|  |  |
| --- | --- |
| Изображение выглядит как эмблема, герб, нашивка, символ  Автоматически созданное описание | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное автономное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

|  |  |
| --- | --- |
| ФАКУЛЬТЕТ | ГОЛОВНОЙ УЧЕБНО-ИССЛЕДОВАТЕЛЬСКИЙ И МЕТОДИЧЕСКИЙ ЦЕНТР |
| ПРОФЕССИОНАЛЬНОЙ РЕАБИЛИТАЦИИ ЛИЦ С ОГРАНИЧЕННЫМИ |
| ВОЗМОЖНОСТЯМИ ЗДОРОВЬЯ |
| КАФЕДРА | СИСТЕМЫ ОБРАБОТКИ ИНФОРМАЦИИ И УПРАВЛЕНИЯ |

Отчёт по лабораторной работе №3 по курсу «Технологии машинного обучения».

«Подготовка обучающей и тестовой выборки, кросс-валидация и подбор гиперпараметров на примере метода ближайших соседей».

Выполнил: Проверил:

Новиков С. А. Гапанюк Ю.Е.

студент группы ИУ5-62Б

Подпись и дата: Подпись и дата:

*2025 г.*

1. **Задание лабораторной работы**

* Выбрать набор данных (датасет) для решения задачи классификации или регрессии.
* С использованием метода train\_test\_split разделить выборку на обучающую и тестовую.
* Обучить модель ближайших соседей для произвольно заданного гиперпараметра K. Оце- нить качество модели с помощью подходящих для задачи метрик.
* Произвести подбор гиперпараметра K с использованием GridSearchCV и/или RandomizedSearchCV и кросс-валидации, оценить качество оптимальной модели. Желательно использование нескольких стратегий кросс-валидации.
* Сравнить метрики качества исходной и оптимальной моделей.

1. **Ячейки Jupyter-ноутбука**

# Выбор и загрузка данных

В качестве датасета будем использовать набор данных, содержащий данные о различных стёклах. Данный набор доступен по адресу: [https://www.kaggle.com/datasets/uciml/glass](http://www.kaggle.com/datasets/uciml/glass)

Набор данных имеет следующие атрибуты:

* RI - Refractive Index - коэффициент преломления
* Na - Sodium - Содержание натрия (массовый процент в соответствующем оксиде)
* Mg - Magnesium - Содержание магния
* Al - Aluminum - Содержание алюминия
* Si - Silicon - Содержание кремния
* K - Potassium - Содержание калия
* Ca - Calcium - Содержание кальция
* Ba - Barium - Содеражние бария
* Fe - Iron - Содержание железа
* Type - Type of glass - тип стекла (1, 2 - стекла для зданий, 3, 4 - стекла для автомобилей, 5

- стеклотара, 6 - tableware - бытовые стекла, 7 - стекла для ламп; 4 отсутствует в данном наборе данных)

## Импорт библиотек

Импортируем библиотеки с помощью команды import:

[1]:

**import numpy as np import pandas as pd import seaborn as sns**

**import matplotlib.pyplot as plt**

%**matplotlib** inline sns.set(style=DticksD)

[2]:

## Загрузка данных

Загрузим набор данных:

data = pd.read\_csv('glass.csv')

[3]:

# 

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [3]: |  | RI | Na | Mg | Al | Si | K | Ca | Ba | Fe | Type |
|  | 0 | 1.52101 | 13.64 | 4.49 | 1.10 | 71.78 | 0.06 | 8.75 | 0.0 | 0.0 | 1 |
|  | 1 | 1.51761 | 13.89 | 3.60 | 1.36 | 72.73 | 0.48 | 7.83 | 0.0 | 0.0 | 1 |
|  | 2 | 1.51618 | 13.53 | 3.55 | 1.54 | 72.99 | 0.39 | 7.78 | 0.0 | 0.0 | 1 |
|  | 3 | 1.51766 | 13.21 | 3.69 | 1.29 | 72.61 | 0.57 | 8.22 | 0.0 | 0.0 | 1 |
|  | 4 | 1.51742 | 13.27 | 3.62 | 1.24 | 73.08 | 0.55 | 8.07 | 0.0 | 0.0 | 1 |

Определим размер датасета и целевого признака (Type):

[4]:

data.shape, data.Type.shape

1. : ((214, 10), (214,))
2. :

## Разделение данных

Разделим данные на столбец с целевым признаком и данные с другими столбцами:

X = data.drop(DTypeD, axis=1) y = data[DTypeD]

1. :

print(X.head(), D**\n**D) print(y.head())

RI Na Mg Al Si K Ca Ba Fe

0 1.52101 13.64 4.49 1.10 71.78 0.06 8.75 0.0 0.0

1 1.51761 13.89 3.60 1.36 72.73 0.48 7.83 0.0 0.0

2 1.51618 13.53 3.55 1.54 72.99 0.39 7.78 0.0 0.0

3 1.51766 13.21 3.69 1.29 72.61 0.57 8.22 0.0 0.0

4 1.51742 13.27 3.62 1.24 73.08 0.55 8.07 0.0 0.0

|  |  |
| --- | --- |
| 0 | 1 |
| 1 | 1 |
| 2 | 1 |
| 3 | 1 |
| 4 | 1 |

Name: Type, dtype: int64

1. :

print(X.shape) print(y.shape)

(214, 9)

(214,)

1. :

## Разделение выборки на обучающую и тестовую

Будем решать задачу классификации - отношения записи к определенному типу стекла. Разделим выборку с помощью функции train\_test\_split:

**from sklearn.model\_selection import** train\_test\_split

1. :

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, random\_state=1)

1. :
2. : ((160, 9), (160,))
3. :

X\_test.shape, y\_test.shape

1. : ((54, 9), (54,))

В выборках остались все типы стекол, доступные в изначальном датасете (4 отсутствует в исходном):

1. :

np.unique(y\_train)

1. : array([1, 2, 3, 5, 6, 7], dtype=int64)
2. :

np.unique(y\_test)

1. : array([1, 2, 3, 5, 6, 7], dtype=int64)

Проверим распределение типов стекол:

1. :

**from typing import** Dict, Tuple

1. :

**def** type\_proportions(array: np.ndarray) -> Dict[int, Tuple[int, float]]: labels, counts = np.unique(array, return\_counts=**True**)

counts\_perc = counts/array.size res = dict()

**for** label, count2 **in** zip(labels, zip(counts, counts\_perc)): res[label] = count2

**return** res

**def** print\_type\_proportions(array: np.ndarray): proportions = type\_proportions(array)

**if** len(proportions)>0:

print('Тип **\t** Количество **\t** Процент встречаемости')

**for** i **in** proportions:

val, val\_perc = proportions[i] val\_perc\_100 = round(val\_perc \* 100, 2)

print('**{} \t {} \t \t {}**%'.format(i, val, val\_perc\_100))

1. :

print\_type\_proportions(data.Type)

|  |  |  |
| --- | --- | --- |
| Тип | Количество | Процент встречаемости |
| 1 | 70 | 32.71% |
| 2 | 76 | 35.51% |
| 3 | 17 | 7.94% |
| 5 | 13 | 6.07% |
| 6 | 9 | 4.21% |
| 7 | 29 | 13.55% |

1. :

print\_type\_proportions(y\_train)

|  |  |  |
| --- | --- | --- |
| Тип | Количество | Процент встречаемости |
| 1 | 47 | 29.38% |
| 2 | 60 | 37.5% |
| 3 | 10 | 6.25% |

|  |  |  |
| --- | --- | --- |
| 5 | 12 | 7.5% |
| 6 | 8 | 5.0% |
| 7 | 23 | 14.37% |

1. :

print\_type\_proportions(y\_test)

|  |  |  |
| --- | --- | --- |
| Тип | Количество | Процент встречаемости |
| 1 | 23 | 42.59% |
| 2 | 16 | 29.63% |
| 3 | 7 | 12.96% |
| 5 | 1 | 1.85% |
| 6 | 1 | 1.85% |
| 7 | 6 | 11.11% |

Видим, что пропорции типов стекол приблизительно сохранились.

# Построение модели ближайших соседей для произвольного гиперпараметра

Пусть гиперпараметр будет равен 20, построим модель:

1. :

**from sklearn.neighbors import** KNeighborsClassifier

1. :

clf\_i = KNeighborsClassifier(n\_neighbors=20) clf\_i.fit(X\_train, y\_train)

target\_i = clf\_i.predict(X\_test) len(target\_i), target\_i

[20]: (54,

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| array([5, | 7, | 2, | 2, | 1, | 1, | 2, | 1, | 2, | 1, 2, 1, 2, 1, 1, 2, | 1, 1, 1, 1, | 1, 7, |
| 2, | 1, | 2, | 1, | 2, | 2, | 2, | 1, | 5, | 7, 1, 1, 1, 1, 2, 2, | 5, 1, 1, 7, | 7, 1, |
| 7, | 1, | 2, | 1, | 2, | 7, | 1, | 2, | 1, | 2], dtype=int64)) |  |  |

Эту модель будем считать исходной.

# Кросс-валидация и подбор гиперпараметра K через GridSearch и RandomizedSearch

## Кросс-валидация

1. :

**from sklearn.model\_selection import** cross\_val\_score, cross\_validate

1. :

scores = cross\_val\_score(KNeighborsClassifier(n\_neighbors=2),

data, data.Type, cv=3)

scores

1. : array([0.91666667, 0.95774648, 0.95774648])

С помощью функции cross\_validate:

1. :

scoring = {'precision': 'precision\_weighted', 'recall': 'recall\_weighted',

'f1': 'f1\_weighted'}

scores = cross\_validate(KNeighborsClassifier(n\_neighbors=2),

data, data.Type, scoring='f1\_weighted', cv=3, return\_train\_score=**True**)

scores

1. : {'fit\_time': array([0.00199509, 0.00199389, 0.00199485]),

'score\_time': array([0.00909901, 0.00448203, 0.00732541]),

'test\_score': array([0.91673789, 0.95774648, 0.95301901]),

'train\_score': array([0.9930479 , 0.98664493, 0.97930491])}

## Стратегия K-Fold

**from sklearn.model\_selection import** KFold

1. :
2. :

kf = KFold(n\_splits=5)

scores = cross\_validate(KNeighborsClassifier(n\_neighbors=2),

data, data.Type, scoring='f1\_weighted', cv=kf)

scores

1. : {'fit\_time': array([0.00261664, 0.00299215, 0.00283647, 0.00299287,

0.00199413]),

'score\_time': array([0.00499034, 0.00514221, 0.00498915, 0.00421095,

0.00345993]),

'test\_score': array([1. , 1. , 0.925 , 0.46055632,

0.01858304])}

## Стратегия Repeated K-Fold

**from sklearn.model\_selection import** RepeatedKFold

1. :
2. :

kf = RepeatedKFold(n\_splits=3, n\_repeats=2)

scores = cross\_validate(KNeighborsClassifier(n\_neighbors=2),

data, data.Type, scoring='f1\_weighted', cv=kf)

scores

1. : {'fit\_time': array([0.00273538, 0.00398898, 0.00199461, 0.00199223, 0.00199747,

0.00374722]),

'score\_time': array([0.00698185, 0.00541639, 0.00603795, 0.00514269,

0.00418472,

0.00698137]),

'test\_score': array([0.98703704, 0.93623007, 0.94372091, 0.94989013,

0.95848844,

0.97184521])}

## Стратегия Leave One Out

**from sklearn.model\_selection import** LeaveOneOut

1. :
2. :

kf = LeaveOneOut()

scores = cross\_validate(KNeighborsClassifier(n\_neighbors=2),

data, data.Type, scoring='f1\_weighted', cv=kf)

scores

1. : {'fit\_time': array([0.00299263, 0.00304389, 0.00299311, 0.00294232, 0.00214601,

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0.00206876, | 0.00199461, | 0.00166726, | 0.00157142, | 0.00199485, |
| 0.002141 , | 0.00199485, | 0.00199509, | 0.00186849, | 0.00209737, |
| 0.00199437, | 0.00199485, | 0.0029912 , | 0.00206327, | 0.0026257 , |
| 0.00199485, | 0.00299263, | 0.00299239, | 0.00182676, | 0.00398898, |
| 0.00299525, | 0.00263333, | 0.00166082, | 0.00299263, | 0.00207448, |
| 0.00278831, | 0.00199389, | 0.00374508, | 0.00220704, | 0.0019176 , |
| 0.00398898, | 0.00254464, | 0.00399017, | 0.0029912 , | 0.00199485, |
| 0.00199556, | 0.00199342, | 0.00199509, | 0.00292563, | 0.00207496, |
| 0.00299168, | 0.00195789, | 0.00209022, | 0.00144935, | 0.00196719, |
| 0.00199485, | 0.00209713, | 0.00199461, | 0.00199509, | 0.00199604, |
| 0.00199437, | 0.00314999, | 0.00299168, | 0.00199461, | 0.00233293, |
| 0.00299168, | 0.00188208, | 0.00263381, | 0.00308251, | 0.00199461, |
| 0.00199461, | 0.00196195, | 0.00230861, | 0.00199437, | 0.00207138, |
| 0.00598431, | 0.00204515, | 0.00178123, | 0.00398684, | 0.00299525, |
| 0.00218606, | 0.00315523, | 0.00199437, | 0.00268507, | 0.00199413, |
| 0.00253081, | 0.00197935, | 0.0026865 , | 0.00299048, | 0.00299287, |
| 0.00262475, | 0.00299191, | 0.00299048, | 0.00297999, | 0.00193882, |
| 0.00201511, | 0.00299215, | 0.00211573, | 0.00199485, | 0.00199485, |
| 0.00199509, | 0.00199389, | 0.00199461, | 0.00199533, | 0.00099754, |
| 0.00199485, | 0.00287938, | 0.00199413, | 0.00199437, | 0.00199533, |
| 0.00299191, | 0.00199294, | 0.00199437, | 0.00295997, | 0.00158596, |
| 0.00302958, | 0.00199366, | 0.00195098, | 0.00207853, | 0.00199461, |
| 0.00199485, | 0.00203109, | 0.00173163, | 0.00175428, | 0.00299048, |
| 0.00299454, | 0.00199437, | 0.00299239, | 0.00199461, | 0.00199437, |
| 0.00271916, | 0.00229812, | 0.00231457, | 0.00199413, | 0.00299144, |
| 0.00199461, | 0.00299239, | 0.00199437, | 0.00211835, | 0.00299025, |
| 0.0029912 , | 0.00198984, | 0.00199127, | 0.00199556, | 0.00199413, |
| 0.00299907, | 0.00299072, | 0.00099707, | 0.0029912 , | 0.00299311, |
| 0.00099802, | 0.00199223, | 0.00199485, | 0.00199866, | 0.00199413, |
| 0.00299144, | 0.00199509, | 0.00202751, | 0.00199461, | 0.00199556, |
| 0.00202799, | 0.00199342, | 0.00199628, | 0.00199437, | 0.0019958 , |
| 0.00195622, | 0.00199485, | 0.00216317, | 0.00298977, | 0.00218344, |
| 0.00208235, | 0.00199461, | 0.00247383, | 0.00182295, | 0.00207329, |
| 0.00199437, | 0.00299239, | 0.00197577, | 0.00199556, | 0.00199485, |
| 0.00199437, | 0.00188446, | 0.00290966, | 0.00129986, | 0.00199485, |
| 0.00199485, | 0.00199485, | 0.00199533, | 0.00199413, | 0.00414681, |
| 0.00299048, | 0.00598407, | 0.00299263, | 0.00208616, | 0.00305486, |
| 0.00365639, | 0.00299263, | 0.00299239, | 0.00199461, | 0.00199509, |
| 0.00270534, | 0.00199509, | 0.00279617, | 0.00299048, | 0.00311613, |
| 0.00334263, | 0.00199294, | 0.00399089, | 0.00150847, | 0.00255609, |
| 0.00199342, | 0.00160646, | 0.0019958 , | 0.0029912 , | 0.00207615, |
| 0.00299454, | 0.00195217, | 0.00299406, | 0.0025053 ]), | |

'score\_time': array([0.00498676, 0.00293851, 0.00304151, 0.00299311,

0.00283837,

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0.00208092, | 0.00398946, | 0.00299168, | 0.00299239, | 0.00299144, |
| 0.00208187, | 0.00199485, | 0.00311756, | 0.00343466, | 0.00209188, |
| 0.00299215, | 0.00299263, | 0.00210524, | 0.00317907, | 0.00299287, |
| 0.00299144, | 0.00299072, | 0.00299239, | 0.00299263, | 0.00351882, |
| 0.00598121, | 0.00342131, | 0.00688767, | 0.00324631, | 0.00385809, |
| 0.00498652, | 0.00299239, | 0.00323677, | 0.0027554 , | 0.00443459, |
| 0.00398993, | 0.00244069, | 0.00398946, | 0.00199437, | 0.0039897 , |
| 0.00398993, | 0.00299168, | 0.00705123, | 0.00298882, | 0.00457907, |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0.00399494, | 0.00202608, | 0.00383186, | 0.00199485, | 0.00208068, |
| 0.00299239, | 0.00245929, | 0.00196958, | 0.00598478, | 0.00398755, |
| 0.00299191, | 0.00383234, | 0.00281739, | 0.00299287, | 0.00365114, |
| 0.00206542, | 0.00299239, | 0.00299096, | 0.0021081 , | 0.00299239, |
| 0.004987 , | 0.00302386, | 0.00355268, | 0.0043931 , | 0.00251102, |
| 0.00398898, | 0.00210381, | 0.00299621, | 0.00324845, | 0.00299239, |
| 0.00479627, | 0.00239229, | 0.004987 , | 0.00329709, | 0.00245738, |
| 0.00299168, | 0.00430846, | 0.00299358, | 0.00399041, | 0.00298977, |
| 0.00236297, | 0.00299335, | 0.00200701, | 0.00199437, | 0.00364923, |
| 0.00299215, | 0.00199747, | 0.0038681 , | 0.00213933, | 0.00295234, |
| 0.00299239, | 0.00299191, | 0.00299168, | 0.00199485, | 0.00299168, |
| 0.0021081 , | 0.00199389, | 0.00299191, | 0.00225091, | 0.00299358, |
| 0.0034833 , | 0.00199485, | 0.00302553, | 0.00299144, | 0.00294352, |
| 0.00195718, | 0.0040319 , | 0.00299191, | 0.00314999, | 0.00398922, |
| 0.00259447, | 0.00295568, | 0.00399017, | 0.00272894, | 0.00299239, |
| 0.00298929, | 0.00299215, | 0.00299191, | 0.0039897 , | 0.00299215, |
| 0.00299096, | 0.00368404, | 0.00269437, | 0.00398993, | 0.00199485, |
| 0.00398874, | 0.00299263, | 0.00299191, | 0.00386834, | 0.00299191, |
| 0.00199914, | 0.00399351, | 0.00299191, | 0.00299144, | 0.00299287, |
| 0.00298524, | 0.00199485, | 0.00398993, | 0.00302601, | 0.00199151, |
| 0.00395894, | 0.00299287, | 0.00322485, | 0.00299287, | 0.00299215, |
| 0.00199485, | 0.0029943 , | 0.00395465, | 0.00299072, | 0.00299215, |
| 0.00299454, | 0.00199437, | 0.00299311, | 0.00298929, | 0.00217223, |
| 0.00299191, | 0.00299311, | 0.00481939, | 0.00299382, | 0.00299191, |
| 0.00206137, | 0.00251293, | 0.00321484, | 0.00309753, | 0.00205994, |
| 0.00299239, | 0.00301027, | 0.00299191, | 0.00210619, | 0.00299239, |
| 0.002105 , | 0.00407147, | 0.00199509, | 0.00268936, | 0.00299168, |
| 0.00464845, | 0.00199509, | 0.00251675, | 0.00316215, | 0.00267005, |
| 0.00199509, | 0.00299191, | 0.00299096, | 0.00290036, | 0.00392723, |
| 0.0023272 , | 0.00398898, | 0.00299072, | 0.00335503, | 0.00399208, |
| 0.002913 , | 0.00318766, | 0.00399113, | 0.00274873, | 0.00386572, |
| 0.00360894, | 0.0051055 , | 0.00298953, | 0.00299311, | 0.00342798, |
| 0.00299191, | 0.00299335, | 0.00398779, | 0.00247765, | 0.00442386, |
| 0.00298882, | 0.00203753, | 0.00299001, | 0.0024817 ]), | |

'test\_score': array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1., 1., | 1., |  | | | | | | | | | | | | | | | |
|  | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., |
|  | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., |
|  | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., |
|  | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., |
|  | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., |
|  | 1., | 0., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., |
|  | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., |
|  | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., |
|  | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 0., | 1., | 1., | 1., | 1., | 1., | 1., |
|  | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 0., | 1., | 0., | 0., | 1., | 1., | 0., |
|  | 0., | 0., | 0., | 0., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 1., | 0., | 1., | 1., |

1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])}

## Стратегия Leave P Out

**from sklearn.model\_selection import** LeavePOut

1. :
2. :

kf = LeavePOut(2)

scores = cross\_validate(KNeighborsClassifier(n\_neighbors=2),

data, data.Type, scoring='f1\_weighted', cv=kf)

scores

1. : {'fit\_time': array([0.00199461, 0.00498605, 0.00250959, …, 0.00135422,

0.00156355,

0.00159883]),

'score\_time': array([0.00398946, 0.00628161, 0.00447178, …, 0.00299239,

0.0025394 ,

0.00239086]),

'test\_score': array([1., 1., 1., …, 1., 1., 1.])}

## Стратегия ShuffleSplit

**from sklearn.model\_selection import** ShuffleSplit

1. :
2. :

kf = ShuffleSplit(n\_splits=5, test\_size=0.25)

scores = cross\_validate(KNeighborsClassifier(n\_neighbors=2),

data, data.Type, scoring='f1\_weighted', cv=kf)

scores

1. : {'fit\_time': array([0.00199342, 0.00301981, 0.00199485, 0.00199246,

0.00206017]),

'score\_time': array([0.00598359, 0.0041585 , 0.00399232, 0.00446057,

0.00345206]),

'test\_score': array([1. , 0.94182566, 0.97016461, 0.96090535,

0.94729345])}

## Стратегия StratifiedKFold

**from sklearn.model\_selection import** StratifiedKFold

1. :
2. :

skf = StratifiedKFold(n\_splits=3)

scores = cross\_validate(KNeighborsClassifier(n\_neighbors=2),

data, data.Type, scoring='f1\_weighted', cv=skf)

scores

1. : {'fit\_time': array([0.00198126, 0.00498486, 0.0029912 ]),

'score\_time': array([0.00500107, 0.00498891, 0.00498557]),

'test\_score': array([0.91673789, 0.95774648, 0.95301901])}

## Оптимизация гиперпараметра

**Через GridSearch**

**from sklearn.model\_selection import** GridSearchCV

1. :
2. :

n\_range = np.array(range(5,55,5)) tuned\_parameters = [{'n\_neighbors': n\_range}]

tuned\_parameters

1. : [{'n\_neighbors': array([ 5, 10, 15, 20, 25, 30, 35, 40, 45, 50])}]
2. :



%%time

clf\_gs = GridSearchCV(KNeighborsClassifier(), tuned\_parameters, cv=5,

*‹→*scoring='accuracy')

clf\_gs.fit(X\_train, y\_train)

CPU times: total: 172 ms Wall time: 263 ms

1. : GridSearchCV(cv=5, estimator=KNeighborsClassifier(),

param\_grid=[{'n\_neighbors': array([ 5, 10, 15, 20, 25, 30, 35, 40,

45, 50])}],

scoring='accuracy')

1. :

clf\_gs.cv\_results\_

1. : {'mean\_fit\_time': array([0.00271177, 0.00205445, 0.00173397, 0.00169015,

0.00211964,

0.00198412, 0.0016819 , 0.00136933, 0.00195174, 0.00172234]),

'std\_fit\_time': array([0.00127936, 0.00053822, 0.00040229, 0.000397 ,

0.00029571,

0.00068425, 0.00026905, 0.00025122, 0.00013681, 0.0004558 ]),

'mean\_score\_time': array([0.0037106 , 0.00255346, 0.00259519, 0.00269642,

0.00253229,

0.00290351, 0.00255089, 0.00291533, 0.00264716, 0.00290046]),

'std\_score\_time': array([0.00116456, 0.00038766, 0.00035512, 0.00033663,

0.00039703,

0.00029038, 0.00044796, 0.00028231, 0.00034596, 0.00027754]),

'param\_n\_neighbors': masked\_array(data=[5, 10, 15, 20, 25, 30, 35, 40, 45, 50], mask=[False, False, False, False, False, False, False, False,

False, False], fill\_value='?',

dtype=object), 'params': [{'n\_neighbors': 5},

{'n\_neighbors': 10},

{'n\_neighbors': 15},

{'n\_neighbors': 20},

{'n\_neighbors': 25},

{'n\_neighbors': 30},

{'n\_neighbors': 35},

{'n\_neighbors': 40},

{'n\_neighbors': 45},

{'n\_neighbors': 50}],

'split0\_test\_score': array([0.5625 , 0.5625 , 0.5625 , 0.59375, 0.5625 ,

0.53125, 0.53125,

0.625 , 0.53125, 0.40625]),

'split1\_test\_score': array([0.65625, 0.6875 , 0.625 , 0.5625 , 0.59375,

0.59375, 0.5625 ,

0.59375, 0.5625 , 0.53125]),

'split2\_test\_score': array([0.625 , 0.5625 , 0.59375, 0.5625 , 0.59375, 0.5625

, 0.59375,

0.5 , 0.5 , 0.53125]),

'split3\_test\_score': array([0.53125, 0.5 , 0.5 , 0.53125, 0.4375 , 0.375

, 0.40625,

0.40625, 0.34375, 0.34375]),

'split4\_test\_score': array([0.75 , 0.71875, 0.6875 , 0.65625, 0.65625,

0.65625, 0.59375,

0.625 , 0.53125, 0.46875]),

'mean\_test\_score': array([0.625 , 0.60625, 0.59375, 0.58125, 0.56875, 0.54375,

0.5375 ,

1. :

0.55 , 0.49375, 0.45625]),

'std\_test\_score': array([0.07654655, 0.08291562, 0.0625 , 0.04238956,

0.07234898,

0.0939581 , 0.06959705, 0.08523864, 0.07756046, 0.0728869 ]),

'rank\_test\_score': array([ 1, 2, 3, 4, 5, 7, 8, 6, 9, 10])}

Лучшая модель:

clf\_gs.best\_estimator\_

1. : KNeighborsClassifier()

Лучшее значение метрики:

1. :

clf\_gs.best\_score\_

[41]: 0.625

Лучшее значение параметров:

1. :

clf\_gs.best\_params\_

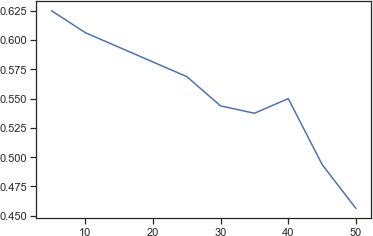
1. : {'n\_neighbors': 5}

Изменение качества на тестовой выборке в зависимости от К-соседей:

1. :

plt.plot(n\_range, clf\_gs.cv\_results\_['mean\_test\_score'])

1. : [<matplotlib.lines.Line2D at 0x26c1bf0f820>]



## Через RandomizedSearch

**from sklearn.model\_selection import** RandomizedSearchCV

1. :
2. :



%%time

clf\_rs = RandomizedSearchCV(KNeighborsClassifier(), tuned\_parameters, cv=5,

*‹→*scoring='accuracy')

clf\_rs.fit(X\_train, y\_train)

CPU times: total: 234 ms Wall time: 297 ms

[45]: RandomizedSearchCV(cv=5, estimator=KNeighborsClassifier(),

param\_distributions=[{'n\_neighbors': array([ 5, 10, 15, 20,

25, 30, 35, 40, 45, 50])}],

scoring='accuracy')

Оптимальные параметры:

[46]:

clf\_rs.best\_score\_, clf\_rs.best\_params\_

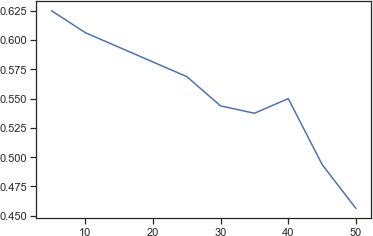
[46]: (0.625, {'n\_neighbors': 5})

Изменение качества на тестовой выборке в зависимости от К-соседей:

[47]:

plt.plot(n\_range, clf\_rs.cv\_results\_['mean\_test\_score'])

[47]: [<matplotlib.lines.Line2D at 0x26c1b366790>]



[49]:

## Построение оптимальной модели

Оптимальное число ближайших соседей = 5. Построим оптимальную модель:

clf\_o = KNeighborsClassifier(n\_neighbors=5) clf\_o.fit(X\_train, y\_train)

target\_o = clf\_o.predict(X\_test) len(target\_o), target\_o

[49]: (54,

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| array([5, | 7, | 2, | 2, | 1, | 1, | 2, | 1, | 2, | 1, 2, 1, 2, 1, 1, 2, 1, 1, 1, 1, | 1, | 7, |
| 2, | 1, | 1, | 1, | 2, | 2, | 1, | 1, | 5, | 6, 1, 1, 1, 2, 1, 1, 5, 1, 1, 7, | 7, | 1, |
| 7, | 1, | 1, | 1, | 2, | 7, | 2, | 2, | 1, | 2], dtype=int64)) |  |  |

[50]:

# Оценка качества исходной и оптимальной модели

## Метрика Accuracy

Она вычисляет процент (в долях) правильно определенных типов.

**from sklearn.metrics import** accuracy\_score, balanced\_accuracy\_score

Проверим для всех типов исходной модели:

[51]:

accuracy\_score(y\_test, target\_i)

[51]: 0.6851851851851852

И оптимальной модели:

[52]:

accuracy\_score(y\_test, target\_o)

[52]: 0.7222222222222222

Видно, что точность оптимальной модели (5 ближайших соседей) выше, чем в исходной модели (20).

Теперь проверим для каждого конкретного типа:

[53]:

**def** accuracy\_score\_for\_types( y\_true: np.ndarray,

y\_pred: np.ndarray) -> Dict[int, float]: d = {'t': y\_true, 'p': y\_pred}

df = pd.DataFrame(data=d) types = np.unique(y\_true) res = dict()

**for** t **in** types:

temp\_data\_flt = df[df['t']==t] temp\_acc = accuracy\_score(

temp\_data\_flt['t'].values, temp\_data\_flt['p'].values)

res[t] = temp\_acc

**return** res

**def** print\_accuracy\_score\_for\_types( y\_true: np.ndarray,

y\_pred: np.ndarray):

accs = accuracy\_score\_for\_types(y\_true, y\_pred)

**if** len(accs)>0:

print('Тип **\t** Accuracy')

**for** i **in** accs:

print('**{} \t {}**'.format(i, accs[i]))

Для исходной модели:

[54]:

print\_accuracy\_score\_for\_types(y\_test, target\_i)

|  |  |
| --- | --- |
| Тип | Accuracy |
| 1 | 0.8260869565217391 |
| 2 | 0.6875 |
| 3 | 0.0 |
| 5 | 1.0 |
| 6 | 0.0 |
| 7 | 1.0 |

[55]:

Видим, что процент “Accuracy” для типа 1 составляет 83%, для типа 2 - 68%, для типа 3 - 0%. Для типов 5, 6, 7 “Accuracy” составляет 100%.

Для оптимальной модели:

print\_accuracy\_score\_for\_types(y\_test, target\_o)

|  |  |
| --- | --- |
| Тип | Accuracy |
| 1 | 0.8695652173913043 |
| 2 | 0.6875 |
| 3 | 0.0 |
| 5 | 1.0 |
| 6 | 1.0 |
| 7 | 1.0 |

Результаты схожи, но у типа 1 метрика составляет уже 87%.

## Метрика balanced\_accuracy\_score

Используется для бинарной классификации. Сконвертируем данные и выведем метрику:

[56]:

**def** convert\_target\_to\_binary(array:np.ndarray, target:int) -> np.ndarray: res = [1 **if** x==target **else** 0 **for** x **in** array]

**return** res

[57]:

bin\_y\_train = convert\_target\_to\_binary(y\_train, 2) list(zip(y\_train, bin\_y\_train))[:10]

|  |  |  |
| --- | --- | --- |
| [57]: | [(5, | 0), |
|  | (2, | 1), |
|  | (7, | 0), |
|  | (1, | 0), |
|  | (7, | 0), |
|  | (7, | 0), |
|  | (2, | 1), |
|  | (2, | 1), |
|  | (2, | 1), |
|  | (2, | 1)] |

[58]:

bin\_y\_test = convert\_target\_to\_binary(y\_test, 2) list(zip(y\_test, bin\_y\_test))[:10]

|  |  |  |
| --- | --- | --- |
| [58]: | [(2, | 1), |
|  | (7, | 0), |
|  | (2, | 1), |
|  | (2, | 1), |
|  | (1, | 0), |
|  | (1, | 0), |

|  |  |
| --- | --- |
| (1, | 0), |
| (3, | 0), |
| (1, | 0), |
| (1, | 0)] |

Для исходной модели:

[59]:

bin\_target\_i = convert\_target\_to\_binary(target\_i, 2)

[60]:

balanced\_accuracy\_score(bin\_y\_test, bin\_target\_i)

[60]: 0.7516447368421053

Для оптимальной модели:

[61]:

bin\_target\_o = convert\_target\_to\_binary(target\_o, 2)

[62]:

balanced\_accuracy\_score(bin\_y\_test, bin\_target\_o)

[62]: 0.7911184210526316

Видно, что у исходной модели метрика составляет 75%, а у оптимальной - 79%.

[63]:

## Метрика “Матрица ошибок”

Создадим матрицу с помощью функции confusion\_matrix:

**from sklearn.metrics import** ConfusionMatrixDisplay

**from sklearn.metrics import** confusion\_matrix

Для исходной модели:

[64]:

confusion\_matrix(y\_test, target\_i, labels=[0, 1, 2, 3, 4, 5, 6, 7])

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [64]: | array([[ | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0], |
|  | [ | 0, | 19, | 4, | 0, | 0, | 0, | 0, | 0], |
|  | [ | 0, | 3, | 11, | 0, | 0, | 2, | 0, | 0], |
|  | [ | 0, | 4, | 3, | 0, | 0, | 0, | 0, | 0], |
|  | [ | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0], |
|  | [ | 0, | 0, | 0, | 0, | 0, | 1, | 0, | 0], |
|  | [ | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 1], |
|  | [ | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 6]], dtype=int64) |

[65]:

Визуально представим матрицу ошибок, показывающую количество верно и ошибочно клас- сифицированных данных:

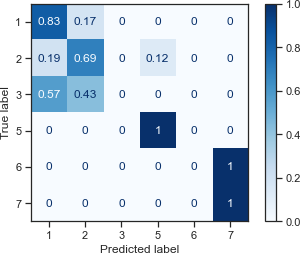
1. : <sklearn.metrics.\_plot.confusion\_matrix.ConfusionMatrixDisplay at 0x26c09b2c580>

ConfusionMatrixDisplay.from\_estimator( clf\_i,

X\_test, y\_test,

display\_labels=clf\_i.classes\_, cmap=plt.cm.Blues, normalize='true',

)



Для оптимальной модели:

1. :

confusion\_matrix(y\_test, target\_o, labels=[0, 1, 2, 3, 4, 5, 6, 7])

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [66]: | array([[ | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0], |
|  | [ | 0, | 20, | 3, | 0, | 0, | 0, | 0, | 0], |
|  | [ | 0, | 3, | 11, | 0, | 0, | 2, | 0, | 0], |
|  | [ | 0, | 6, | 1, | 0, | 0, | 0, | 0, | 0], |
|  | [ | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0], |
|  | [ | 0, | 0, | 0, | 0, | 0, | 1, | 0, | 0], |
|  | [ | 0, | 0, | 0, | 0, | 0, | 0, | 1, | 0], |
|  | [ | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 6]], dtype=int64) |

1. :

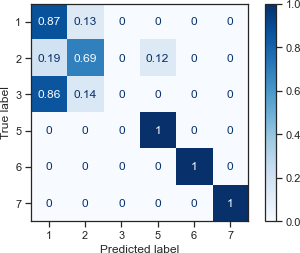
ConfusionMatrixDisplay.from\_estimator( clf\_o,

X\_test, y\_test,

display\_labels=clf\_o.classes\_, cmap=plt.cm.Blues, normalize='true',

)

1. : <sklearn.metrics.\_plot.confusion\_matrix.ConfusionMatrixDisplay at 0x26c1bfd7970>



Точность у оптимальной модели выше, чем у исходной.

1. :
2. :

## Метрика Precision

Precision показывает долю верно предсказанных классификатором положительных объек- тов из всех объектов, которые классификатор верно или неверно определил как положитель- ные.

**from sklearn.metrics import** precision\_score

Для исходной модели:

precision\_score(bin\_y\_test, bin\_target\_i)

[69]: 0.6111111111111112

Для оптимальной модели:

1. :

precision\_score(bin\_y\_test, bin\_target\_o)

[70]: 0.7333333333333333

Также видна улучшенная точность у оптимальной по отношению к исходной (73% и 61%).

1. :
2. :

## Метрика Recall

Recall показывает долю верно предсказанных классификатором положительных объектов, из всех действительно положительных объектов.

**from sklearn.metrics import** recall\_score

Для исходной:

recall\_score(bin\_y\_test, bin\_target\_i)

[72]: 0.6875

Для оптимальной:

1. :

recall\_score(bin\_y\_test, bin\_target\_o)

[73]: 0.6875

Точность моделей одинакова - 69%.

1. :
2. :

## Метрика F1-мера

Для объединения метрик Precision и Recall используют F-меру - среднее гармоническое от Precision и Recall. В F1 мере вес точности = 1.

**from sklearn.metrics import** f1\_score

Для исходной:

f1\_score(bin\_y\_test, bin\_target\_i)

[75]: 0.6470588235294118

Для оптимальной:

1. :

f1\_score(bin\_y\_test, bin\_target\_o)

[76]: 0.7096774193548386

Точность оптимальной модели выше.

1. :
2. :

## Вывод метрик через classification\_report

Функция classification\_report позволяет выводить значения точности, полноты и F-меры для всех классов выборки:

**from sklearn.metrics import** classification\_report

Для исходной:

**import warnings** warnings.filterwarnings('ignore') classification\_report(y\_test, target\_i,

target\_names=clf\_i.classes\_, output\_dict=**True**)

1. : {1: {'precision': 0.7307692307692307,

'recall': 0.8260869565217391,

'f1-score': 0.7755102040816326,

'support': 23},

2: {'precision': 0.6111111111111112,

'recall': 0.6875,

'f1-score': 0.6470588235294118,

'support': 16},

3: {'precision': 0.0, 'recall': 0.0, 'f1-score': 0.0, 'support': 7},

5: {'precision': 0.3333333333333333,

'recall': 1.0,

'f1-score': 0.5,

'support': 1},

6: {'precision': 0.0, 'recall': 0.0, 'f1-score': 0.0, 'support': 1},

7: {'precision': 0.8571428571428571,

'recall': 1.0,

'f1-score': 0.923076923076923,

'support': 6},

'accuracy': 0.6851851851851852,

'macro avg': {'precision': 0.42205942205942204, 'recall': 0.5855978260869565,

'f1-score': 0.4742743251146612,

'support': 54},

'weighted avg': {'precision': 0.593734454845566,

1. :

'recall': 0.6851851851851852,

'f1-score': 0.6338543964594385,

'support': 54}}

Для оптимальной:

**import warnings** warnings.filterwarnings('ignore') classification\_report(y\_test, target\_o,

target\_names=clf\_o.classes\_, output\_dict=**True**)

1. : {1: {'precision': 0.6896551724137931,

'recall': 0.8695652173913043,

'f1-score': 0.7692307692307693,

'support': 23},

2: {'precision': 0.7333333333333333,

'recall': 0.6875,

'f1-score': 0.7096774193548386,

'support': 16},

3: {'precision': 0.0, 'recall': 0.0, 'f1-score': 0.0, 'support': 7},

5: {'precision': 0.3333333333333333,

'recall': 1.0,

'f1-score': 0.5,

'support': 1},

6: {'precision': 1.0, 'recall': 1.0, 'f1-score': 1.0, 'support': 1},

7: {'precision': 1.0, 'recall': 1.0, 'f1-score': 1.0, 'support': 6},

'accuracy': 0.7222222222222222,

'macro avg': {'precision': 0.6260536398467433, 'recall': 0.7595108695652174,

'f1-score': 0.6631513647642681,

'support': 54},

'weighted avg': {'precision': 0.6468284376330352, 'recall': 0.7222222222222222,

'f1-score': 0.6767990074441687,

'support': 54}}

1. :

## ROC-кривая и ROC AUC

Используется для оценки качества бинарной классификации.

Обучим исходную модель на основе бинарной классифкации, чтобы получить вероятности типов:

bin\_clf\_i = KNeighborsClassifier(n\_neighbors=20) bin\_clf\_i.fit(X\_train, bin\_y\_train) bin\_clf\_i.predict(X\_test)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [80]: array([0, | 0, | 0, | 0, | 0, | 0, | 1, | 0, | 1, | 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, |
| 1, | 0, | 0, | 0, | 1, | 1, | 0, | 0, | 0, | 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, |
| 0, | 0, | 1, | 0, | 1, | 0, | 0, | 1, | 0, | 1]) |

Предскажем вероятности типов:

1. :

proba\_target\_i = bin\_clf\_i.predict\_proba(X\_test) len(proba\_target\_i), proba\_target\_i

[81]: (54,

|  |  |
| --- | --- |
| array([[0.75, | 0.25], |
| [1. , | 0. ], |
| [0.55, | 0.45], |
| [0.55, | 0.45], |
| [0.6 , | 0.4 ], |
| [0.8 , | 0.2 ], |
| [0.45, | 0.55], |
| [0.7 , | 0.3 ], |
| [0.3 , | 0.7 ], |
| [0.75, | 0.25], |
| [0.2 , | 0.8 ], |
| [0.95, | 0.05], |
| [0.45, | 0.55], |
| [0.8 , | 0.2 ], |
| [0.65, | 0.35], |
| [0.55, | 0.45], |
| [0.55, | 0.45], |
| [0.75, | 0.25], |
| [0.95, | 0.05], |
| [0.95, | 0.05], |
| [0.8 , | 0.2 ], |
| [1. , | 0. ], |
| [0.25, | 0.75], |
| [0.75, | 0.25], |
| [0.6 , | 0.4 ], |
| [0.65, | 0.35], |
| [0.15, | 0.85], |
| [0.3 , | 0.7 ], |
| [0.5 , | 0.5 ], |
| [0.9 , | 0.1 ], |
| [0.75, | 0.25], |
| [0.85, | 0.15], |
| [0.8 , | 0.2 ], |
| [0.75, | 0.25], |
| [0.8 , | 0.2 ], |
| [0.55, | 0.45], |
| [0.45, | 0.55], |
| [0.55, | 0.45], |
| [0.75, | 0.25], |
| [0.8 , | 0.2 ], |
| [0.8 , | 0.2 ], |
| [1. , | 0. ], |
| [1. , | 0. ], |
| [0.9 , | 0.1 ], |
| [1. , | 0. ], |
| [0.75, | 0.25], |
| [0.4 , | 0.6 ], |
| [0.85, | 0.15], |
| [0.2 , | 0.8 ], |
| [1. , | 0. ], |
| [0.65, | 0.35], |
| [0.4 , | 0.6 ], |

1. :

[0.8 , 0.2 ],

[0.3 , 0.7 ]]))

Вероятность единичного класса:

true\_proba\_target\_i = proba\_target\_i[:,1] true\_proba\_target\_i

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [82]: | array([0.25, | 0. , | 0.45, | 0.45, | 0.4 , | 0.2 , | 0.55, | 0.3 , | 0.7 , | 0.25, | 0.8 | , |
|  | 0.05, | 0.55, | 0.2 , | 0.35, | 0.45, | 0.45, | 0.25, | 0.05, | 0.05, | 0.2 , | 0. | , |
|  | 0.75, | 0.25, | 0.4 , | 0.35, | 0.85, | 0.7 , | 0.5 , | 0.1 , | 0.25, | 0.15, | 0.2 | , |
|  | 0.25, | 0.2 , | 0.45, | 0.55, | 0.45, | 0.25, | 0.2 , | 0.2 , | 0. , | 0. , | 0.1 | , |
|  | 0. , | 0.25, | 0.6 , | 0.15, | 0.8 , | 0. , | 0.35, | 0.6 , | 0.2 , | 0.7 ]) |  |  |

ROC-кривая:

1. :

**from sklearn.metrics import** roc\_curve, roc\_auc\_score

1. :

**def** draw\_roc\_curve(y\_true, y\_score, pos\_label, average): fpr, tpr, thresholds = roc\_curve(y\_true, y\_score,

pos\_label=pos\_label) roc\_auc\_value = roc\_auc\_score(y\_true, y\_score, average=average) plt.figure()

lw = 2

plt.plot(fpr, tpr, color='darkorange',

lw=lw, label='ROC curve (area = **%0.2f**)' % roc\_auc\_value) plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')

plt.xlim([0.0, 1.0])

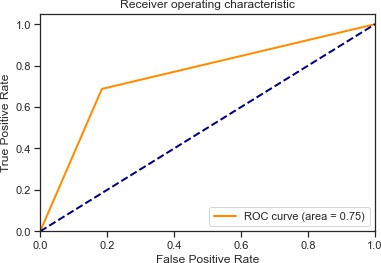
plt.ylim([0.0, 1.05]) plt.xlabel('False Positive Rate') plt.ylabel('True Positive Rate')

plt.title('Receiver operating characteristic') plt.legend(loc=Dlower rightD)

plt.show()

1. :

draw\_roc\_curve(bin\_y\_test, bin\_target\_i, pos\_label=1, average='micro')



Сделаем тоже самое для оптимальной модели:

1. :

bin\_clf\_o = KNeighborsClassifier(n\_neighbors=5) bin\_clf\_o.fit(X\_train, bin\_y\_train) bin\_clf\_o.predict(X\_test)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [86]: | array([0, | 0, | 1, | 1, | 0, | 0, | 1, | 0, | 1, | 0, 1, 0, 1, 0, | 0, | 1, | 0, | 0, | 0, | 0, | 0, | 0, |
|  | 1, | 0, | 0, | 0, | 1, | 1, | 0, | 0, | 0, | 0, 0, 0, 0, 1, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, |
|  | 0, | 0, | 0, | 0, | 1, | 0, | 1, | 1, | 0, | 1]) |  |  |  |  |  |  |  |  |

1. :

proba\_target\_o = bin\_clf\_o.predict\_proba(X\_test) len(proba\_target\_o), proba\_target\_o

[87]: (54,

|  |  |
| --- | --- |
| array([[1. , | 0. ], |
| [1. , | 0. ], |
| [0.4, | 0.6], |
| [0.2, | 0.8], |
| [0.8, | 0.2], |
| [1. , | 0. ], |
| [0.4, | 0.6], |
| [0.8, | 0.2], |
| [0.4, | 0.6], |
| [1. , | 0. ], |
| [0. , | 1. ], |
| [1. , | 0. ], |
| [0.2, | 0.8], |
| [1. , | 0. ], |
| [1. , | 0. ], |
| [0.2, | 0.8], |
| [0.6, | 0.4], |
| [0.8, | 0.2], |
| [1. , | 0. ], |
| [1. , | 0. ], |
| [0.8, | 0.2], |
| [1. , | 0. ], |
| [0.2, | 0.8], |
| [1. , | 0. ], |
| [0.8, | 0.2], |
| [0.8, | 0.2], |
| [0.2, | 0.8], |
| [0. , | 1. ], |
| [0.8, | 0.2], |
| [1. , | 0. ], |
| [1. , | 0. ], |
| [0.8, | 0.2], |
| [1. , | 0. ], |
| [1. , | 0. ], |
| [0.8, | 0.2], |
| [0.4, | 0.6], |
| [0.8, | 0.2], |
| [0.8, | 0.2], |
| [0.8, | 0.2], |
| [1. , | 0. ], |
| [1. , | 0. ], |
| [1. , | 0. ], |

|  |  |  |  |
| --- | --- | --- | --- |
| [1. | , | 0. | ], |
| [1. | , | 0. | ], |
| [1. | , | 0. | ], |
| [0.8, | | 0.2], | |
| [0.6, | | 0.4], | |
| [1. , | | 0. ], | |
| [0.2, | | 0.8], | |
| [1. , | | 0. ], | |
| [0.4, | | 0.6], | |
| [0.4, | | 0.6], | |
| [0.6, | | 0.4], | |
| [0. , | | 1. ]])) | |

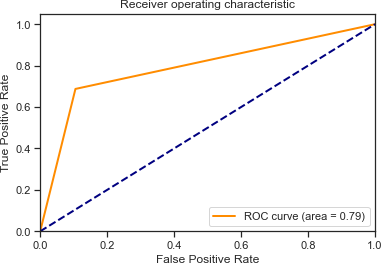
1. :

true\_proba\_target\_o = proba\_target\_o[:,1] true\_proba\_target\_o

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [88]: | array([0. , | 0. | , | 0.6, | 0.8, | 0.2, | 0. , | 0.6, | 0.2, | 0.6, | 0. , | 1. , | 0. , | 0.8, |
|  | 0. , | 0. | , | 0.8, | 0.4, | 0.2, | 0. , | 0. , | 0.2, | 0. , | 0.8, | 0. , | 0.2, | 0.2, |
|  | 0.8, | 1. | , | 0.2, | 0. , | 0. , | 0.2, | 0. , | 0. , | 0.2, | 0.6, | 0.2, | 0.2, | 0.2, |
|  | 0. , | 0. | , | 0. , | 0. , | 0. , | 0. , | 0.2, | 0.4, | 0. , | 0.8, | 0. , | 0.6, | 0.6, |
|  | 0.4, | 1. | ]) |  |  |  |  |  |  |  |  |  |  |  |

1. :

draw\_roc\_curve(bin\_y\_test, bin\_target\_o, pos\_label=1, average='micro')



Видно, что у оптимальной модели выше точность, чем у исходной.