорные сведения о процессе генерации ваших данных, исходя из которых есть основания полагать, что наиболее подходящим будет тот или иной набор базовых функций, – то можно использовать его.