Отчёт

Гуляев Сергей Владимирович

группа: 501-И

Языки программирования:

- **Python**

Используемые пакеты:

- **numpy**
  Пакет для научных вычислений
- **pandas**
  Пакет для быстрого и простого хранения, структурирования и анализа данных
- **sklearn**
  Пакет содержащий простые и эффективные инструменты для сбора и анализа данных
- **bokeh**
  Пакет для создания интерактивной визуализации данных, для отображения в современных браузерах.

Оглавление

Метрические алгоритмы классификации

Пусть задано множество объектов X и конечное множество классов Y , при этом существует зависимость $y^*: X \to Y$, известная для некоторых объектов конечного множества $X^l \in X$ где l мощность множества X^l . Также пусть задана некоторая функция расстояния $\rho: X \times X \to \mathbb{R}^+ \cup \{0\}$.Требуется построить алгоритм, как можно более точно аппроксимирующий функцию $y^*(x)$ на всё множество X .

Далее в работе над данным методом в качестве функции расстояния будет использоваться функция $\rho(x, y) = \sum (x_i + y_i)^2$ а алгоритмы классификации взяты из пакета **sklearn**

Метрический алгоритм классификации с обучающей выборкой X^l относит объект x к тому классу $y \in Y$, для которого суммарный вес обучающих объектов $\Gamma_y(x, X^l)$ максимален. $\Gamma_y(x, X^l)$ называется *оценкой близости объекта x к классу y*

$$a(x; X^l) = arg\max_{y \in Y} \Gamma_y(x, X^l); \qquad \Gamma_y(x, X^l) = \sum_{i=1}^{l} [y_x^{(i)} = y] w(i, x);$$

Где функция $w(i, x)$ является коэффициентом важности соседа i для объекта x .

Недостатки:

- Алгоритм хранит выборку на всё время работы
- Малое количество параметров для настройки алгоритма

– Метод ближайших соседей

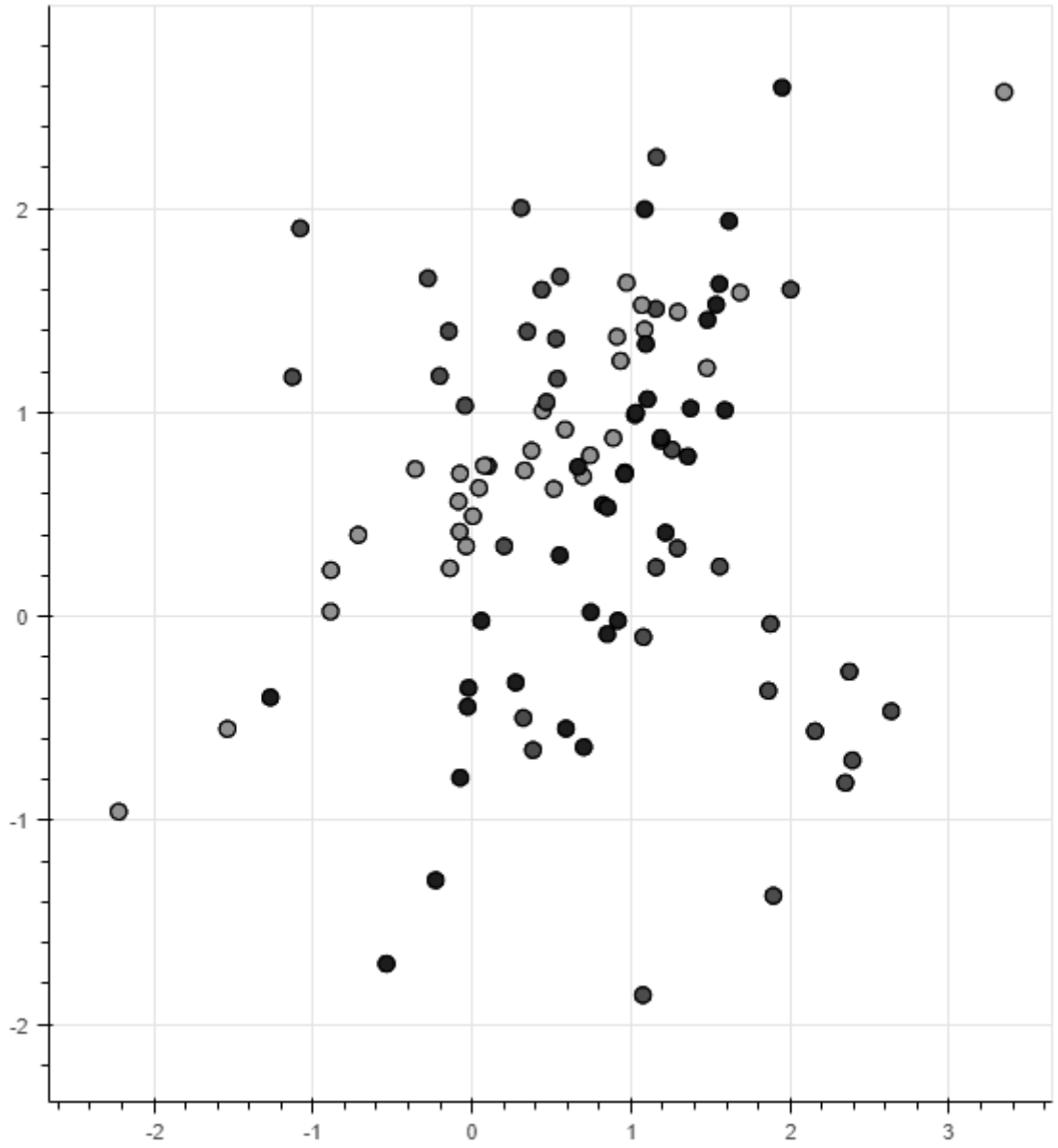
Рассмотрим алгоритм классификации, который относит объект к тому классу, элементов которого окажется больше среди k его ближайших соседей.

В пакете **sklearn** данный классификатор описан в классе **KneighborsClassifier**

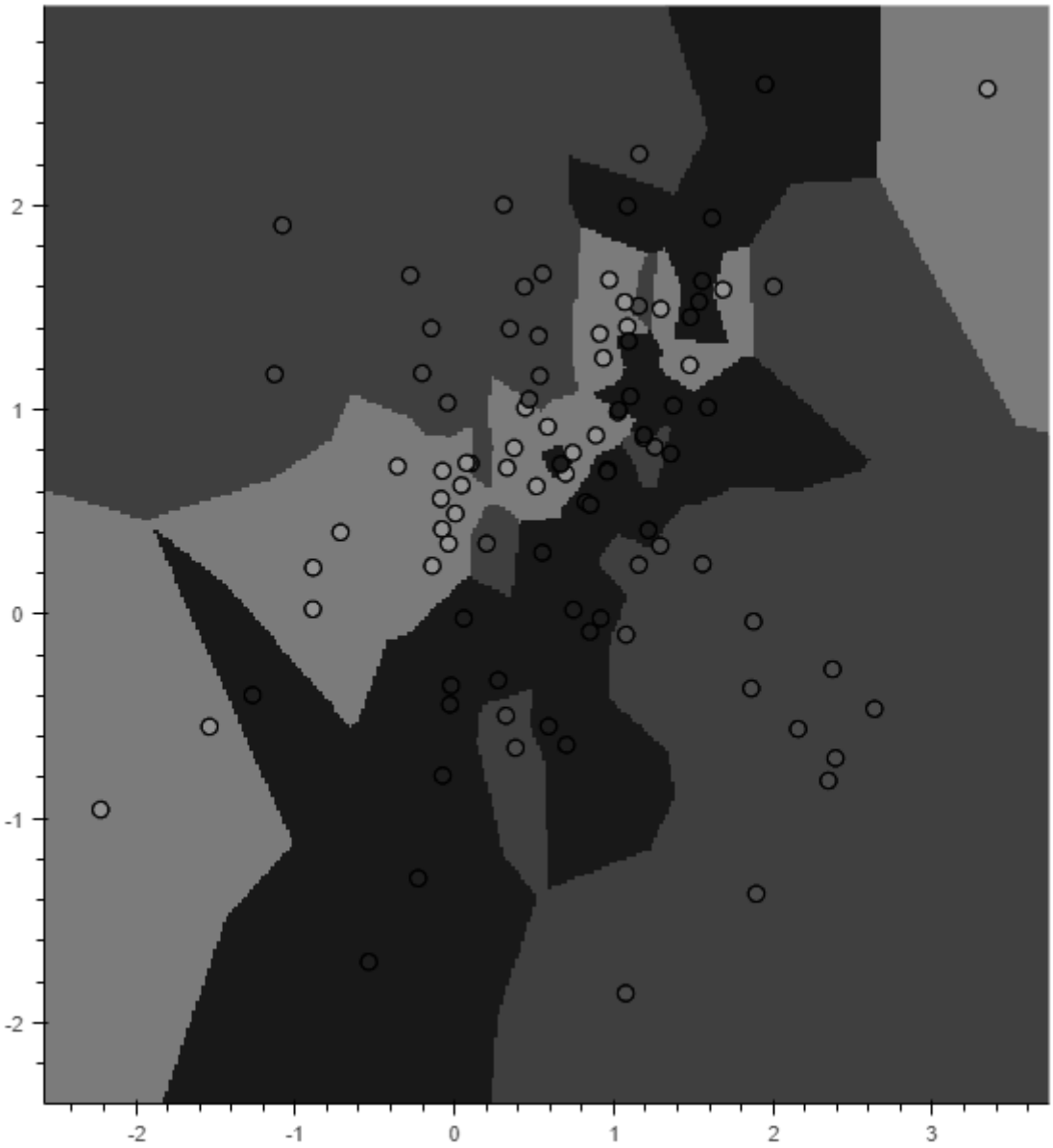
KneighborsClassifier при инициализации позволяет установить не только k **,** но также метрику, алгоритм внутренней обработки данных и функцию весов w .

```
n_neighbors,cross_validation_result = leave_one_out(X,Y)
clf = neighbors.KNeighborsClassifier(n_neighbors, weights='distance')
clf.fit(X, Y)
```

Для классификации будет использоваться данная выборка:

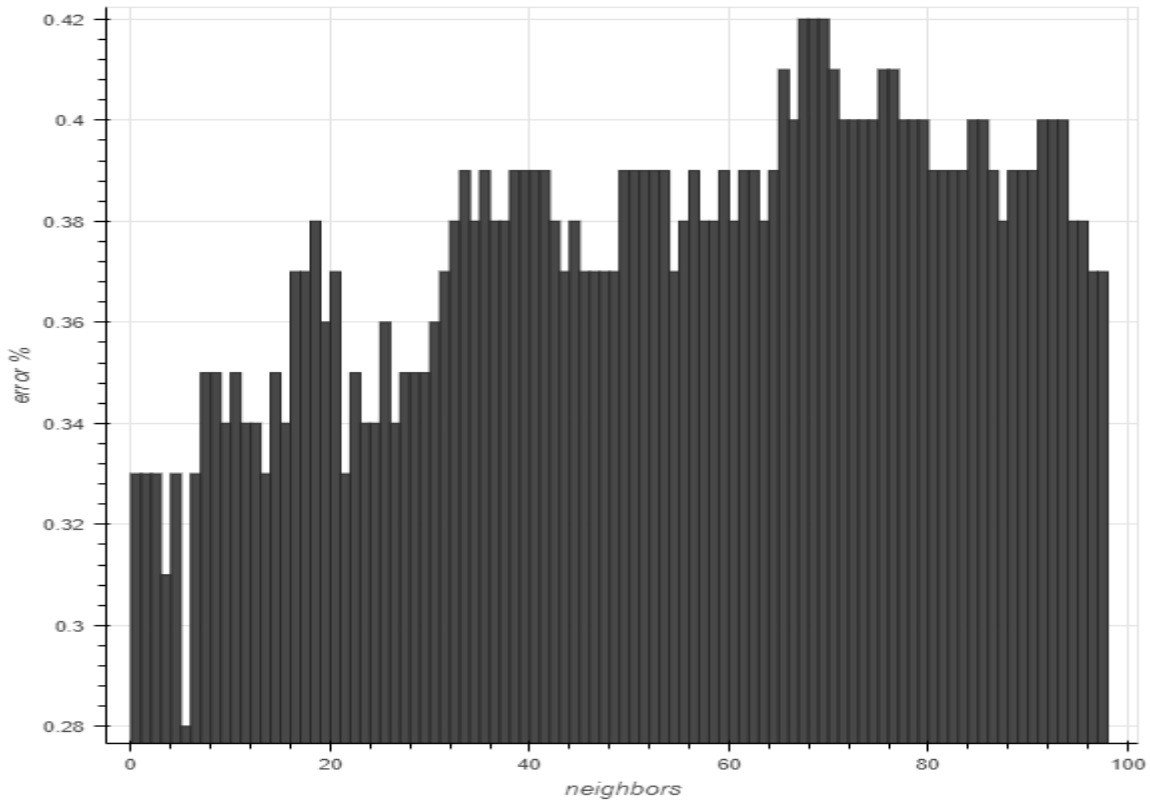


Как можно заметить по следующему рисунку при значении $k=1$, данный метод будет очень чувствителен к выбросам. Поэтому для определения оптимального значения k будет использоваться функция LOO(leave one out):

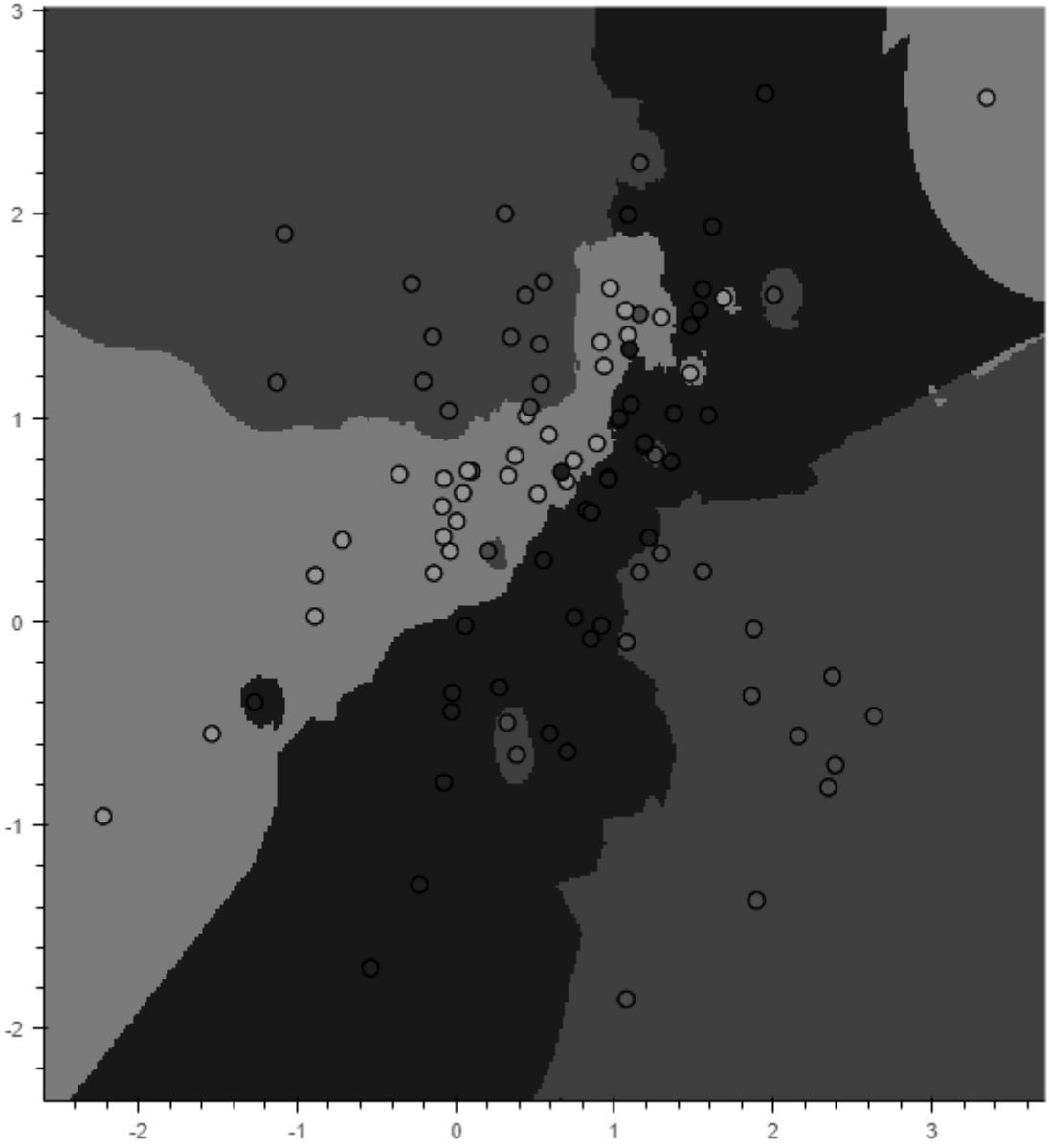


```
def leave_one_out(X,Y):
    item_amount = len(X)
    error_percentage = []
    minimal_good_neighbors = 0
    minimal_errors = len(X)
    for n_neighbors in range(1,item_amount - 1):
        errors = 0
        for i in range(item_amount):
            item = X[i]
            item_class = Y[i]
            X_t = np.delete(X,i,0)
            Y_t = np.delete(Y,i,0)
            clf = neighbors.KNeighborsClassifier(n_neighbors, weights='distance')
            clf.fit(X_t, Y_t)
            predicted_class = clf.predict(item.reshape(1, -1))
            if(predicted_class != item_class):
                errors = errors + 1
        error_percentage.append(errors/item_amount)
        if(errors<minimal_errors):
            minimal_errors = errors
            minimal_good_neighbors = n_neighbors
    return minimal_good_neighbors,error_percentage
```

После работы функции мы можем увидеть какой процент ошибок классификации на существующей выборке возникал при каждом k .



Также видим , что при выборе оптимального $k=6$ метод стал значительно менее чувствителен к выбросам.



– Метод окна Парзена

Рассмотрим алгоритм классификации, который относит объект к тому классу, элементов которого окажется больше среди его соседей находящихся на определённом расстоянии.

В пакете **sklearn** данный классификатор описан в классе **RadiusNeighborsClassifier**

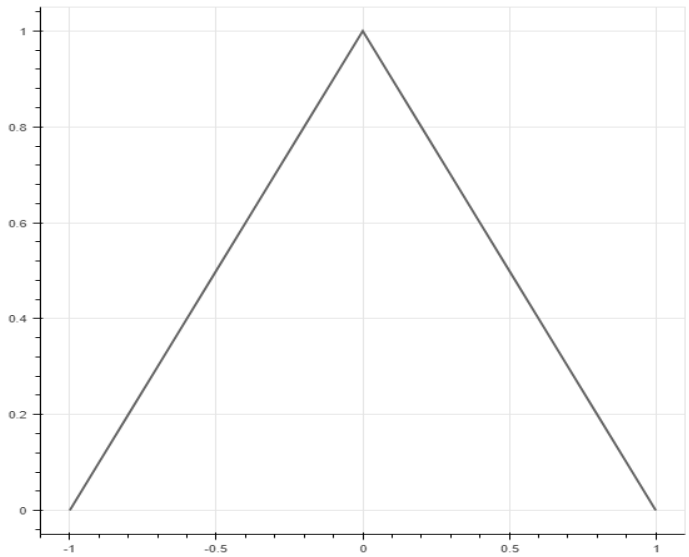
RadiusNeighborsClassifier при инициализации позволяет установить не только ширину окна**,** но также метрику, алгоритм внутренней обработки данных и функцию весов w , как и в случае с методом ближайших соседей.

В разборе данного метода будут применены два ядра:

- Ядро расстояния:

```
def kernel_distance(weights):
    if(type(weights[0]) == type(1.1)):
        res = np.ndarray(shape = (1))
        res[0] = 0
        return res
    else:
        maxDist = 0
        for x in weights[0]:
            maxDist = max(maxDist,x)
        return np.asarray([np.asarray([1-(abs(x)/maxDist) for x in weights[0]])])
```

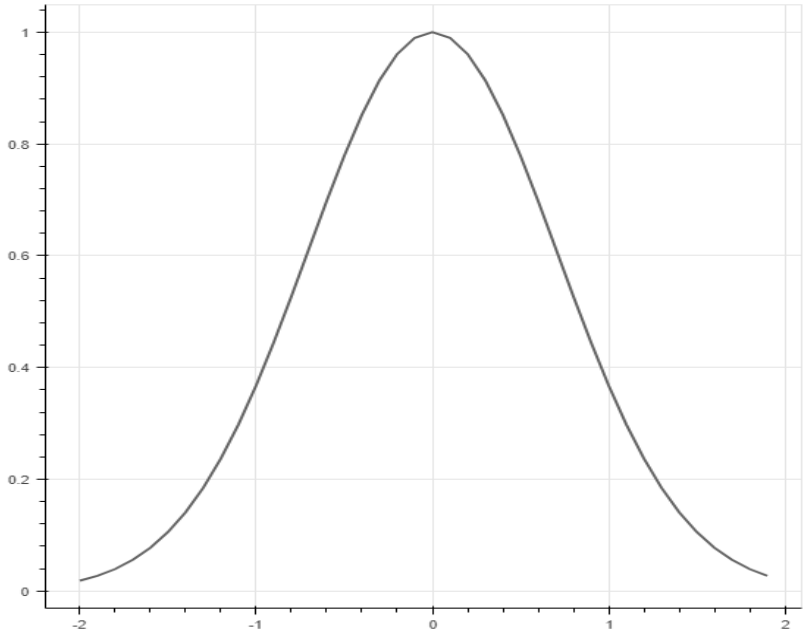
Имеющее такой график:



- Ядро RBF:

```
def kernel_rbf(weights):
    if(type(weights[0]) == type(1.1)):
        res = np.ndarray(shape = (1))
        res[0] = 0
        return res
    else:
        return np.asarray([np.asarray([metrics.pairwise.rbf_kernel([[x]],[[0]])[0][0] for x in weights[0]])])
```

Имеющее такой график:

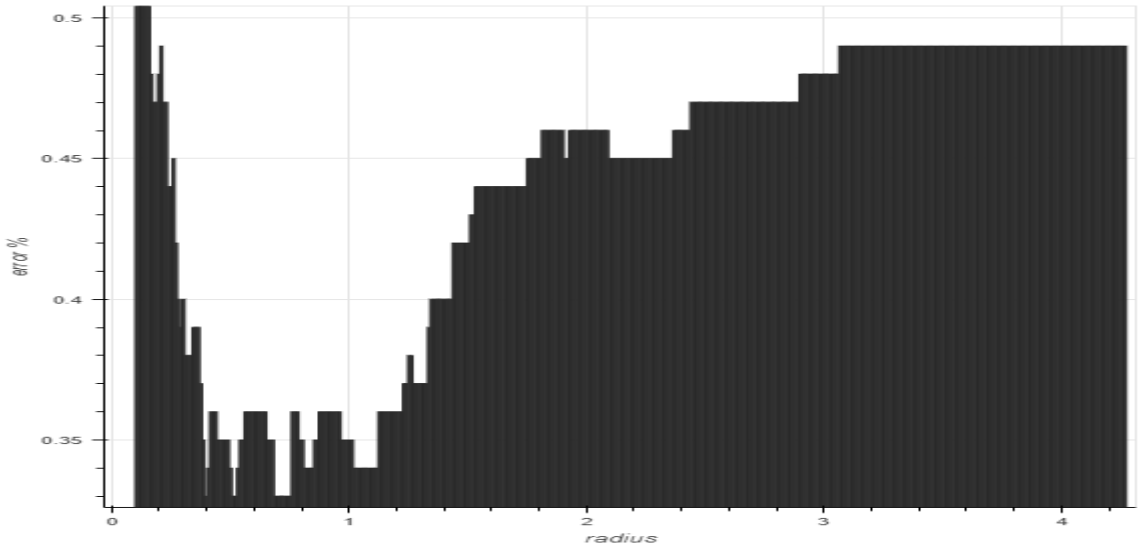


Для выбора оптимальной ширины окна будет использован алгоритм loo для окна Парзена.

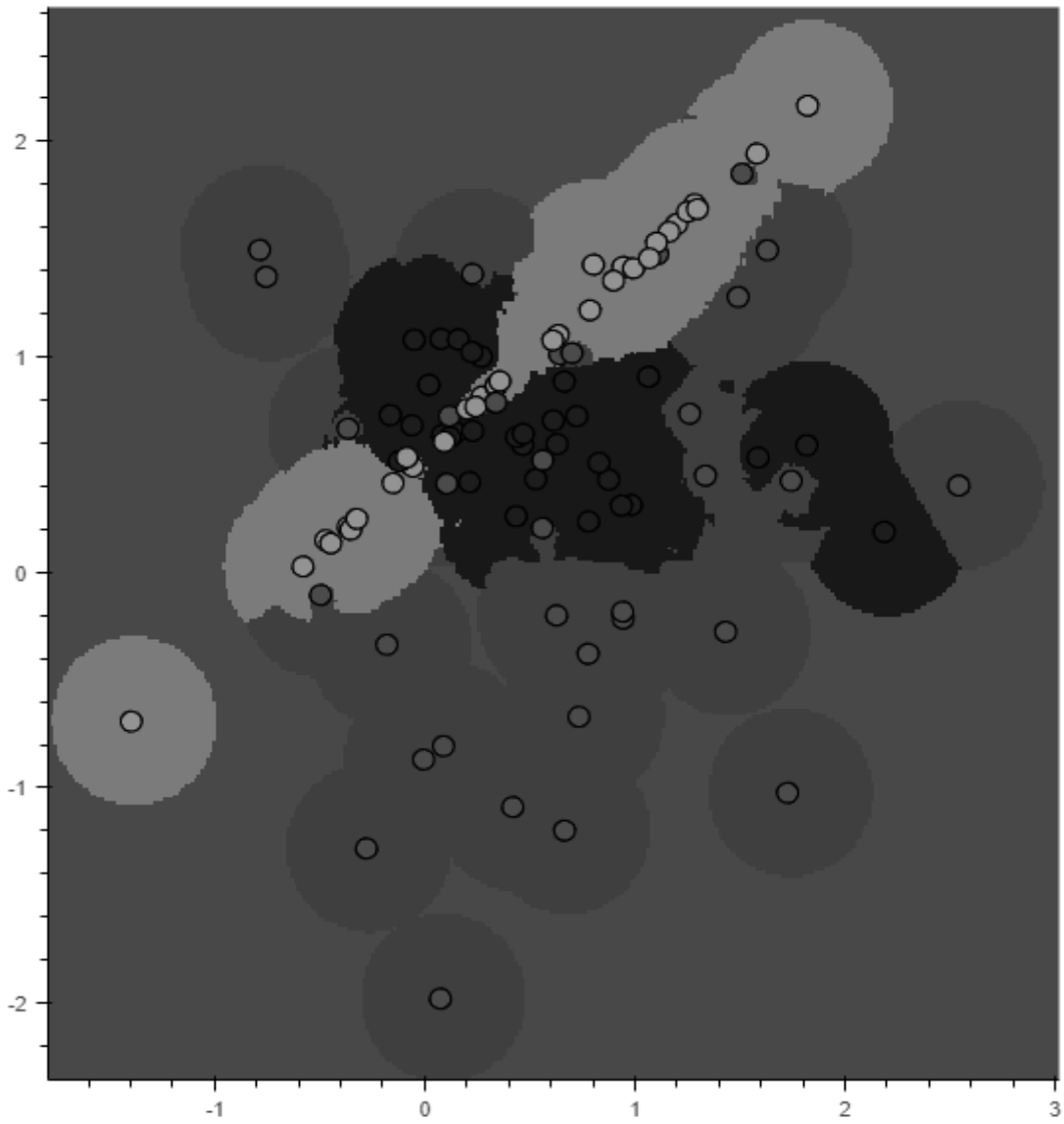
```
def leave_one_out(X,Y,start,end,step,weight):
    item_amount = len(X)
    error_percentage = []
    minimal_good_radius = 0
    minimal_errors = len(X)
    for radius in frange(start,end, step):
        errors = 0
        for i in range(item_amount):
            item = X[i]
            item_class = Y[i]
            X_t = np.delete(X,i,0)
            Y_t = np.delete(Y,i,0)
            clf = neighbors.RadiusNeighborsClassifier(radius, weights=weight,outlier_label=-2)
            clf.fit(X_t, Y_t)
            predicted_class = clf.predict(item.reshape(1, -1))
            if(predicted_class != item_class):
                errors = errors + 1
        error_percentage.append(errors/item_amount)
        if(errors<minimal_errors):
            minimal_errors = errors
            minimal_good_radius = radius
    return minimal_good_radius,error_percentage
```

Для ядра расстояния получился такой график процента ошибки относительно ширины окна:

В

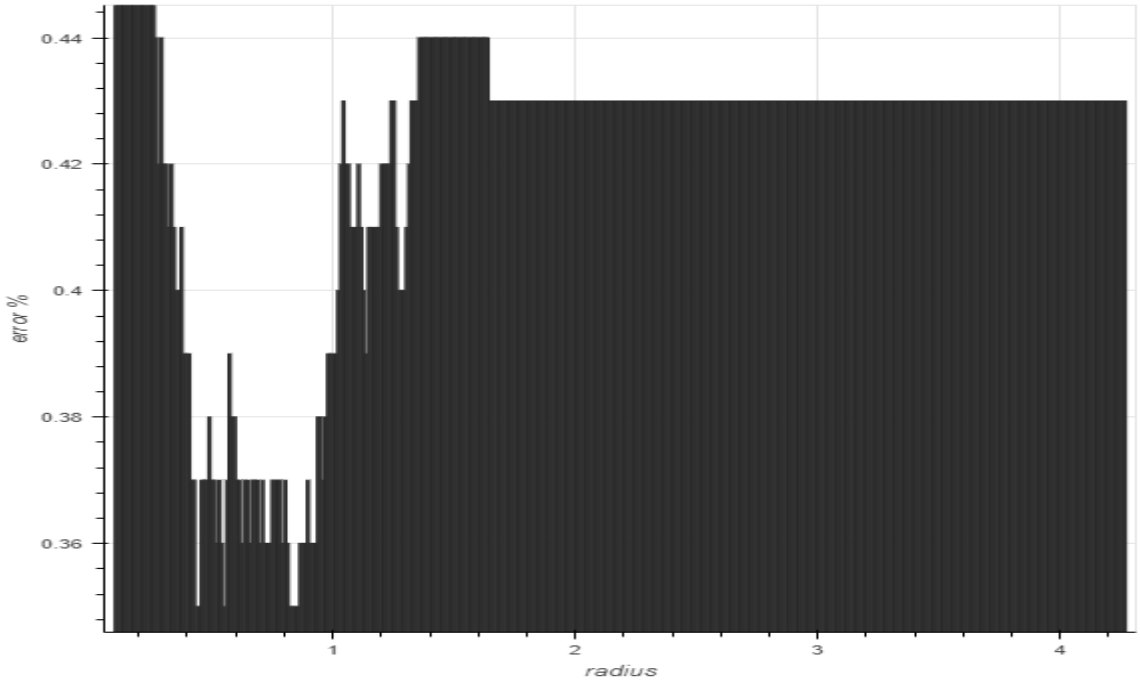


следствии чего шириной окна было выбрано значение 0.39209

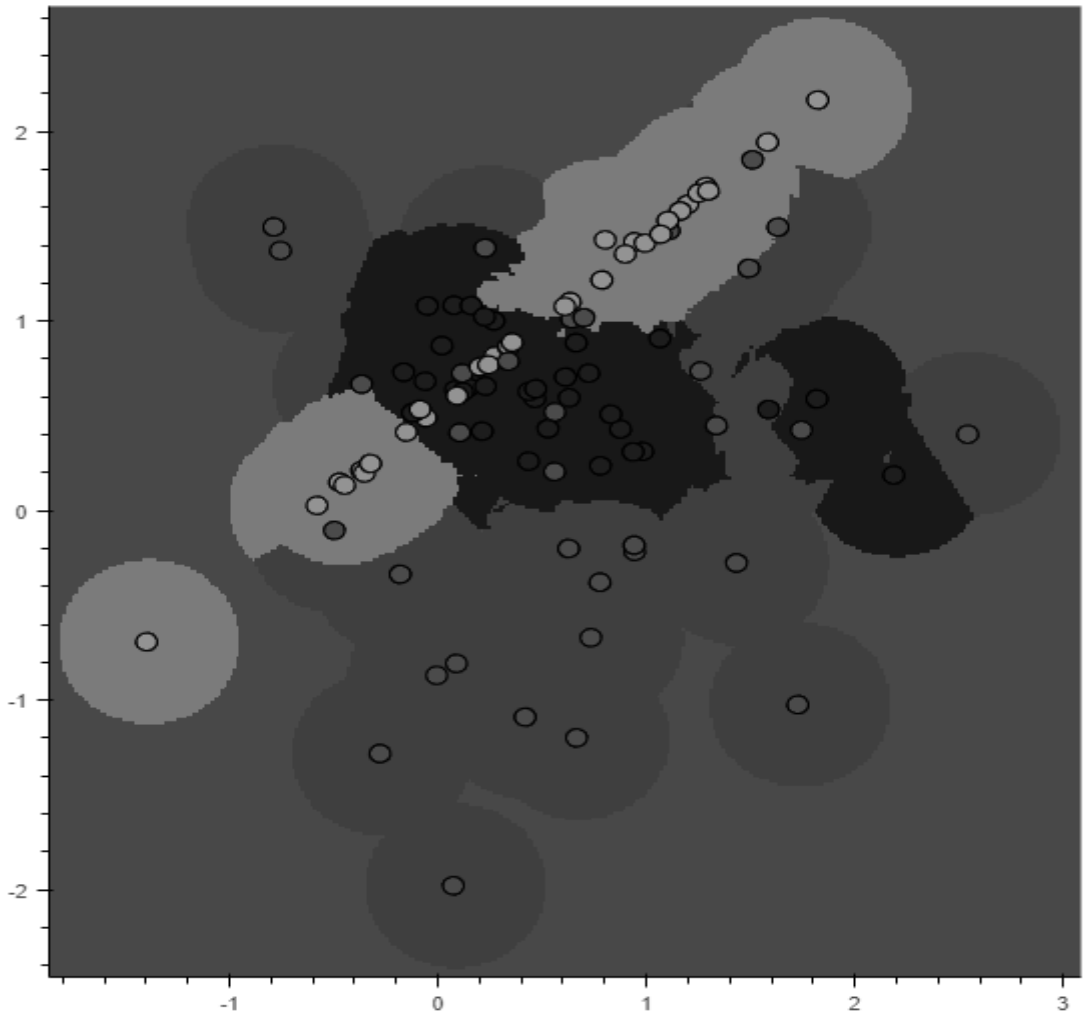


Для ядра RBF получился такой график процента ошибки относительно ширины окна:

В



следствии чего шириной окна было выбрано значение 0.433817



Байесовские методы классификации

Идея данных методов заключается в теореме утверждающей, что если известны плотности распределения классов, то алгоритм классификации a можно выписать в явном виде.

$$a(x)=arg\max_{y\in Y}\lambda_y P_y p_y(x)$$

– Наивный Байесовский классификатор

Данный классификатор основан на гипотезе о том, что если объекты $x\in X$ описываются n числовыми признаками $f_1(x),\dots,f_n(x)$, и эти признаки независимые случайные величины, то функции правдоподобия классов представлены в виде:

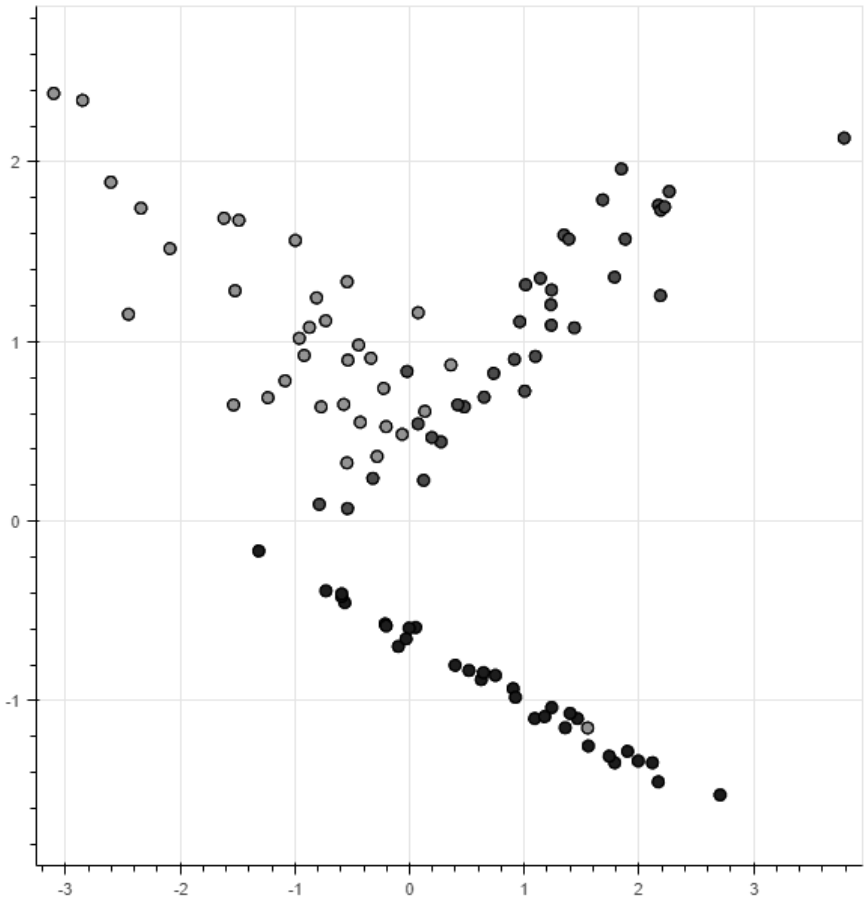
$$p_y(x)=p_{y1}(\xi_1)\dots p_{yn}(\xi_n), Y\in Y$$

где $\xi_j=f_j(x)$, а $p_{yj}(\xi_j)$ - плотность распределения j-го признака для класса y.

В пакете **sklearn** данный классификатор описан в классе **GaussianNB**

Класс **GaussianNB** позволяет установить априорные вероятности для каждого из исследуемых классов.

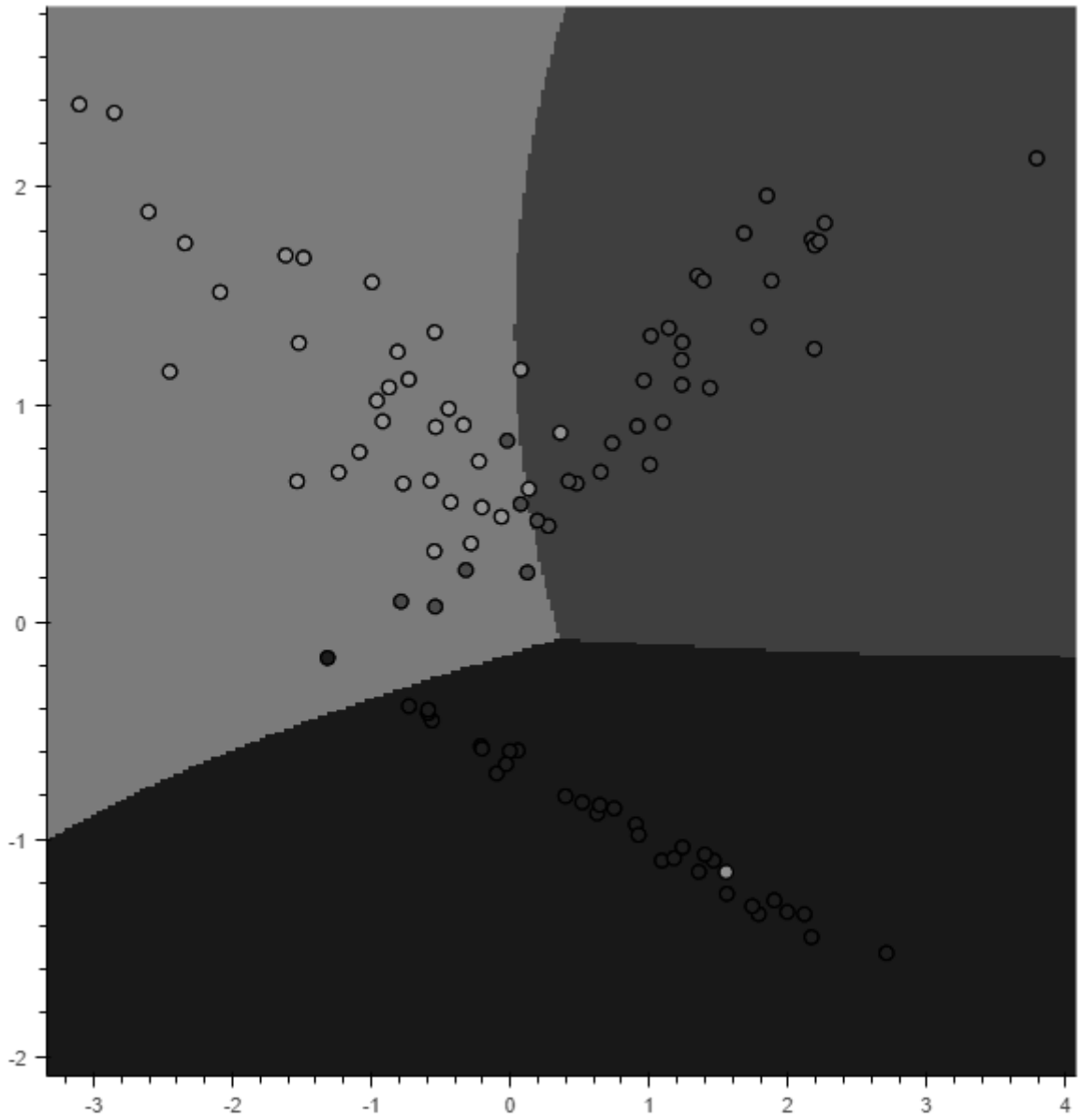
Рассмотрим работу данного алгоритма на показанной ниже выборке.



Так как нам не известны дополнительные параметры, а именно априорные вероятности классов, то инициализация класса **GaussianNB** будет выглядеть так:

```
clf = GaussianNB()
clf.fit(X,Y)
```

Результат работы алгоритма:



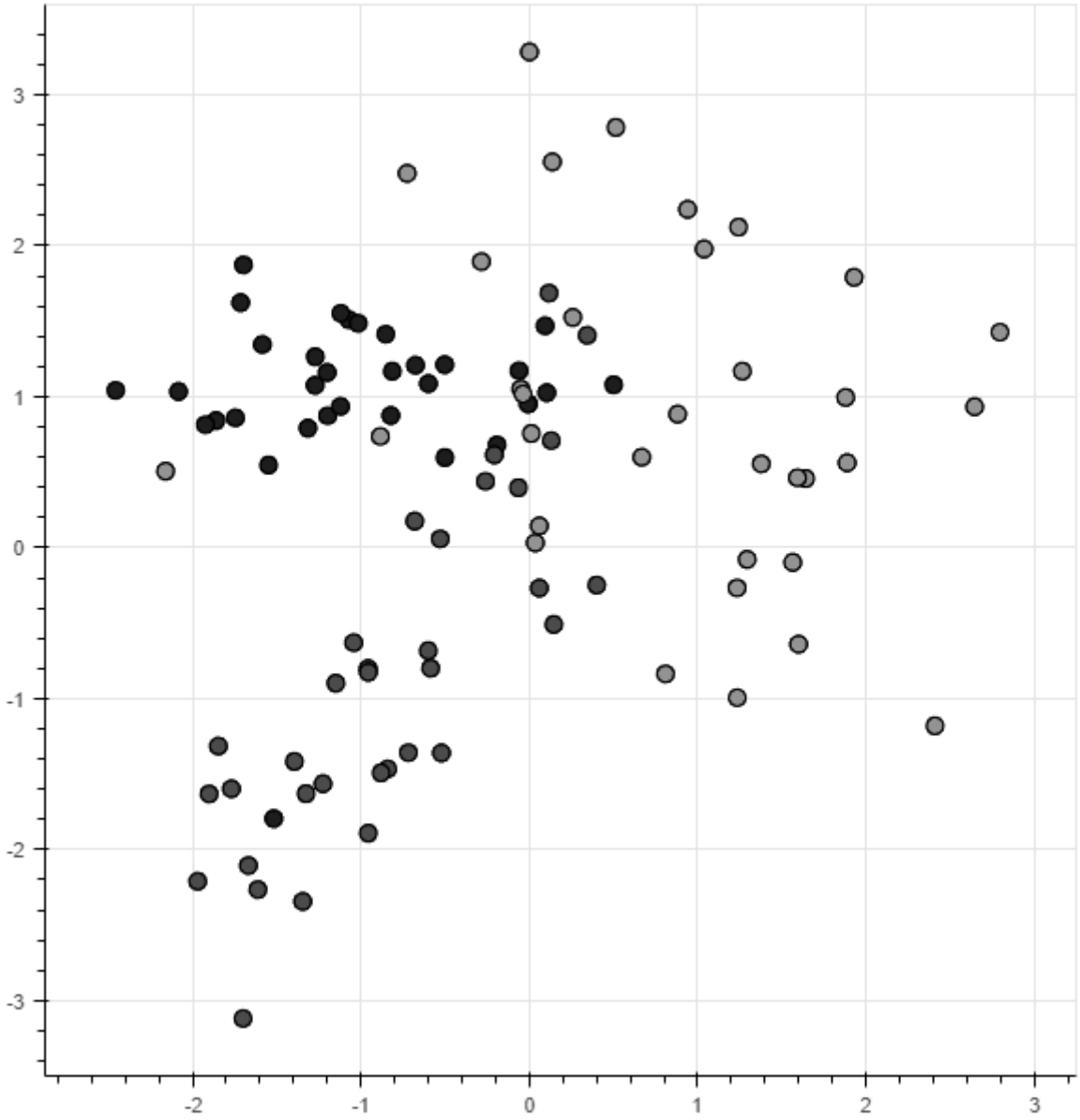
– Линейный дискриминант Фишера

В пакете **sklearn** данный классификатор описан в классе **LinearDiscriminantAnalysis**

При инициализации данного класса можно установить такие атрибуты как:

- метод решения (изначально установлен «Singular value decomposition »)
  - svd
  - lsqr – решениее наименьших квадратов.
  - Eigen – разложение по собственным значениям
- приоритеты классов

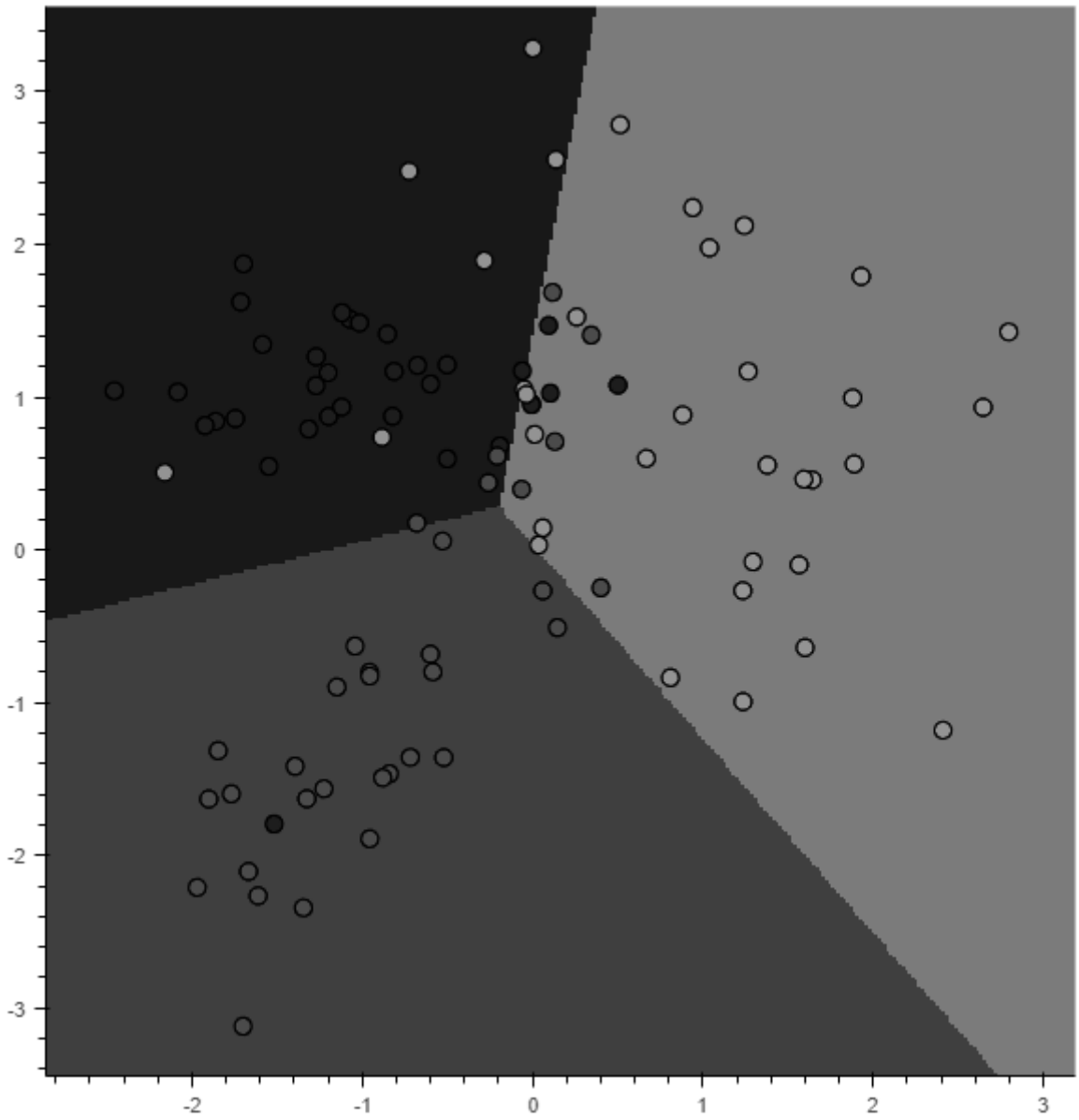
Рассмотрим работу данного алгоритма на выборке показанной ниже:



Так как нам не известны приоритеты классов, то инициализируем классификатор:

```
clf = LinearDiscriminantAnalysis()
clf.fit(X,Y)
```

В результате получаем такой результат:



Линейный метод классификации

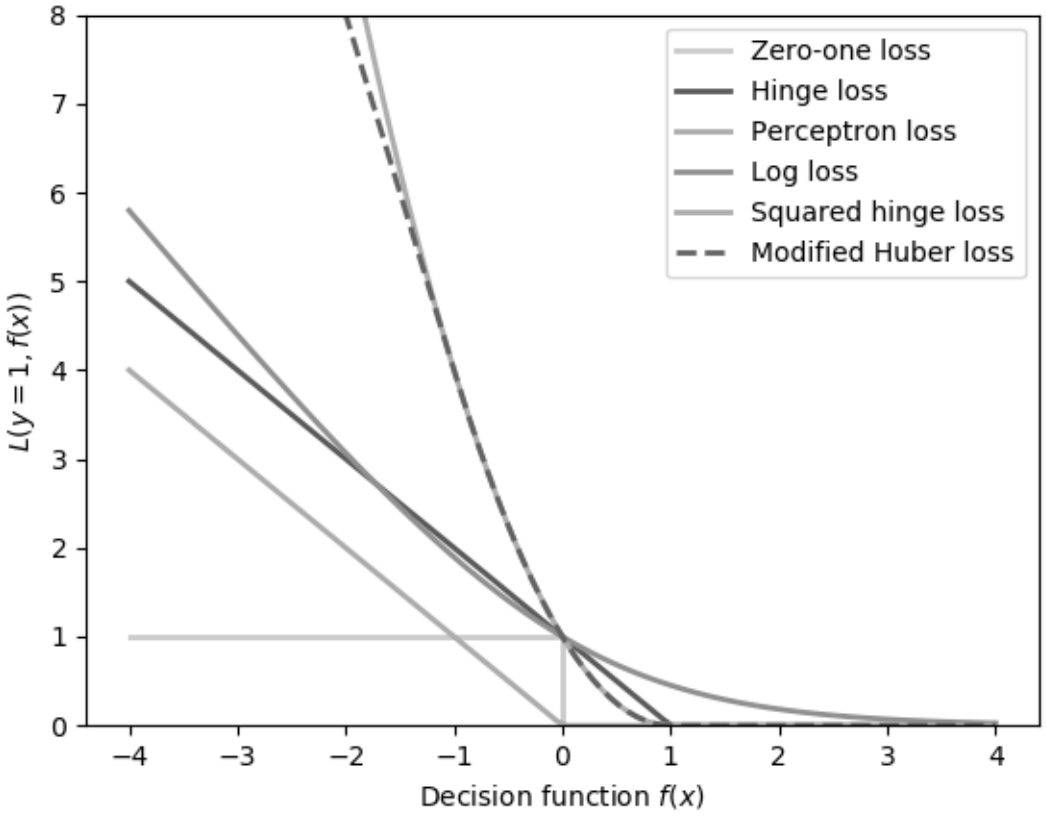
В данном методе задача сводиться на несколько подзадач по нахождению гиперплоскости оптимально разделяющей элементы класса i и элементы относящиеся ко всем остальным классам. Нахождение данной гиперплоскости организуется разными способами, но в данном случае будет рассмотрен классификатор использующий для этого метод градиентного спуска.

В пакете **sklearn** данный классификатор описан в классе **SGDClassifier**

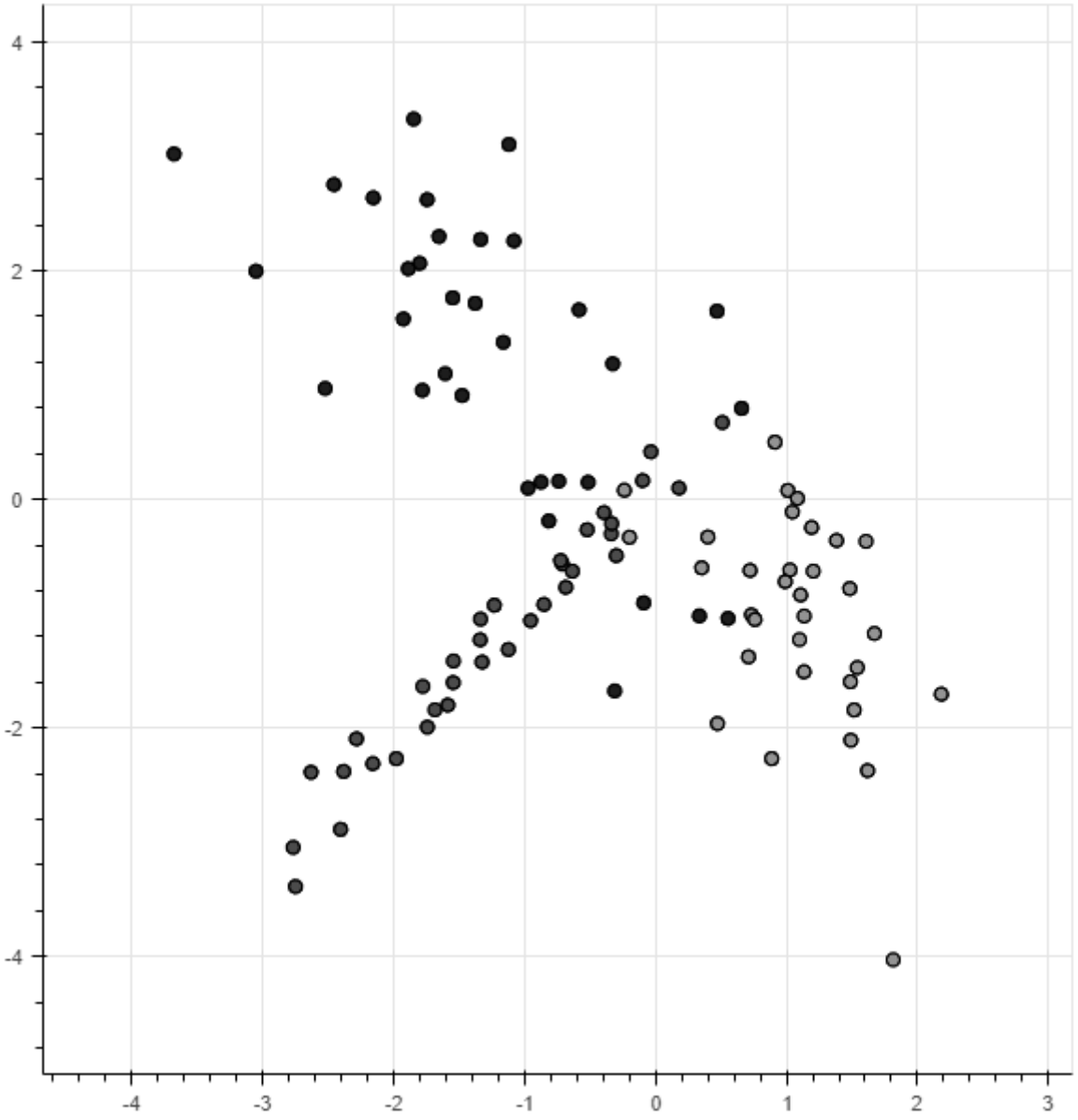
При инициализации можно выбрать такие параметры, как:

- функция потерь
- количество итераций градиентного спуска

В качестве функций потерь представлены функции с такими графиками:

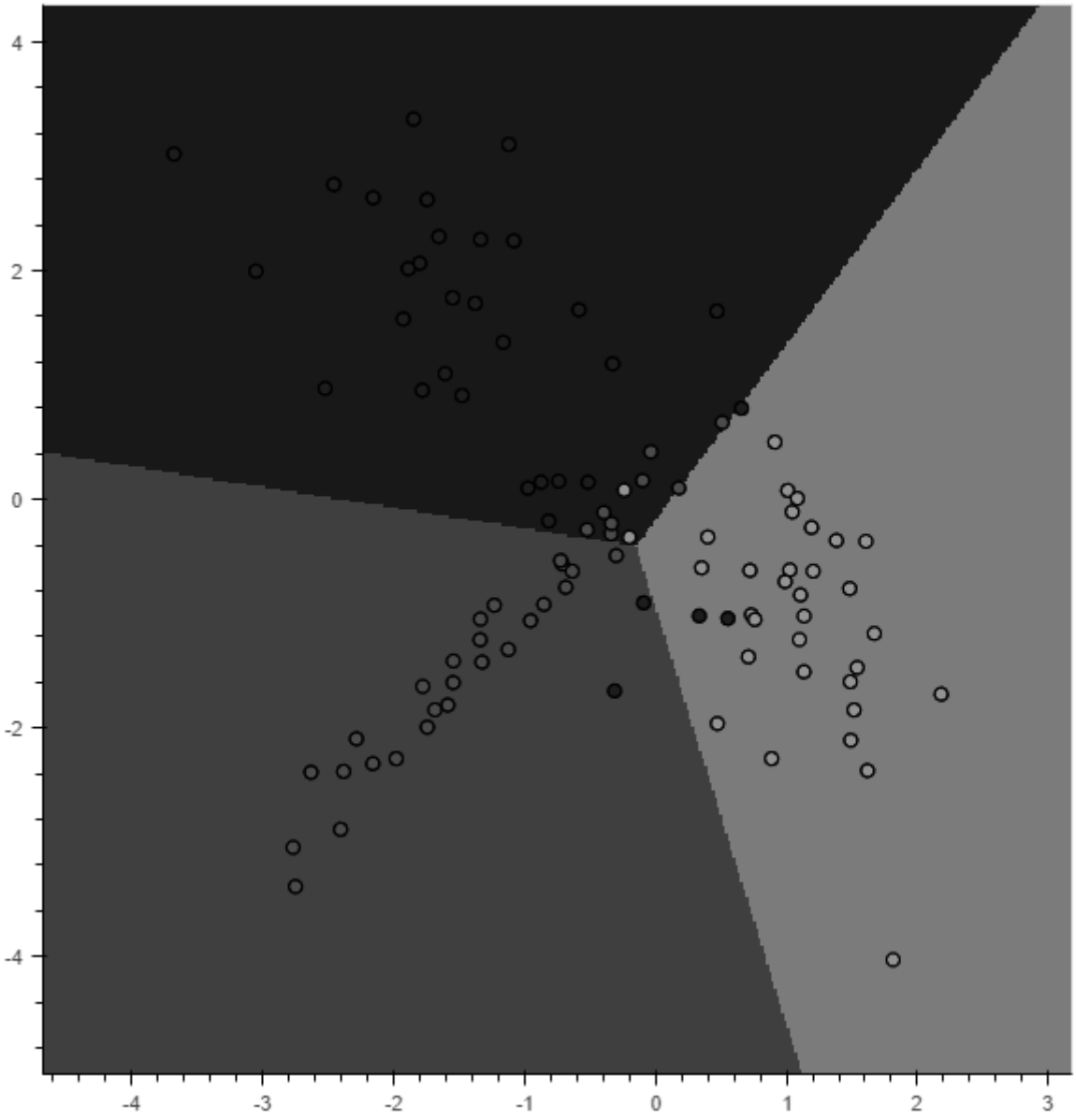


Введем выборку:



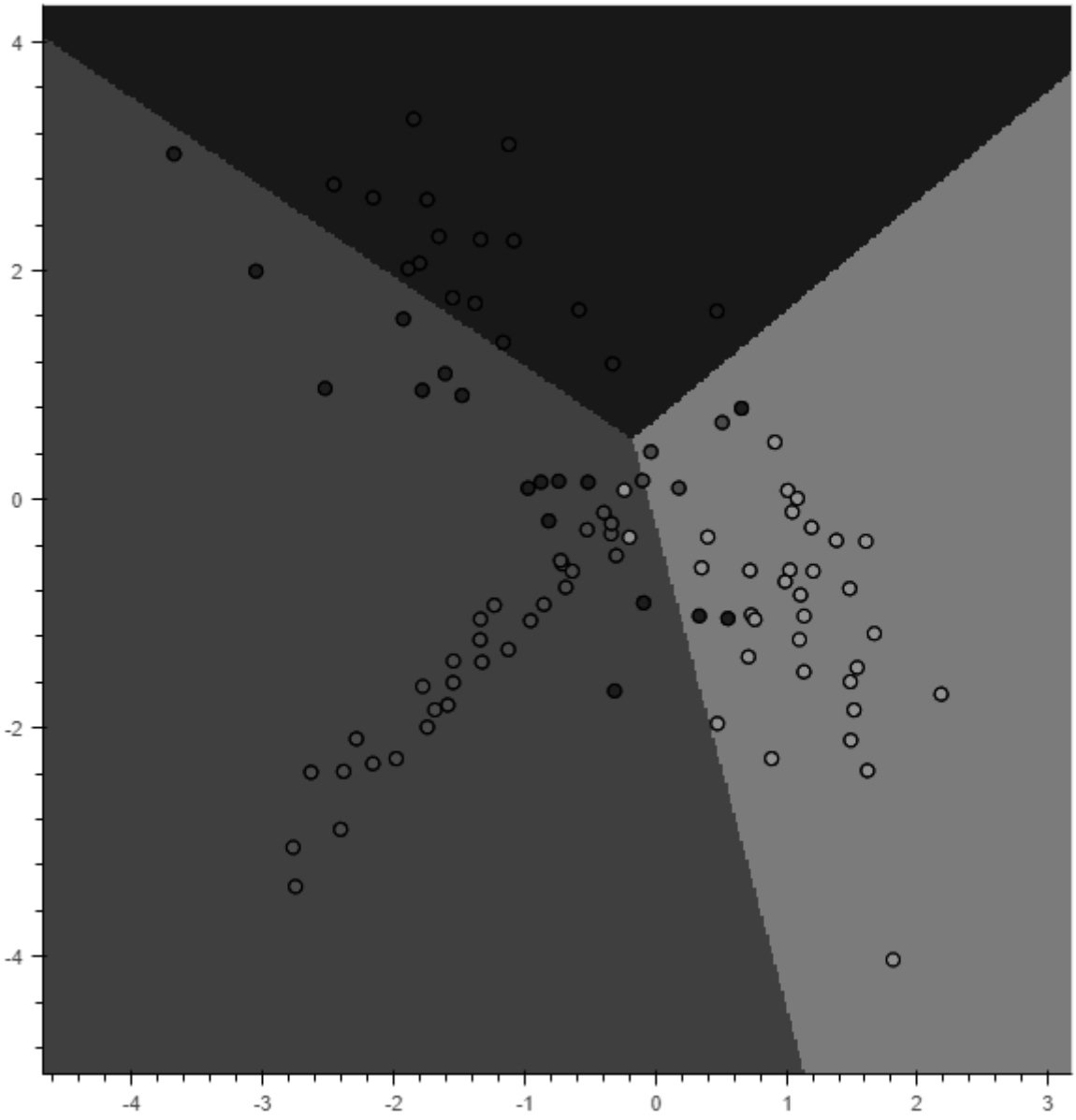
Рассмотрим работу алгоритма с функцией потерь Hinge:

```
clf = SGDClassifier(loss="hinge", penalty="l2", max_iter=300)
clf.fit(X,Y)
```



Рассмотрим работу алгоритма с функцией потерь Epsilon:

```
clf = SGDClassifier(loss="epsilon_insensitive", penalty="l2", max_iter=300)
clf.fit(X,Y)
```



Непараметрическая регрессия. Формула Надарая-Ватсона

Идея метода состоит в том, что результат алгоритма регрессии $a(x)$ вычисляется по ближайшим к объекту x объектам выборки X. К полученным ближайшим к x объектам применяется функция ядра, обозначающая «значение» данного объекта для оценки.

В данной работе будет рассмотрено две функции ядра.

1. Квартическое ядро

$$K(x)=\begin{cases}\frac{15}{16}(1-x^2)^2;|x|\le 1\\ 0;|x|>1\end{cases}$$

```
def kernel_cvart(val):
    return (15/16)*((1-val**2)**2)*(1 if math.fabs(val)<=1 else 0)
```

2. Гауссово ядро

$$K(x)=\sqrt{\Pi}e^{-\frac{x^2}{2}}$$

```
def kernel_gauss(val):
    return (math.pi**(-0.5))*(math.e**(-(val**2)/2))
```

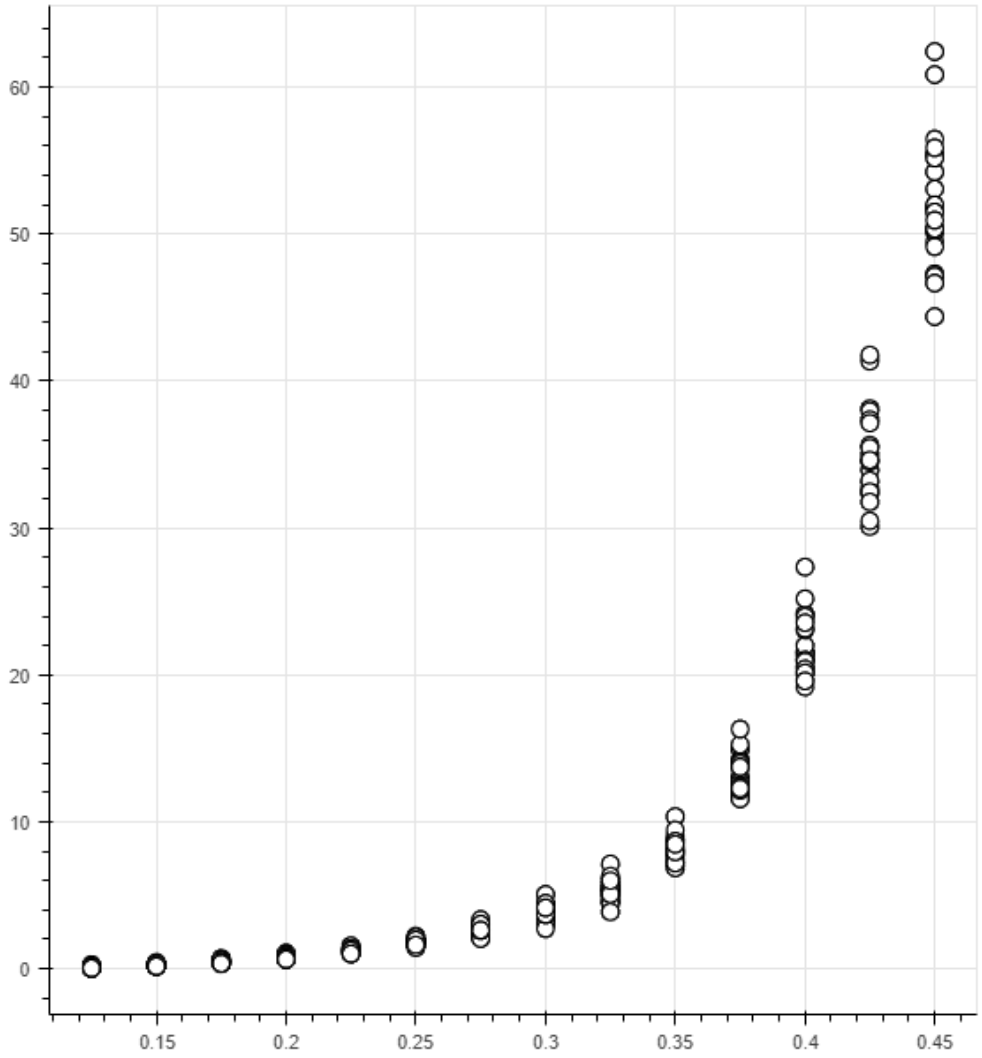
Выпишем формулу ядерного сглаживания Надарая-Ватсона, где h - ширина окна:

$$a_h(x;X^l)=\frac{\sum_{i=1}^{l} y_i K(\frac{\rho(x,x_i)}{h})}{\sum_{i=1}^{l} K(\frac{\rho(x,x_i)}{h})}$$

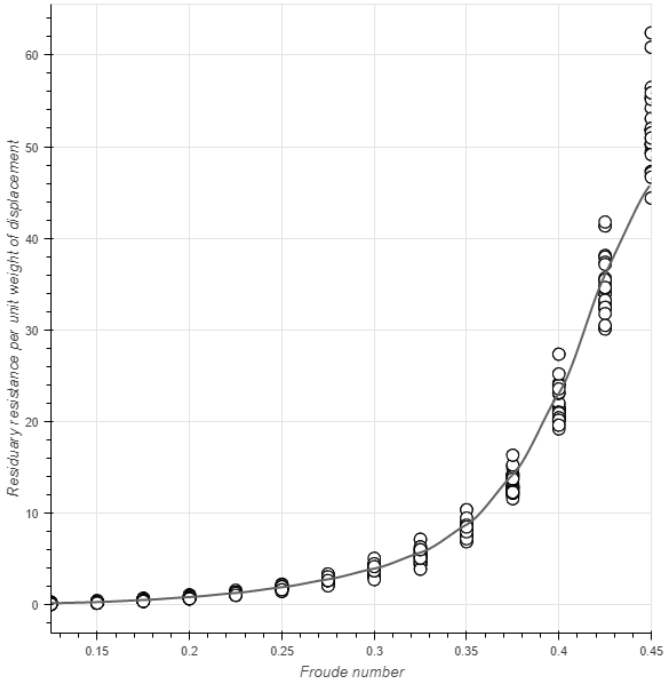
```
def nadarai_watson(x,X,Y,h,kernel):
    upper_sum = 0
    down_sum = 0
    for i in range(len(X)):
        upper_sum = upper_sum + Y[i]*kernel(metrics.pairwise.euclidean_distances([x],[X[i]])[0][0]/h)
        down_sum = down_sum + kernel(metrics.pairwise.euclidean_distances([x],[X[i]])[0][0]/h)
    return upper_sum/down_sum
```

Для данного алгоритма была выбрана выборка где в качестве параметров различные свойства кораблей. В качестве множества Y выберем значение водоизмещения корабля.

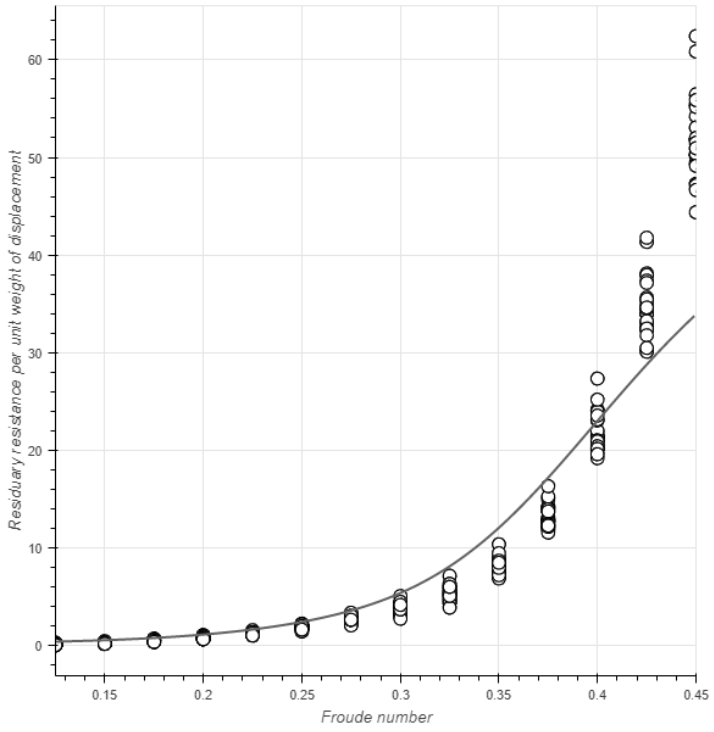
Считать будем относительно параметра “Froude number”



Результат при использовании квартического ядра:



Результат при использовании гауссова ядра:



Метод LOWESS для непараметрической регрессии

Алгоритм Lowess является улучшением формулы Надарая-Ватсона. Для каждого объекта выборки вводится вес δ_i , обозначающий значение этого объекта для алгоритма.

Выпишем формулу LOWESS, где h - ширина окна:

$$a_h(x;X^l)=\frac{\sum_{i=1}^{l}\delta_i y_i K(\frac{\rho(x,x_i)}{h})}{\sum_{i=1}^{l}\delta_i K(\frac{\rho(x,x_i)}{h})}$$

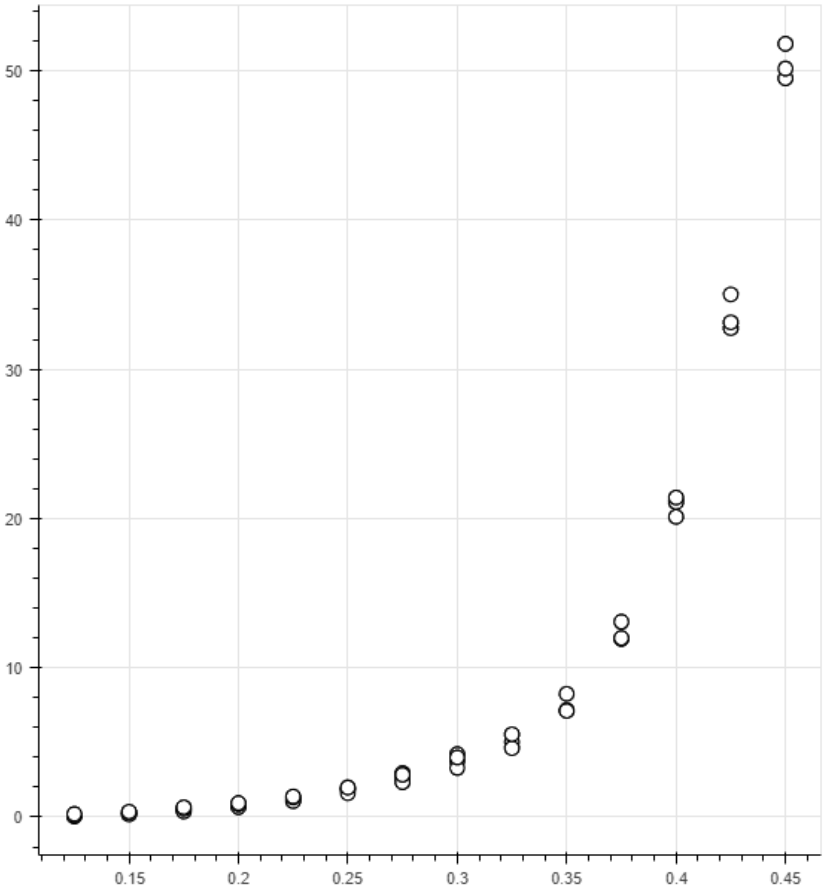
```
def lowess(x,X,Y,weights,kernel,h):
    if h == None:
        h = 1
    upper_sum = 0
    down_sum = 0
    for i in range(len(X)):
        upper_sum = upper_sum + Y[i]*weights[i]*kernel(metrics.pairwise.euclidean_distances([x],[X[i]])[0][0]/h)
        down_sum = down_sum + weights[i]*kernel(metrics.pairwise.euclidean_distances([x],[X[i]])[0][0]/h)
    if(upper_sum==0):
        return 0
    return upper_sum/down_sum
```

Для того, чтобы найти оптимальные значения весов w применяется алгоритм cross validation:

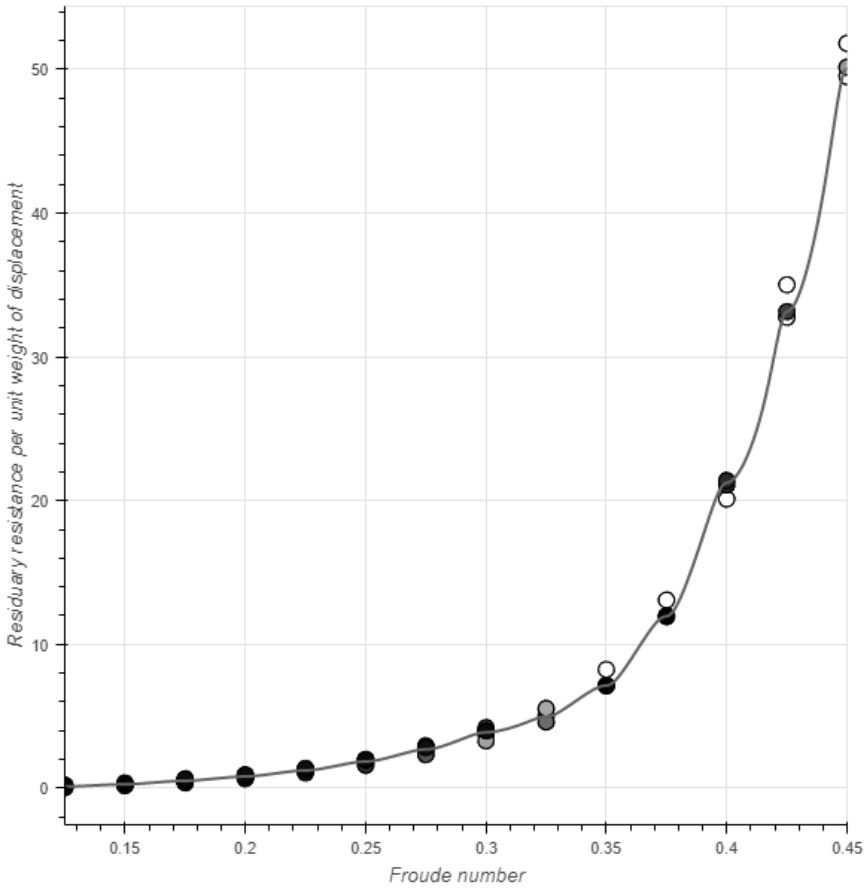
```
def cross_validation(X,Y,kernel,steps,h):
    weights = [1 for x in X]
    errors = [1 for x in X]
    for i in range(steps):
        for k in range(len(X)):
            item = X[k]
            val = Y[k]
            X_t = np.delete(X,k,0)
            Y_t = np.delete(Y,k)
            weights_t = np.delete(weights,k)
            val_t = lowess(item,X_t,Y_t,weights_t,kernel,h)
            errors[k] = abs(val_t-val)
        s = np.median(errors)
        weights = [(1-abs(e/(6*s))**2)**2 if abs((e/(6*s)))<=1 else 0 for e in errors]
    return weights
```

Для данного алгоритма была выбрана выборка, как и для Надарая-Ватсона где в качестве параметров различные свойства кораблей. В качестве множества Y выберем значение водоизмещения корабля.

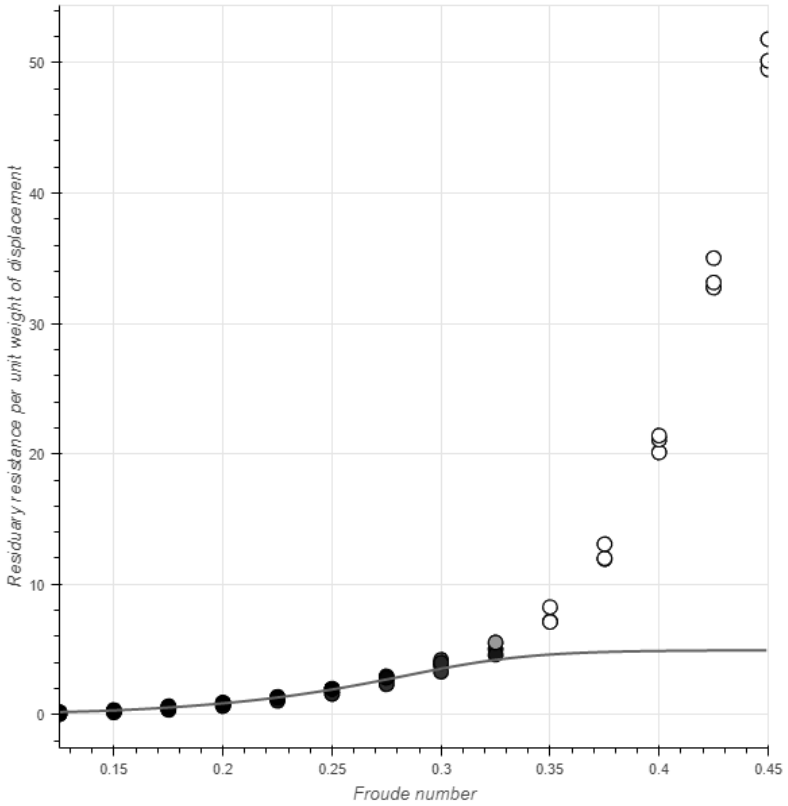
Считать будем относительно параметра “Froude number”



Результат при использовании квартического ядра:



Результат при использовании гауссова ядра:



Линейная регрессия

Требуется найти линию, как можно более аппроксимирующую результаты измерений.

Добавим к выборке X единичный столбец.

Тогда формула линии будет выглядеть так:

$$g(x;a)=\sum_{j=1}^{n} a_j f_j(x)$$

Вектор a будем находить из следующей системы:

$$X^t X a - X^t Y = 0$$

Нахождение вектора a реализовано функцией $linear(X, y)$

Также функцией $takeFromSvd(X, amnt)$ реализовано взятие двух основных влияющих параметров из сингулярного разложения.

```
def linear(X,y):
    a = mmult(X.T,X)
    b = mmult(X.T,y)
    return linalg.solve(a,b)

def takeFromSvd(X,amnt):
    V, D, U = linalg.svd(X,compute_uv=True, full_matrices=False)
    D = mIs(D,len(D))
    V = V[:,:amnt]
    D = D[:amnt,:amnt]
    U = U[:amnt,:amnt]
    nX = mmult(V,mmult(D,U))
    return nX
```

В качестве выборки было взята информация о пожарах в лесной зоне.

В качестве вектора Y были взяты значения площади возгорания.

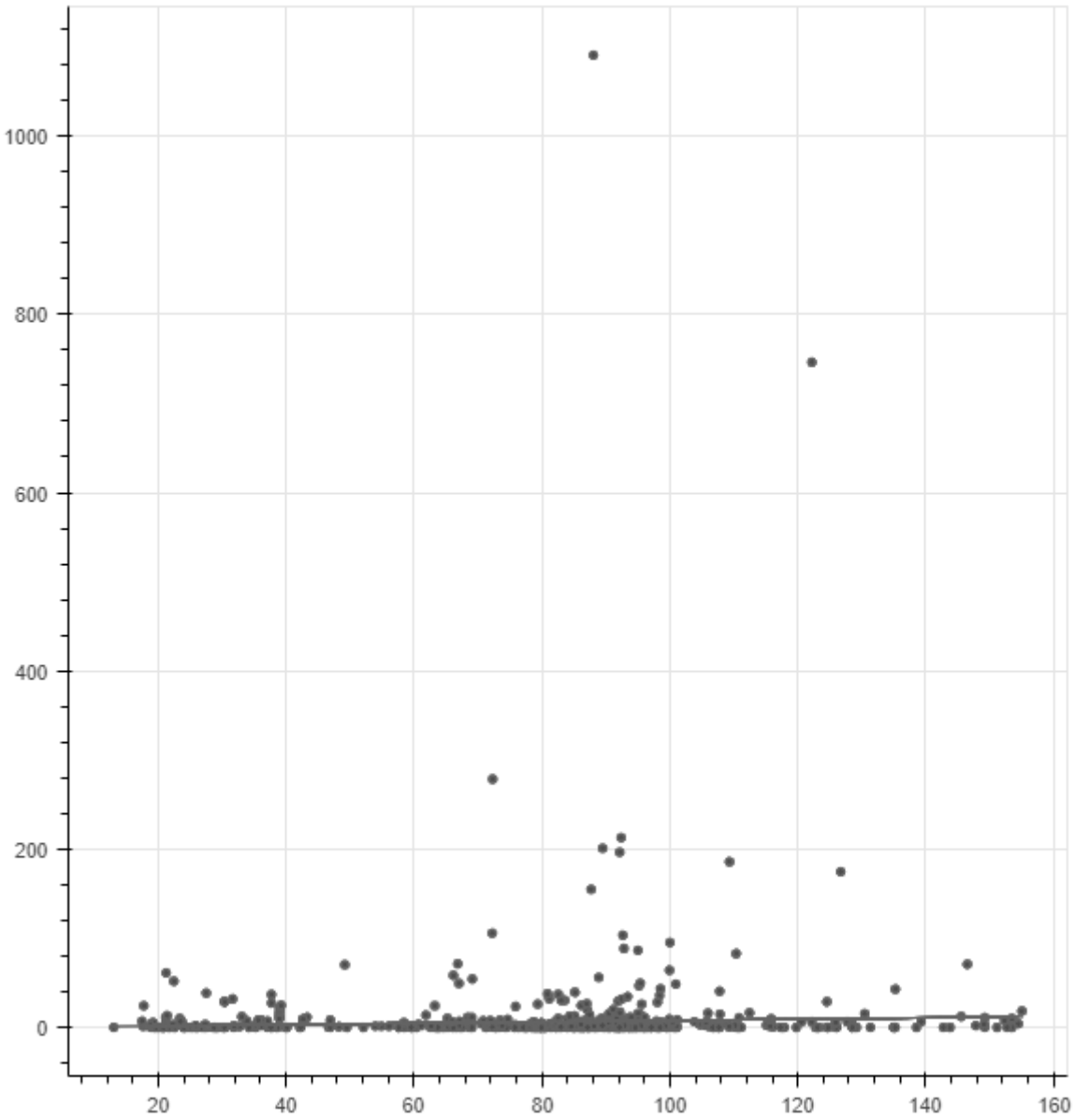
В качестве матрицы X были взяты значения погодных условий а также единичный вектор.

Результат работы для полной выборки:

$$y = -0.032\,x_1 + 0.0768\,x_2 - 0.0055\,x_3 - 0.696\,x_4 + 0.819\,x_5 - 0.207\,x_6 + 1.491\,x_7 + 4.277$$

Результат работы при выборе двух компонент

$$y = 0.076\,x_1 + 0.056$$



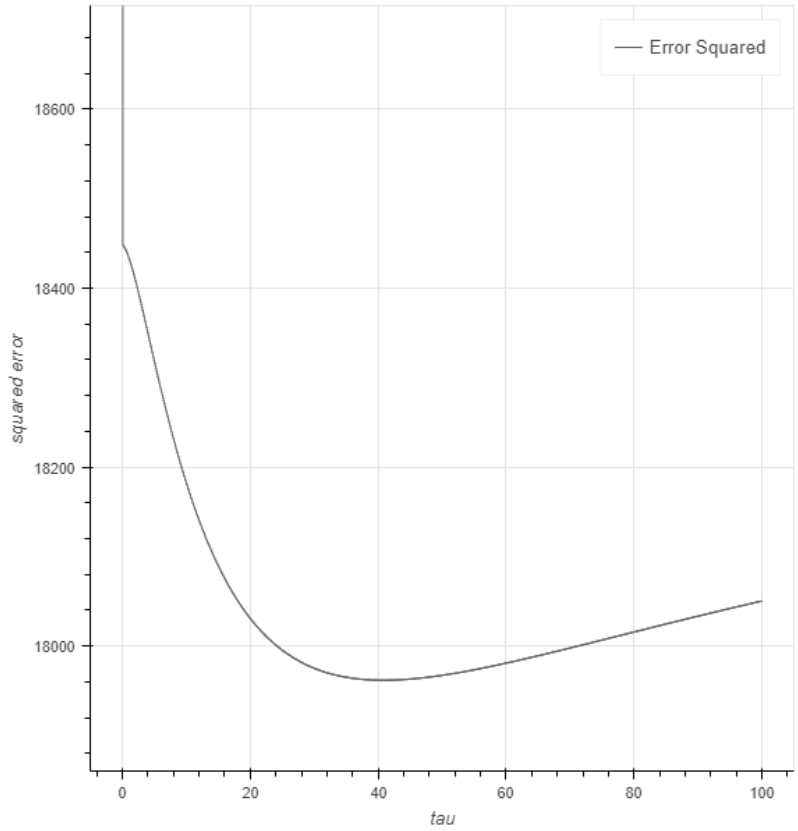
Гребневая регрессия

Гребневая регрессия по своей идее и почти по реализации не отличается от линейной. Отличием является, что для того чтобы понизить влияние мультиколлинеарности на результат вычислений все собственные значения при вычислении a увеличивают на некоторое неотрицательное τ (“Гребень”). В результате получая такую систему для вычисления a :

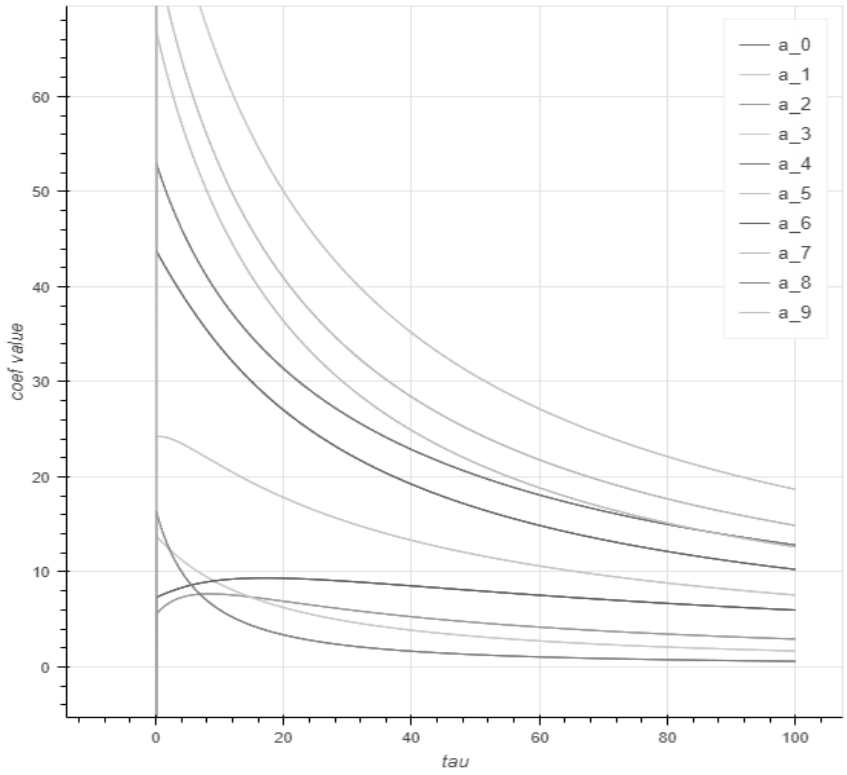
$$(X^t X+\tau E)a-X^t Y=0$$

```
def ridge(X,y,tau):
    a = mmult(X,X.T)
    a = msum(a,mI(tau,len(a)))
    a = minv(a)
    a = mmult(X.T,a)
    a = mmult(a,y)
    return a
```

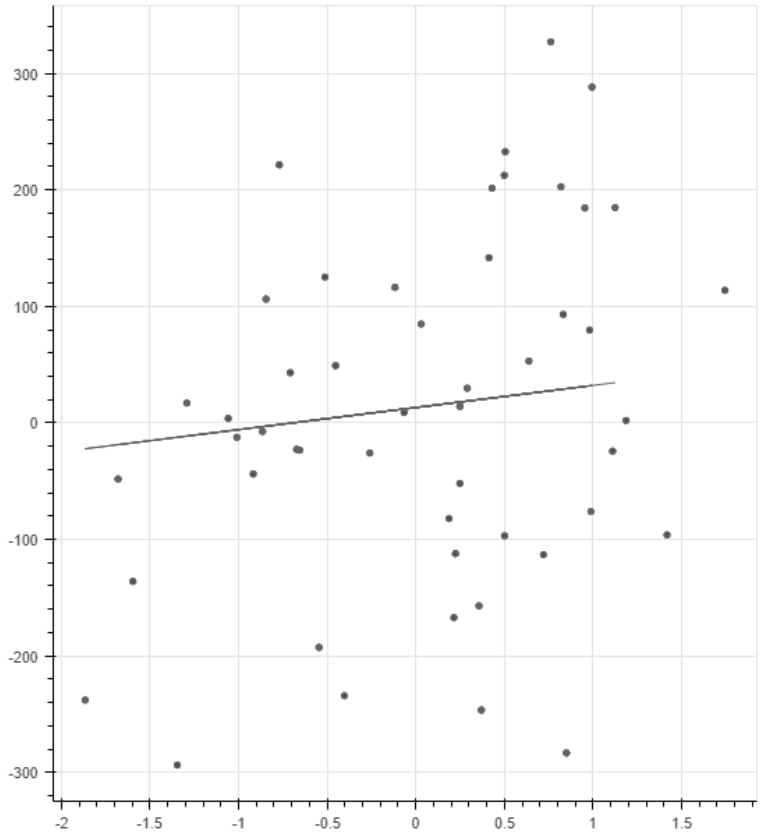
Далее любым способом ищется минимум квадратичной ошибки регрессии относительно τ



При этом, в зависимости от значений τ меняются и значения коэффициентов a .



Результат работы алгоритма:



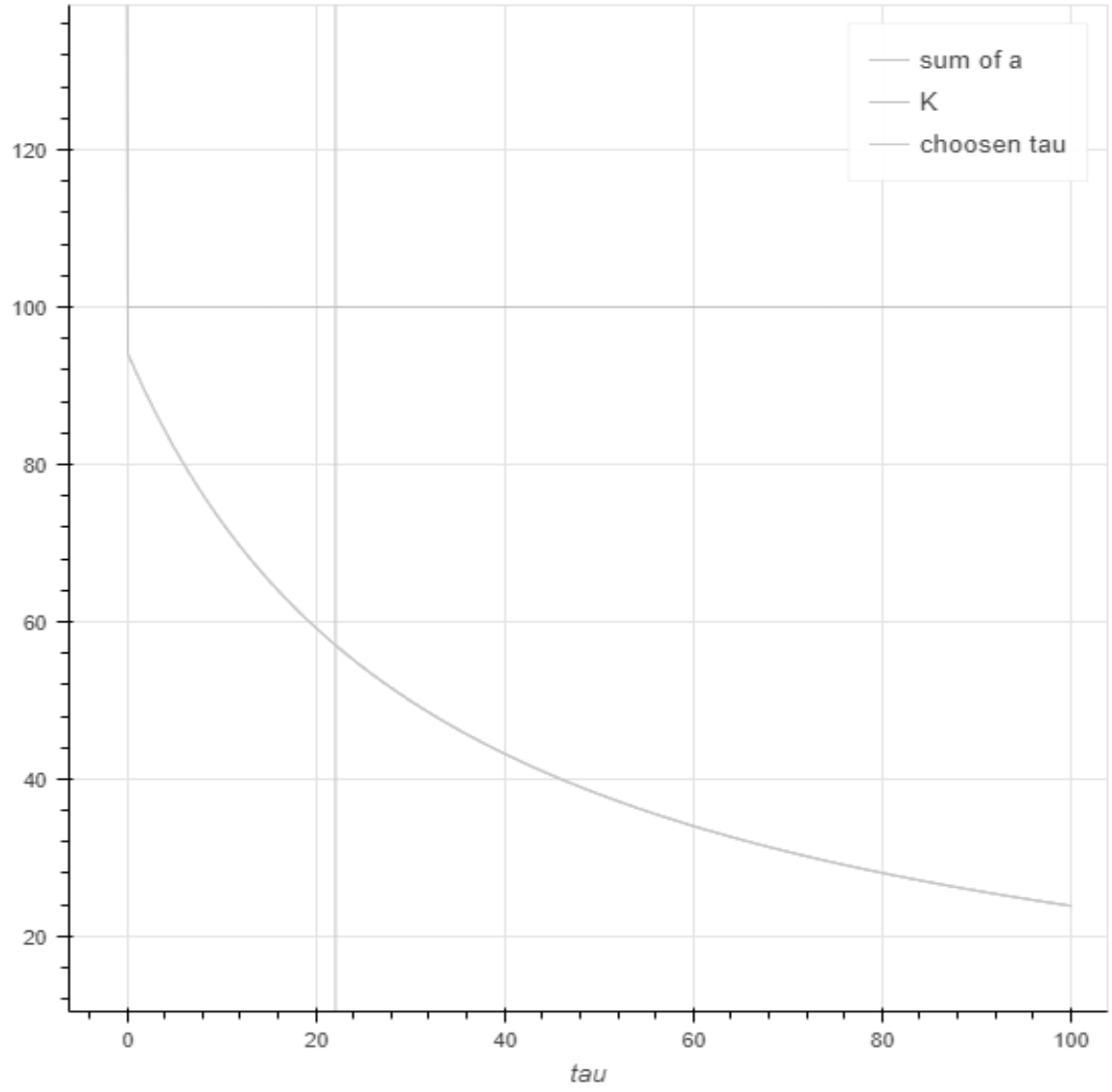
Метод Lasso

Ещё один способ избавления от мультиколлинеарности, который просто запрещает коэффициенты сумма которых больше определённого χ , при этом устремляя значение ошибки алгоритма к минимуму:

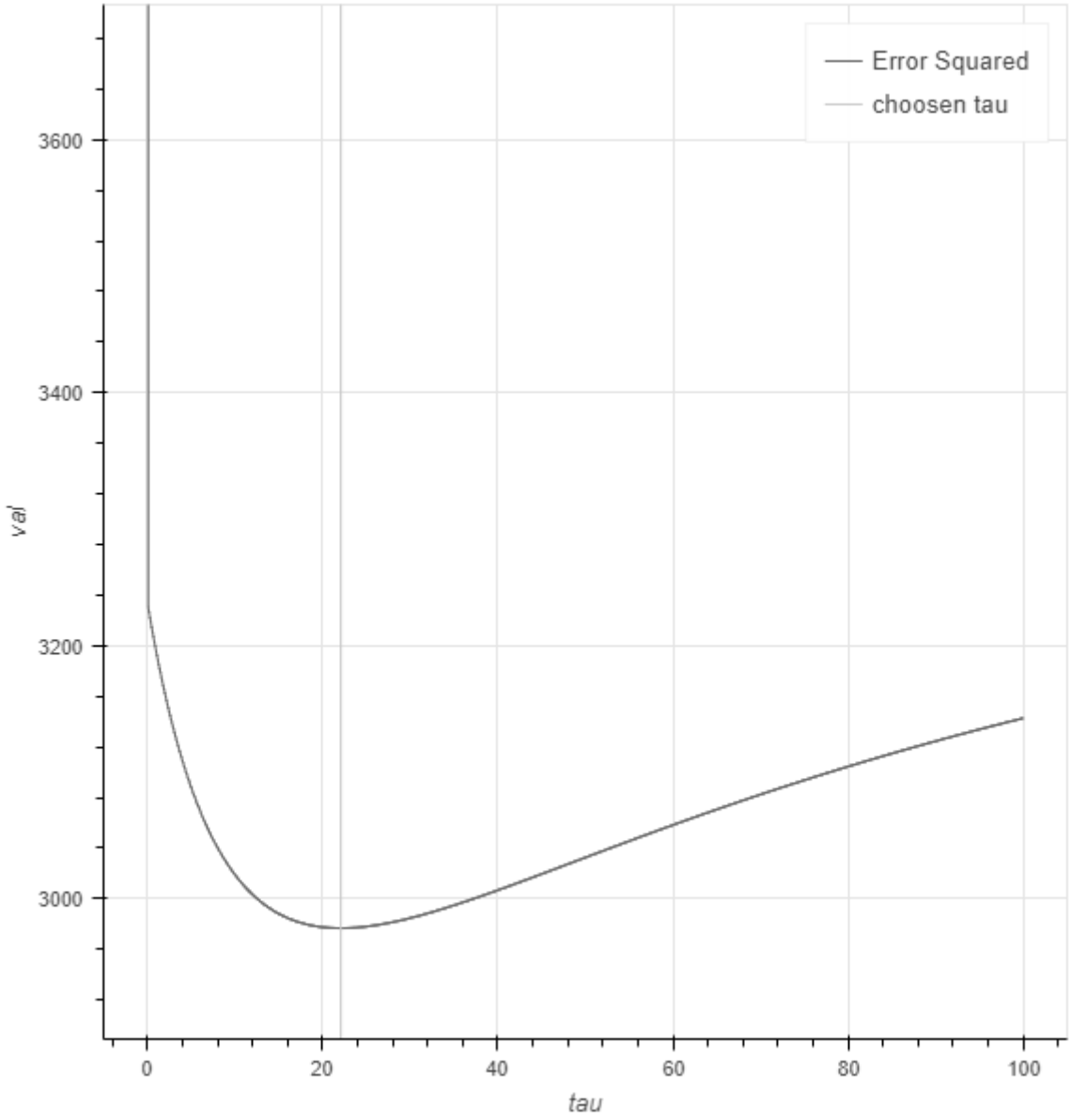
$$\begin{cases} Q(a) = \|Fa - y\|^2 \to \min\limits_{a} \\ \sum\limits_{j=1}^{n} |a_j| \le \chi \end{cases}$$

В данном примере, в качестве значения χ было взято число 100

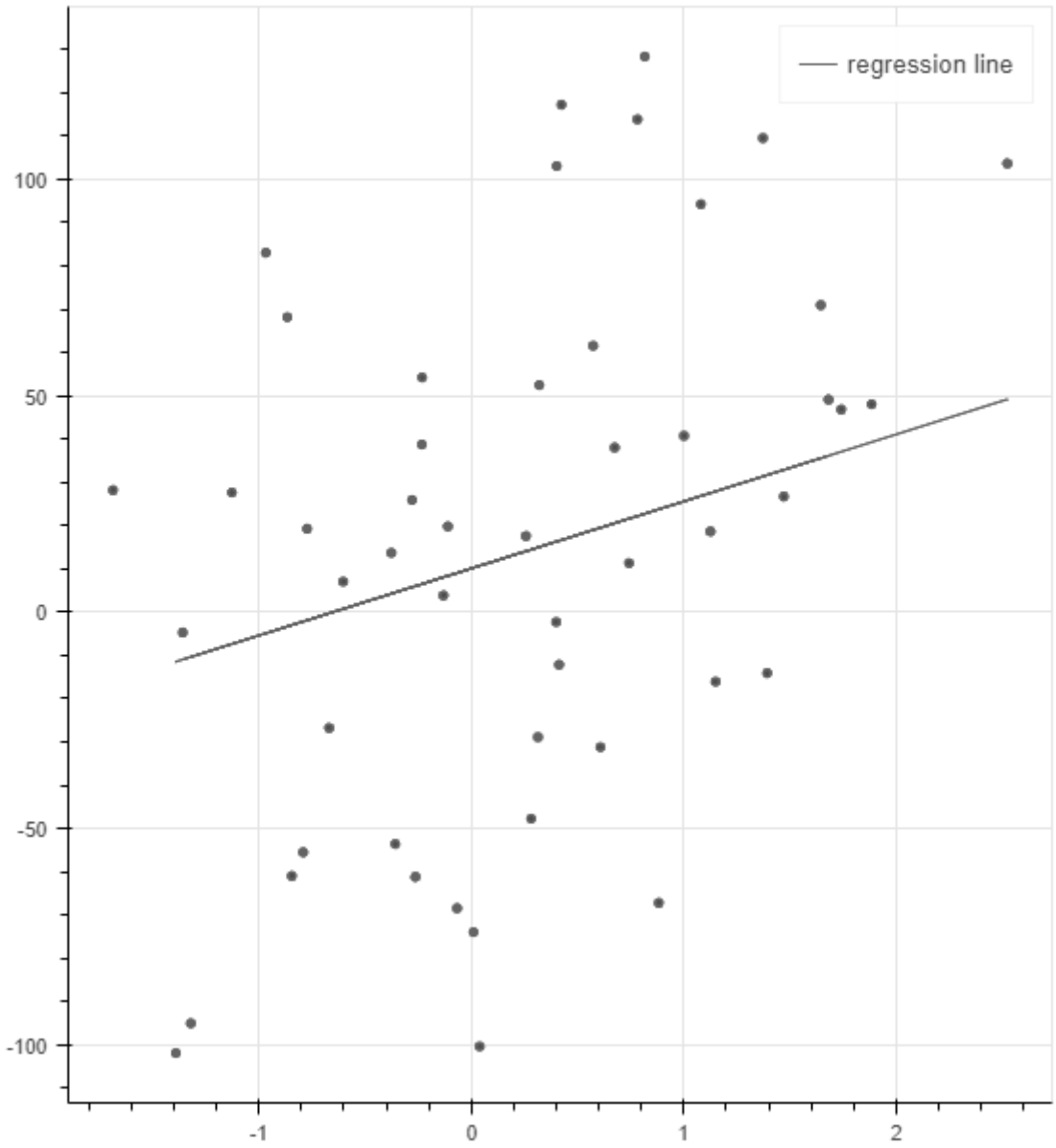
На графике видно, что было выбранное значение τ выбрано для значений суммы коэффициентов меньше значения χ



Далее были найдены коэффициенты дающие минимальную ошибку алгоритма:



И была построена регрессия:



Кластеризация WEB документов

Целью задачи было с помощью патентно-семантического анализа кластеризовать документы.

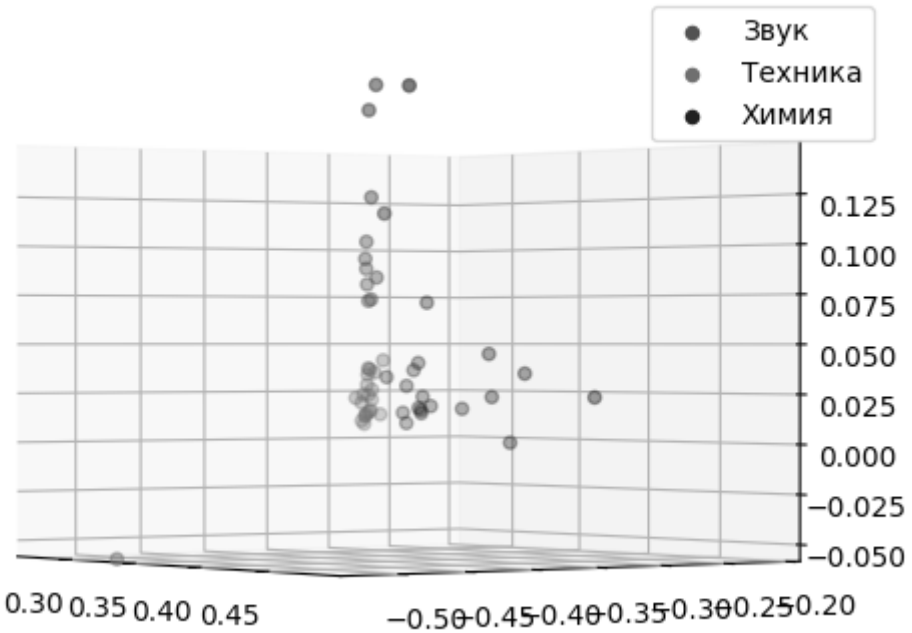
В качестве данных были выбраны по 20 документов из 3 тематик:

- Звук
- Техника
- Химия

В программе документы были обработаны и упорядочены.

Из каждого документа выбиралась частотная матрица русских слов, из которой в свою очередь удалялись стоп-слова, а также все слова подвергались обработке алгоритмом Стеммера Портера для русского языка.

Далее все частотные матрицы были объедены в одну и над ней было проведено сингулярное разложение. На основе полученных матриц было построено и визуализировано множество исследуемых документов.



Можно легко заметить, что алгоритм разделил документы на 3 группы, однако по причине некоторой схожести документов, а также частого упоминания в них названий стран, городов, некоторых имён и физических материалов, полученные кластеры довольно близки.

Далее с использованием кластеризатора **MiniBatchKMeans** поставляемого с пакетом **sklearn** была предпринята попытка разделить данное множество документов на 3 кластера, основываясь на их расположении в трёхмерном пространстве. Однако, как можно заметить на изображении ниже, кластеризатор, благодаря достаточно сильной близости кластеров, не смог правильно разделить объекты по классам.