**Отчет по дисциплине:**  Теория машинного обучения

**Язык программирования:** Python

**Тема №1:** Метрические алгоритмы классификации

**Ход работы программы:**

Используя готовые методы из библиотеки sklearn, применить алгоритмы классификации, основанные на **методе ближайших соседей** и **методе парзеновского окна** (использовать различные ядра) для классификации исходных данных.

Задано некое множество объектов **Х** и конечное множество классов **Y**, при чем, существует зависимость **.** Так же известно конечное множество для некоторых объектов **,** где **L** является мощность множества **.**

Пусть задана некоторая функция расстояния **,** где требуется построить алгоритм, как можно более точно аппроксимирующий функцию на всё множество **X.**

Для удобства работы над данными в качестве функции расстояния будет использоваться функция **,** а алгоритмы классификации импортированы из пакета sklearn.

Метрический алгоритм классификации относит объект **Х** к тому же классу **,** для которого суммарный вес обучающих объектов максимален и называется оценкой близости объекта **Х** к классу **Y.**

Где функция является коэффициентом важности соседа для объекта **.**

Теперь разберем подробнее **метод ближайших соседей.** Алгоритм классификации относит объект к классу, где элементов больше среди **-ого** ближайшего порядка.

Данные были описаны модельным образом.

X,Y = datasets.make\_classification(n\_samples=100, n\_features=2, n\_informative=2, n\_redundant=0,n\_clusters\_per\_class = 1, n\_repeated=0, n\_classes=3,shift =0.5,hypercube=False)

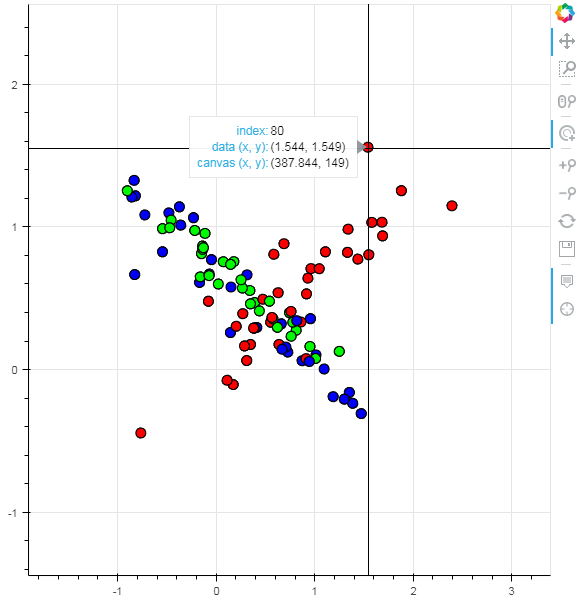
Классификатор описан в классе **KneighborsClassifier.** Что бы его найти следует зайти в пакет **sklean.** Данный класс при инициализации позволяет установить не только , но также метрику, алгоритм внутренней обработки данных и функцию весов .

n\_neighbors,cross\_validation\_result = leave\_one\_out(X,Y)

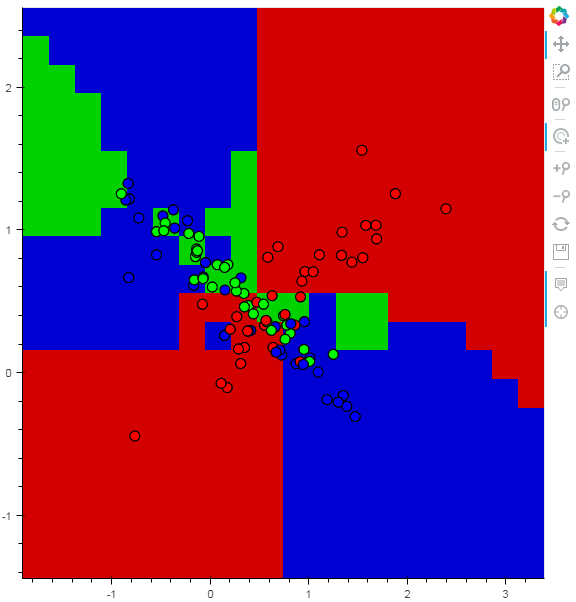
clf = neighbors.KNeighborsClassifier(n\_neighbors, weights='distance')

clf.fit(X, Y)

Для классификации используется выборка:



При метод будет очень чувствителен к выбросам. Для того что бы определить самое лучшее значение требуется использовать функцию **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):

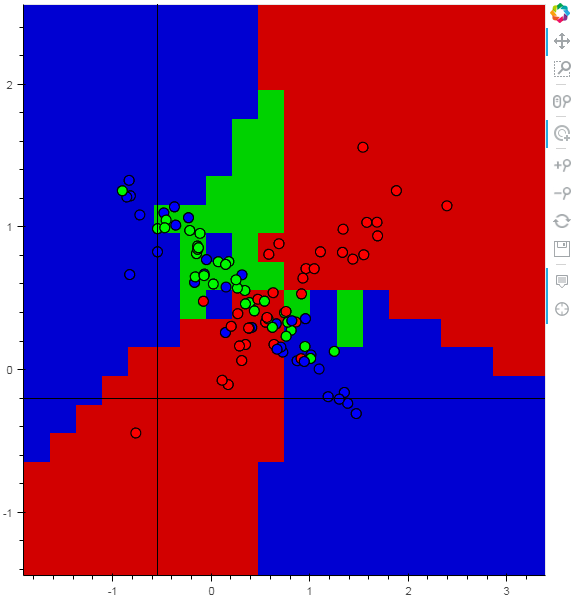
minimal\_errors = errors

minimal\_good\_neighbors = n\_neighbors

return minimal\_good\_neighbors,error\_percentage

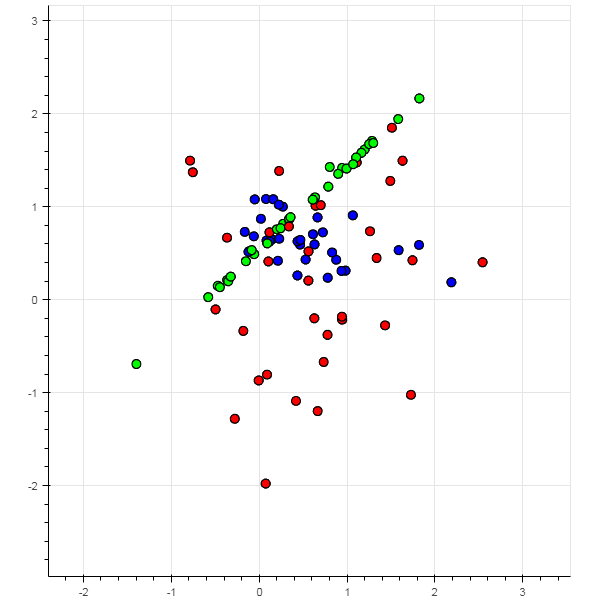
После проделанной работы функции можно увидеть процент погрешности классификации на выборке при каждом

Оптимальное значение, принимающее менее чувствительное к выбросам. После чего выборка преобразовалась к виду:



Для сравнения рассмотрим ещё один **метод окно Парзена**, который так же входит в метрические алгоритмы. Алгоритм классификации, относить объект к тому классу, элементов которого окажется больше среди его соседей находящихся на определенном расстоянии.

Используется выборка:



Классификатор описан в классе **RadiusNeighborsClassifier.** Что бы его найти следует зайти в пакет **sklean.** Данный класс при инициализации позволяет установить не только весов , как и в случае с методом ближайших соседей.

Для данного метода применены два ядра. Первое ядро расстояния :

def kernel\_distance(weights):

if(type(weights[0]) == type(1.1)):

res = np.ndarray(shape = (1))

res[0] = 0

return res

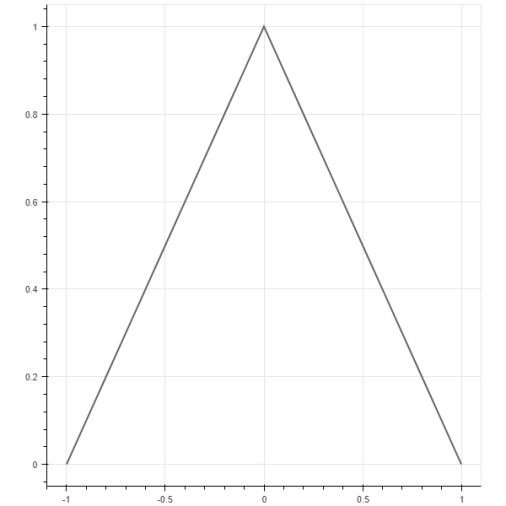
else:

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]])])



А так же второе ядро RBF:

def kernel\_rbf(weights):

if(type(weights[0]) == type(1.1)):

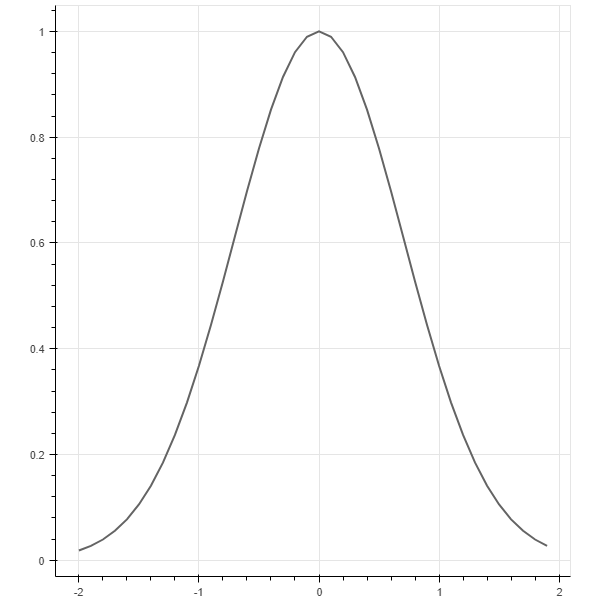
res = np.ndarray(shape = (1))

res[0] = 0

return res

else:

return np.asarray([np.asarray([metrics.pairwise.rbf\_kernel([[x]],[[0]])[0][0] for x in weights[0]])])



Сравним эти два ядра. Алгоритм действий представлен в коде на языке Python:

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)

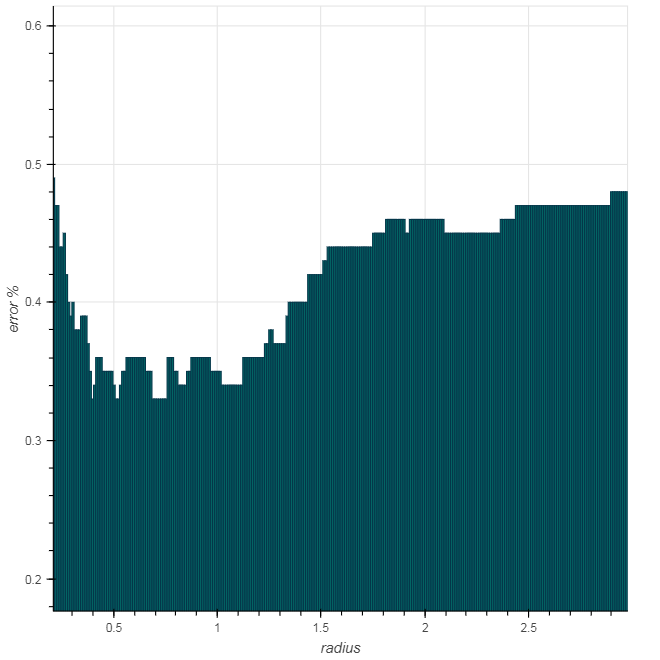
if(errors<minimal\_errors):

minimal\_errors = errors

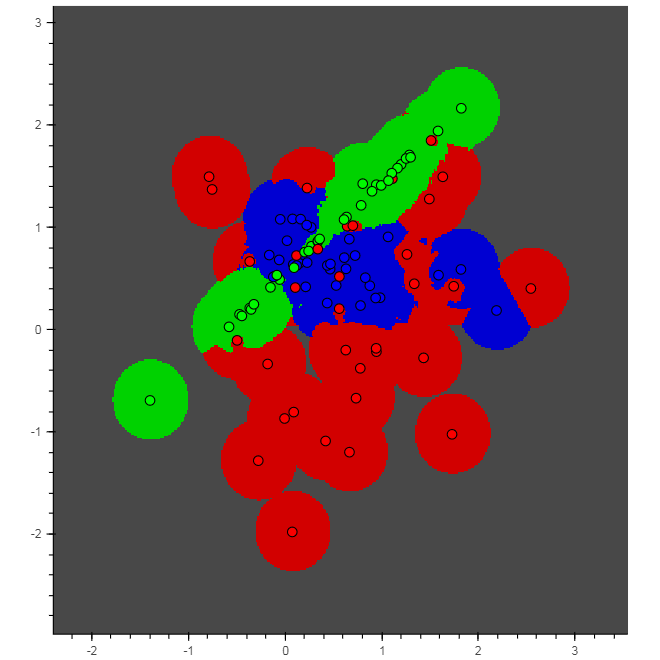
minimal\_good\_radius = radius

return minimal\_good\_radius,error\_percentage

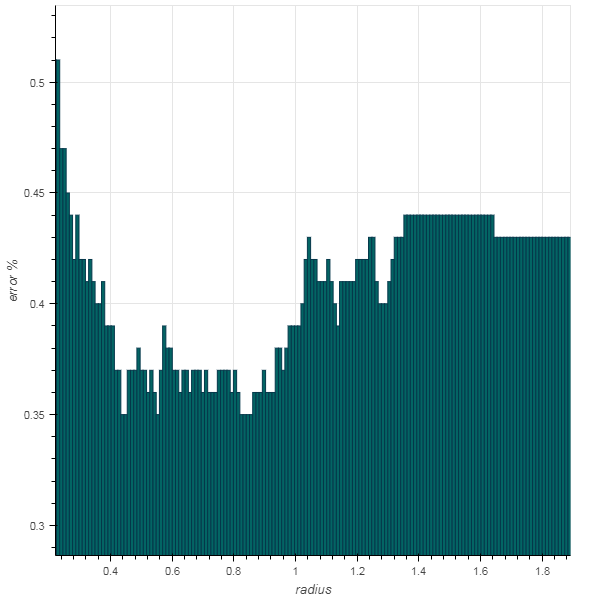
Так же для выбора оптимальной ширины окна используется алгоритм LOO для окна Парзена.

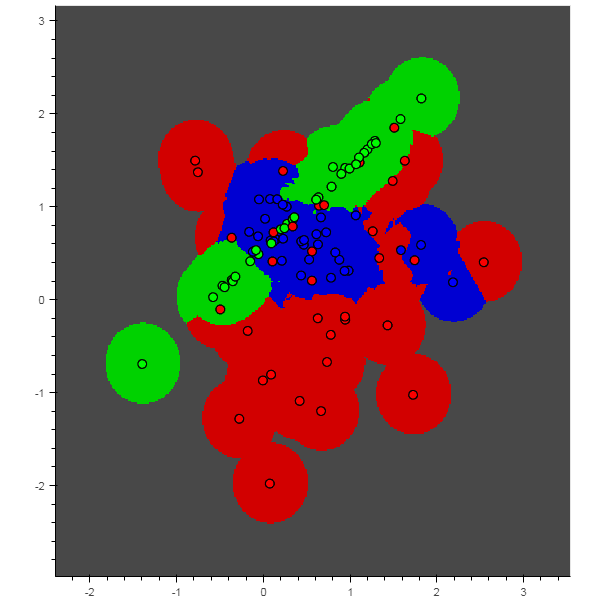


Получен следующий график с процентом ошибки относительно ширины окна:



Для ядра RBF график процента ошибки относительно ширины окна следующий:





Следовательно можно твердить что разница не большая, но ядро RBF менее погрешна к ошибкам.