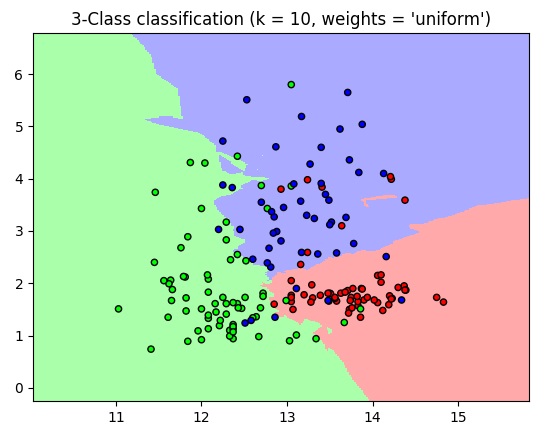
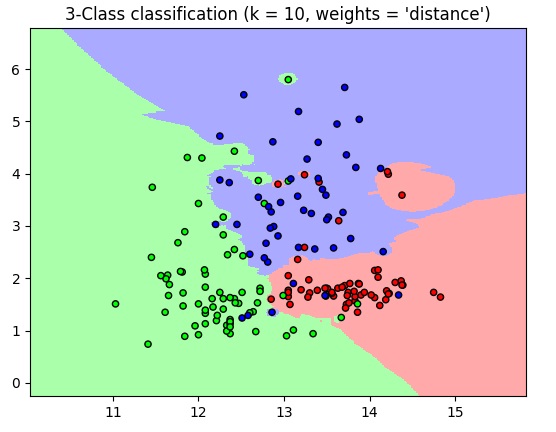
**KNN**

Алгоритм k ближайших соседей относится к классу метрических алгоритмов. Принцип, лежащий в основе методов ближайших соседей, заключается в том, чтобы найти количество объектов, наиболее близких к новой точке, и спрогнозировать принадлежность этой точки классу. Он относит классифицируемый объект к тому классу, которому принадлежит больше объектов, если рассматривать k ближайших соседей из обучающий выборки. k можно задать самостоятельно или применить LOO для нахождения оптимального k.

Код программы:

**import** numpy **as** np  
**import** matplotlib.pyplot **as** plt  
**from** matplotlib.colors **import** ListedColormap  
**from** sklearn **import** neighbors, datasets  
  
print(\_\_doc\_\_)  
  
n\_neighbors = 10  
  
wine = datasets.load\_wine()  
  
X = wine.data[:, :2]  
y = wine.target  
  
h = .02  
  
cmap\_light = ListedColormap([**'#FFAAAA'**, **'#AAFFAA'**, **'#AAAAFF'**])  
cmap\_bold = ListedColormap([**'#FF0000'**, **'#00FF00'**, **'#0000FF'**])  
  
**for** weights **in** [**'uniform'**, **'distance'**]:  
 clf = neighbors.KNeighborsClassifier(n\_neighbors, weights=weights)  
 clf.fit(X, y)  
 x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1  
 y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1  
 xx, yy = np.meshgrid(np.arange(x\_min, x\_max, h), np.arange(y\_min, y\_max, h))  
 Z = clf.predict(np.c\_[xx.ravel(), yy.ravel()])  
 Z = Z.reshape(xx.shape)

Результат работы программы для выборки Ирисы Фишера при количестве соседей - 10



**Формула Надарая – Ватсона**

Формула Надарая – Ватсона используется для решения задач непараметрической регрессии . a(x) вычисляется для каждого объекта по нескольким ближайшим к нему объектам обучающей выборки.

Рассмотрим самую простую модель регрессии — константу *φ(x, α) = α,* *α ∈ Rn*, но при этом каждому объекту *xi* выборки назначим веса согласно весовой функции *wi(x)*, где *x* — объект, в котором мы хотим вычислить значение *a(x) = φ(x, α)*.

Чтобы вычислить оптимальное *α* для произвольного объекта *x* воспользуемся МНК:

Зададим веса таким образом, чтобы они убывали по мере увеличения расстояния до x. Для этого введем ядро — невозрастающую, гладкую, ограниченную функцию *K* : [0, ∞) → [0, ∞):

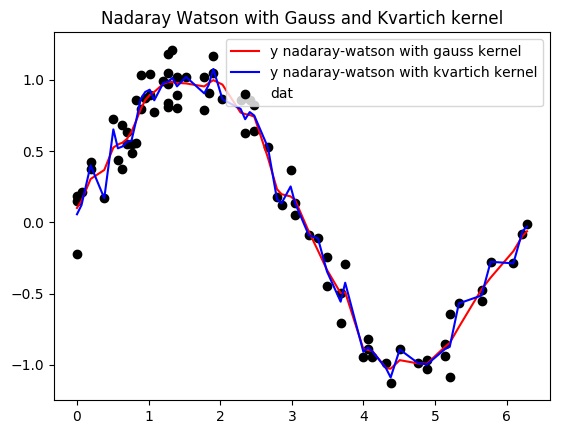
Параметр *h* называется шириной ядра или шириной окна сглаживания.

Приравнивая к нулю производную , получим формулу ядерного сглаживания Надарая-Ватсона:

Код программы:

**import** numpy **as** np  
**from** scipy.spatial **import** distance  
**from** pylab **import** plot, legend, scatter, show, clf, title  
  
**from** sklearn **import** datasets  
  
*# Gauss kernel***def** Gauss(z):  
 **return** (1.0 / np.sqrt(2 \* np.pi)) \* np.exp(- 0.5 \* z \*\* 2)  
  
*# Kvartich kernel***def** Kvartich(z):  
 **if** abs(z) <= 1:  
 **return** (1 - z \*\* 2) \*\* 2  
 **else**:  
 **return** 0  
  
**def** nadaray\_watson(x, y, kernel=**"Gauss"**, h=0.5):  
 possibles = globals().copy()  
 possibles.update(locals())  
 K = possibles.get(kernel)  
  
 n = len(x)  
 w = []  
 **for** t **in** range(n):  
 w.append([])  
 **for** i **in** range(n):  
 w[t].append(K(distance.euclidean(x[t], x[i]) / h))  
 w = np.array(w)  
 yest = (w \* y[:, **None**]).sum(axis=0) / w.sum(axis=0)  
 **return** yest  
  
  
**def** SSE(y\_pred, y\_real):  
 res = y\_pred - y\_real  
 res \*= res  
 **return** np.sum(res)  
  
  
**def** generate\_wave\_set(n\_support=1000, n\_train=250):  
 data = {**'support'**: np.linspace(0, 2 \* np.pi, num=n\_support)}  
 data[**'x\_train'**] = np.sort(np.random.choice(data[**'support'**], size=n\_train, replace=**True**))  
 data[**'y\_train'**] = np.sin(data[**'x\_train'**]).ravel()  
 data[**'y\_train'**] += 0.5 \* (0.55 - np.random.rand(data[**'y\_train'**].size))  
 **return** data  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 data = generate\_wave\_set(100, 80)  
 x = data[**'x\_train'**]  
 y = data[**'y\_train'**]  
 boston = datasets.load\_boston()

Результат работы. В программе был реализован метод с гауссовским и квартическим ядрами на выборке содержащей цены домов в Бостоне.



Ошибка при h = 3 для гауссовского ядра составила 1.13448272531, для квартического 0.728463283908.

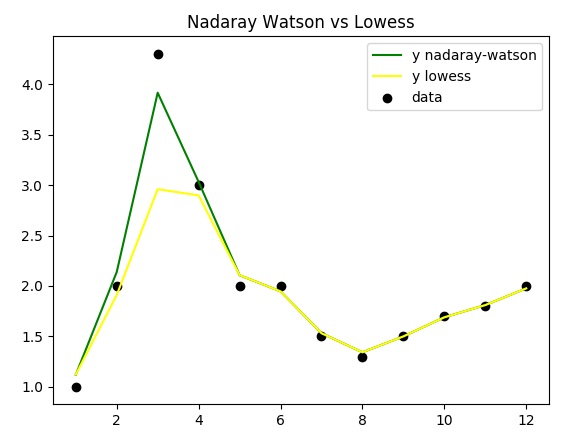
**Метод LOWESS**

Оценка Надарая–Ватсона крайне чувствительна к большим одиночным выбросам. Идея обнаружения выбросов заключается в том, что чем больше величина ошибки , тем в большей степени прецедент является выбросом, и тем меньше должен быть его вес. Эти соображения приводят к идее домножить веса на коэффициенты , где — ещё одно ядро, вообще говоря, отличное от .

Код программы:

**import** numpy **as** np  
**from** scipy.spatial **import** distance  
**import** pylab **as** plt  
**from** sklearn.metrics **import** mean\_squared\_error  
  
*# Gauss kernel***def** Gauss(z):  
 **return** (1.0 / np.sqrt(2 \* np.pi)) \* np.exp(- 0.5 \* z \*\* 2)  
*# Kvartich kernel***def** Kvartich(z):  
 **if** abs(z) <= 1:  
 **return** (1 - z \*\* 2) \*\* 2  
 **else**:  
 **return** 0  
  
**def** stability(arr, eps):  
 **for** el **in** arr:  
 **if** el > eps:  
 **return True  
 return False  
  
def** nadaray\_watson(x, y, kernel=**"Gauss"**, h=0.5):  
 possibles = globals().copy()  
 possibles.update(locals())  
 K = possibles.get(kernel)  
  
 n = len(x)  
 w = []  
 **for** t **in** range(n):  
 w.append([])  
 **for** i **in** range(n):  
 w[t].append(K(distance.euclidean(x[t], x[i]) / h))  
 w = np.array(w)  
 yest = (w \* y[:, **None**]).sum(axis=0) / w.sum(axis=0)  
 **return** yest  
  
  
**def** lowess(x, y, kernel=**"Gauss"**, kernel1=**"Kvartich"**, h=0.5, eps=1e-5):  
 possibles = globals().copy()  
 possibles.update(locals())  
 K = possibles.get(kernel)  
 K1 = possibles.get(kernel1)  
 n = len(x)  
  
 gamma = np.ones(n)  
 gamma\_old = np.zeros(n)  
 yest = np.zeros(n)  
 cnt = 0  
  
 **while** stability(np.abs(gamma - gamma\_old), eps):  
 cnt += 1  
 *# find weights as multiplication of gammas and kernel function on dist* w = []  
 **for** t **in** range(n):  
 w.append([])  
 **for** i **in** range(n):  
 w[t].append(K(distance.euclidean(x[t], x[i]) / h) \* gamma[t])  
 w = np.array(w)  
 yest = (w \* y[:, **None**]).sum(axis=0) / w.sum(axis=0)  
  
 *# calc new gammas as Kernel function(error)* err = np.abs(yest - y)  
 gamma = [K1(err[j]) **for** j **in** range(n)]  
 **if** cnt > 5:  
 **break  
 return** yest  
  
  
**def** SSE(y\_pred, y\_real):  
 res = y\_pred - y\_real  
 res \*= res  
 **return** np.sum(res)  
  
  
**def** generate\_wave\_set(n\_support=1000, n\_train=250):  
 data = {**'support'**: np.linspace(0, 2 \* np.pi, num=n\_support)}  
 data[**'x\_train'**] = np.sort(np.random.choice(data[**'support'**], size=n\_train, replace=**True**))  
 data[**'y\_train'**] = np.sin(data[**'x\_train'**]).ravel()  
 data[**'y\_train'**] += 0.5 \* (0.55 - np.random.rand(data[**'y\_train'**].size))  
 **return** data  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 x = [[1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12]]  
 x = np.array(x)  
 y = [1, 2, 4.3, 3, 2, 2, 1.5, 1.3, 1.5, 1.7, 1.8, 2]  
 y = np.array(y)

На данном графике видно, что алгоритм LOWESS сглаживает лучше:



**Гребневая регрессия**

Гребневая регрессия или гребневая регрессия - это один из методов понижения размерности. Часто его применяют для борьбы с мультиколлинеарностью (наличие линейно зависимых столбцов матрицы). Для решения проблемы необходимо добавить к функционалу 𝑄 регуляризатор, штрафующий большие значения нормы вектора весов ||α||:

𝑄𝑟(𝛼)=||𝐹𝛼−𝑦||2+𝜏||𝛼||2, где 𝜏 – неотрицательный параметр регуляризации.

Код программы:

**import** numpy **as** np  
**import** matplotlib.pyplot **as** plt  
**from** sklearn.metrics **import** mean\_squared\_error  
  
  
**def** ridge(X, y, cov\_matrix, n, tau=0.01):  
 cov\_matrix\_ridge = cov\_matrix + tau \* np.eye(n)  
 print(cov\_matrix\_ridge)  
 eigenval\_ridge, eigenvect\_ridge = np.linalg.eigh(cov\_matrix\_ridge)  
 print(**"new eigen values:"**)  
 print(**", "**.join(**"%.2f"** % f **for** f **in** eigenval\_ridge))  
  
 **return** np.linalg.inv(cov\_matrix\_ridge).dot(X.T).dot(y)  
  
  
**def** SSE(y\_pred, y\_real):  
 **return** np.linalg.norm(y\_pred - y\_real) \*\* 2  
  
  
**def** generate\_wave\_set(n\_support=1000, n\_train=250):  
 data = {}  
 data[**'support'**] = np.linspace(0, 2 \* np.pi, num=n\_support)  
 data[**'x\_train'**] = np.sort(np.random.choice(data[**'support'**], size=n\_train, replace=**True**))  
 data[**'y\_train'**] = np.sin(data[**'x\_train'**]).ravel()  
 data[**'y\_train'**] += 0.5 \* (0.55 - np.random.rand(data[**'y\_train'**].size))  
 **return** data  
  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 n = 20  
 data = generate\_wave\_set(100, n)  
 X = data[**'x\_train'**]  
 y = data[**'y\_train'**]  
  
 fX = np.array([X]).T  
 fX = np.concatenate((fX, np.power(fX, 2), np.power(fX, 3), 2 \* fX, 3 \* fX), axis=1)  
  
 cov\_matrix\_standart = fX.T.dot(fX)eigenval, eigenvect = np.linalg.eigh(cov\_matrix\_standart)  
 ans = ridge(fX, y, cov\_matrix\_standart, n=5, tau=0.3)

Имеется выборка с линейно зависимыми признаками. Ее ковариационная матрица имеет вид:

[[ 180.12808457 731.58833996 3235.92520849 360.25616914 540.3842537 ]

[ 731.58833996 3235.92520849 15164.78229118 1463.17667991 2194.76501987]

[ 3235.92520849 15164.78229118 74107.77282671 6471.85041698 9707.77562547]

[ 360.25616914 1463.17667991 6471.85041698 720.51233827 1080.76850741]

[ 540.3842537 2194.76501987 9707.77562547 1080.76850741 1621.15276111]]

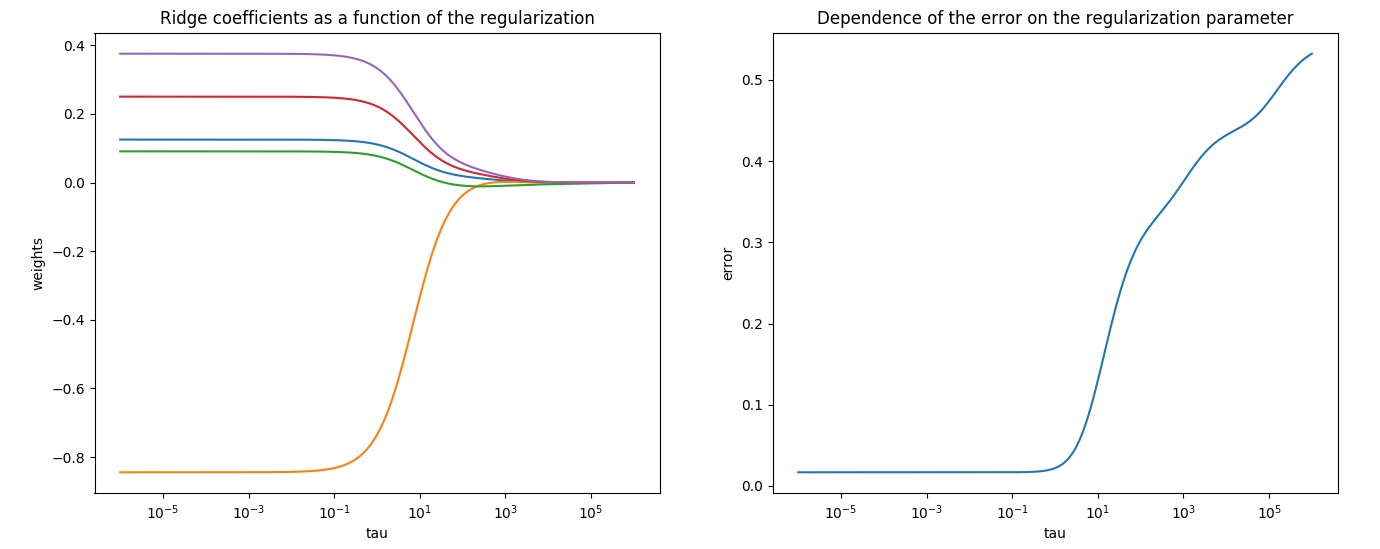
а собственные значения (первые два нулевые):

-0.00, 0.00, 6.93, 634.12, 79224.44

Добавим "гребень" с параметром  . Получим новые собственные значения, которые уже не будут нулевыми:

0.30, 0.30, 7.23, 634.42, 79224.74

В результате работы программы получим графики



**Нелинейная модель регрессии**

Нелинейная регрессия  - частный случай регрессионного анализа, в котором рассматриваемая регрессионная модель есть функция, зависящая от параметров и от одной или нескольких свободных переменных. Зависимость от параметров предполагается нелинейной.

Код программы:

**import** numpy **as** np  
**import** matplotlib.pyplot **as** plt  
**from** scipy **import** optimize  
  
**def** func(x, val):  
 c0, c1 = val  
 **return** c0 + np.log(x)\*c1  
  
  
**def** derivative(x, y, val):  
 c0, c1 = val  
 f = c0 + c1\*np.log(x) - y  
 dc0 = f  
 dc1 = f\*np.log(x)  
 **return** np.array([dc0, dc1])  
  
**def** hessian(x, y, val):  
 c0, c1 = val  
 val\_cnt = 2  
 hes = np.empty((val\_cnt, val\_cnt))  
 grad = derivative(x, y, val)  
 f = func(x, val)  
  
 hes[0, 0] = 0  
 hes[0, 1] = 0  
 hes[1, 0] = 0  
 hes[1, 1] = 0  
  
 **for** i **in** range(val\_cnt):  
 **for** j **in** range(val\_cnt):  
 hes[i, j] = grad[i] \* grad[j] + (f - y) \* hes[i, j]  
 **return** hes  
  
  
**def** SSE(x, y, val):  
 res = 0  
 **for** i **in** range(len(x)):  
 res += 0.5\*(func(x[i], val) - y[i])\*\*2  
 **return** res  
  
  
**def** stability(arr, eps):  
 **for** el **in** arr:  
 **if** el > eps:  
 **return True  
 return False  
  
  
def** newton\_raphson(x, y, maxcnt=100, eps=1e-5):  
 params\_cnt = 2  
 params = np.random.rand(params\_cnt)  
 params\_old = params+10  
 cnt = 0  
 **while** (stability(np.abs(params-params\_old), eps) **and** cnt < maxcnt):  
 grad = np.zeros(params\_cnt)  
 hes = np.zeros((params\_cnt, params\_cnt))  
 **for** i **in** range(len(X)):  
 grad += derivative(X[i], y[i], params)  
 hes += hessian(X[i], y[i], params)  
 *# find best step gamma* f = **lambda** val: SSE(x, y, params-val\*np.linalg.pinv(hes).dot(grad))  
 gamma = optimize.brent(f)  
  
 params\_old = params.copy()  
 *# make iteration as difference of old params value and  
 # step \* gradient \* inverse(hessian)* params -= np.linalg.pinv(hes).dot(grad) \* gamma  
 cnt += 1  
 print(**"iteration cnt: "**, cnt)  
 **return** params  
  
**def** gradient\_descent(X, y, maxcnt = 100, eps=1e-5):  
 params\_cnt = 2  
 params = np.random.randn(params\_cnt)  
 params\_old = params+10  
 cnt = 0  
 **while** (stability(np.abs(params-params\_old), eps) **and** cnt < maxcnt):  
 grad = np.zeros(params\_cnt)  
 **for** i **in** range(len(X)):  
 grad += derivative(X[i], y[i], params)  
 *# find best step gamma* f = **lambda** val: SSE(X, y, params - val \* grad)  
 gamma = optimize.brent(f)  
  
 params\_old = params.copy()  
 params -= grad \* gamma  
 cnt += 1  
 print(**"iteration cnt: "**, cnt)  
 **return** params  
  
**def** generate\_set(n = 100):  
 data = {}  
 data[**'x'**] = np.linspace(1, n, num=n)  
 data[**'y'**] = -2\*np.random.rand(data[**'x'**].size) + np.log(data[**'x'**])\*3.3  
 **return** data  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 n = 20  
 data = generate\_set(n)  
 X = data[**'x'**]  
 y = data[**'y'**]  
 params\_vec = gradient\_descent(X, y)   
 yest = [func(X[i], params\_vec) **for** i **in** range(n)]  
 params\_vec = newton\_raphson(X, y)  
 yest = [func(X[i], params\_vec) **for** i **in** range(n)]

