Лабораторная работа №3 по курсу "Технологии машинного обучения"

Выполнила Попова Дарья, студентка группы РТ5-61Б

Метод k ближайших соседей, метрики оценки качества модели, кросс-валидация и подбор гиперпараметров для задачи бинарной классификации

1. Подготовка датасета

In [1]:

```
import pandas as pd
import numpy as np
```

In [2]:

```
heart = pd.read_csv('C:\\Users\\Дасупс\\Downloads\\heart.csv')
```

Будем использовать датасет с данными по сердечно-сосудистым заболеваниям и соответствующим биометрическим показателям. Признаки обозначают:

- age - возраст
- sex - пол пациента
- cp - chest pain - тип грудной боли (закодирован числами от 0 до 3)
- trestbps - артериальное давление в состоянии покоя
- chol - холестерин
- fbs (fasting blood sugar) - уровень сахара в крови натощак
- restecg (resting electrocardiographic results) - результаты ЭКГ (закодирован числами от 0 до 2)
- thalach - максимальное ЧСС
- exang (exercise induced angina) - стенокардия после нагрузки
- oldpeak - снижение ST-сегмента на ЭКГ

In [3]:

```
heart.head()
```

Out[3]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2	1
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2	1
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1

In [4]:

```
heart.shape
```

Out[4]:

```
(303, 14)
```

In [5]:

```
heart.target.unique()
```

Out[5]:

```
array([1, 0], dtype=int64)
```

Как видно из значений целевого признака target, нам предстоит решать задачу бинарной классификации.

In [6]:

```
labels, counters = np.unique(heart.target, return_counts = True)
```

In [7]:

```
type(labels), type(counters)
```

Out[7]:

```
(numpy.ndarray, numpy.ndarray)
```

In [8]:

```
# сделаем из нумпаевских ndarray питоновские списки
labels = labels.tolist()
counters = counters.tolist()
```

Посчитаем и выведем, сколько всего в изначальной выборке образцов каждого из двух классов и соответствующий процент.

In [9]:

```
for i in range(heart.target.nunique()):
    print('Количество образцов класса {} = {} ({}%)'.format(
    labels[i], counters[i], round(100 * counters[i] / heart.shape[0], 2)))
```

```
Количество образцов класса 0 = 138 (45.54%)
Количество образцов класса 1 = 165 (54.46%)
```

Сделаем два датасета: в одном (heart_unscaled) оставим всё как есть, в другом отмасштабируем данные и посмотрим, как это скажется на качестве построенной модели.

In [10]:

```
heart_unscaled = pd.read_csv('C:\\Users\\Дасупс\\Downloads\\heart.csv')
```

In [11]:

```
from sklearn.preprocessing import MinMaxScaler
```

In [12]:

```
minmax_scaler = MinMaxScaler()
```

In [13]:

```
heart[['chol']] = minmax_scaler.fit_transform(heart[['chol']])
heart[['trestbps']] = minmax_scaler.fit_transform(heart[['trestbps']])
heart[['thalach']] = minmax_scaler.fit_transform(heart[['thalach']])
heart[['age']] = minmax_scaler.fit_transform(heart[['age']])
```

In [14]:

```
heart.head()
```

Out[14]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	0.708333	1	3	0.481132	0.244292	1	0	0.603053	0	2.3	0	0	1	1
1	0.166667	1	2	0.339623	0.283105	0	1	0.885496	0	3.5	0	0	2	1
2	0.250000	0	1	0.339623	0.178082	0	0	0.770992	0	1.4	2	0	2	1
3	0.562500	1	1	0.245283	0.251142	0	1	0.816794	0	0.8	2	0	2	1
4	0.583333	0	0	0.245283	0.520548	0	1	0.702290	1	0.6	2	0	2	1

In [15]:

```
heart_unscaled.head()
```

Out[15]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2	1
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2	1
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1

2. Разделение выборки на обучающую и тестовую

Отделим целевой признак (y) от всех остальных (X) и запишем их в соответствующие переменные.

In [16]:

```
X = heart.drop('target', axis=1)
y = heart.target
```

In [17]:

```
type(X), type(y)
```

Out[17]:

```
(pandas.core.frame.DataFrame, pandas.core.series.Series)
```

In [18]:

```
# тот же фокус проделаем с неотмасштабированным датасетом
X_unscaled = heart_unscaled.drop('target', axis=1)
y_unscaled = heart_unscaled.target
```

In [19]:

```
from sklearn.model_selection import train_test_split
```

In [20]:

```
heart_X_train, heart_X_test, heart_y_train, heart_y_test = train_test_split(
    X, y, test_size = 0.3, random_state = 42)
```

In [21]:

```
unscaled_X_train, unscaled_X_test, unscaled_y_train, unscaled_y_test = train_test_split(
    X_unscaled, y_unscaled, test_size = 0.3, random_state = 42)
```

Теперь, когда мы разделили нашу выборку на тестовую обучающую, можно проверить, сохранил ли метод train_test_split баланс классов в heart_y_train и в heart_y_test.

In [22]:

```
labels_y_train, counters_y_train = np.unique(heart_y_train, return_counts = True)
```

In [23]:

```
print('Распределение классов в обучающей выборке:\n')
for i in range(heart_y_train.nunique()):
    print('Количество образцов класса {} = {} ({}%)'.format(
    labels_y_train[i], counters_y_train[i], round(100 * counters_y_train[i] / heart_y_train.shape[0], 2)))
```

```
Распределение классов в обучающей выборке:

Количество образцов класса 0 = 97 (45.75%)
Количество образцов класса 1 = 115 (54.25%)
```

In [24]:

```
labels_y_test, counters_y_test = np.unique(heart_y_test, return_counts = True)
```

In [25]:

```
print('Распределение классов в тестовой выборке:\n')
for i in range(heart_y_test.nunique()):
    print('Количество образцов класса {} = {} ({}%)'.format(
    labels_y_test[i], counters_y_test[i], round(100 * counters_y_test[i] / heart_y_test.shape[0], 2)))
```

```
Распределение классов в тестовой выборке:

Количество образцов класса 0 = 41 (45.05%)
Количество образцов класса 1 = 50 (54.95%)
```

У функции довольно хорошо получилось сохранить баланс классов в обеих выборках.

Модель для алгоритма k ближайших соседей (для произвольного заданного k)

In [26]:

```
from sklearn.neighbors import KNeighborsClassifier
```

In [27]:

```
kneighbors_classifier = KNeighborsClassifier(n_neighbors = 2)

# для отмасштабированных данных
kneighbors_classifier.fit(heart_X_train, heart_y_train)
y_predicted = kneighbors_classifier.predict(heart_X_test)

unscaled_kneighbors_classifier = KNeighborsClassifier(n_neighbors = 2)
# для неотмасштабированных данных
unscaled_kneighbors_classifier.fit(unscaled_X_train, unscaled_y_train)
unscaled_y_predicted = unscaled_kneighbors_classifier.predict(unscaled_X_test)
```

Модель с решающим деревом (с произвольно заданной глубиной)

In [28]:

```
from sklearn import tree
```

In [29]:

```
# для отмасштабированных данных
tree_classifier = tree.DecisionTreeClassifier(criterion = 'entropy', max_depth=4)
tree_classifier.fit(heart_X_train, heart_y_train)
tree_predicted = tree_classifier.predict(heart_X_test)

# для неотмасштабированных данных
tree_classifier = tree.DecisionTreeClassifier(criterion = 'entropy', max_depth=4)
tree_classifier.fit(unscaled_X_train, unscaled_y_train)
unscaled_tree_predicted = tree_classifier.predict(unscaled_X_test)
```

Оценка качества модели (kNN)

In [30]:

```
from sklearn.metrics import accuracy_score, balanced_accuracy_score
from sklearn.metrics import plot_confusion_matrix
from sklearn.metrics import precision_score, recall_score, f1_score
from sklearn.metrics import confusion_matrix
```

Сначала вычислим метрики для модели с отмасштабированными данными.

In [31]:
```
accuracy_score(y_predicted, heart_y_test)
```

Out[31]:
```
0.6923076923076923
```

In [32]:
```
balanced_accuracy_score(y_predicted, heart_y_test)
```

Out[32]:
```
0.7269597457627119
```

В нашем случае нет практически никакого дисбаланса классов (54.5% и 45.5%), поэтому можно заметить, что критичной разницы в значениях между accuracy_score и balanced_accuracy_score не наблюдается.

In [33]:
```
tn, fp, fn, tp = confusion_matrix(heart_y_test, y_predicted).ravel()
```

In [34]:
```
tn, fp, fn, tp
```

Out[34]:
```
(36, 5, 23, 27)
```

In [35]:
```
import matplotlib.pyplot as plt
```

In [36]:
```
plot_confusion_matrix(kneighbors_classifier,
                      heart_X_test, heart_y_test, cmap=plt.cm.Blues)
```

Out[36]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x196a764f700>
```

In [37]:
```
# метрики для модели с отмасштабированными данными
acc, prec, rec, f1 = accuracy_score(y_predicted, heart_y_test), \
precision_score(heart_y_test,y_predicted), \
recall_score(heart_y_test,y_predicted), \
f1_score(heart_y_test,y_predicted)
```

In [38]:
```
# метрики для модели с неотмасштабированными данными
acc_unscaled, prec_unscaled, rec_unscaled, f1_unscaled = \
accuracy_score(unscaled_y_predicted, unscaled_y_test), \
precision_score(unscaled_y_test, unscaled_y_predicted), \
recall_score(unscaled_y_test, unscaled_y_predicted), \
f1_score(unscaled_y_test, unscaled_y_predicted)
```

In [39]:
```
print("{:<12} {:<8} {:<8}".format('metrics','scaled','unscaled'))
print()
metrics_dict = {'accuracy':[round(acc,2), round(acc_unscaled,2)],
                'precision':[round(prec,2), round(prec_unscaled,2)],
                'recall':[round(rec,2), round(rec_unscaled,2)],
                'f1':[round(f1,2), round(f1_unscaled,2)]}
for key, value in metrics_dict.items():
    scaled, unscaled = value
    print ("{:<12} {:<8} {:<8}".format(key, scaled, unscaled))
```

```
metrics      scaled   unscaled

accuracy     0.69     0.62
precision    0.84     0.74
recall       0.54     0.46
f1           0.66     0.57
```

Как и следовало ожидать, та же самая модель (с выбранным наугад значением гиперпараметра) работает *лучше* на отмасштабированных данных.

Вычислим также площадь под ROC-кривой. Для этого сначала запишем вероятности появления классов (степень уверенности нашего классификатора).

In [40]:

```
proba = kneighbors_classifier.predict_proba(heart_X_test)
```

In [41]:

```
proba.shape
```

Out[41]:

```
(91, 2)
```

In [42]:

```
proba
```

Out[42]:

```
array([[1. , 0. ],
       [0.5, 0.5],
       [0. , 1. ],
       [1. , 0. ],
       [0. , 1. ],
       [0. , 1. ],
       [0.5, 0.5],
       [1. , 0. ],
       [1. , 0. ],
       [0.5, 0.5],
       [0. , 1. ],
       [0.5, 0.5],
       [0. , 1. ],
       [1. , 0. ],
       [0. , 1. ],
       [0. , 1. ],
       [0. , 1. ],
       [1. , 0. ],
       [1. , 0. ],
       [1. , 0. ],
       [0.5, 0.5],
       [1. , 0. ],
       [0. , 1. ],
       [0. , 1. ],
       [0. , 1. ],
       [0. , 1. ],
       [0. , 1. ],
       [0.5, 0.5],
       [1. , 0. ],
       [0.5, 0.5],
       [1. , 0. ],
       [1. , 0. ],
       [1. , 0. ],
       [1. , 0. ],
       [0. , 1. ],
       [1. , 0. ],
       [1. , 0. ],
       [0.5, 0.5],
       [1. , 0. ],
       [0.5, 0.5],
       [0.5, 0.5],
       [0. , 1. ],
       [0. , 1. ],
       [1. , 0. ],
       [0.5, 0.5],
       [0.5, 0.5],
       [0.5, 0.5],
       [0. , 1. ],
       [0.5, 0.5],
       [1. , 0. ],
       [0.5, 0.5],
       [1. , 0. ],
       [0. , 1. ],
       [0. , 1. ],
```

```
       [1. , 0. ],
       [1. , 0. ],
       [0.5, 0.5],
       [0. , 1. ],
       [1. , 0. ],
       [1. , 0. ],
       [1. , 0. ],
       [0. , 1. ],
       [0.5, 0.5],
       [0.5, 0.5],
       [0. , 1. ],
       [1. , 0. ],
       [0.5, 0.5],
       [0.5, 0.5],
       [0. , 1. ],
       [0.5, 0.5],
       [0. , 1. ],
       [0. , 1. ],
       [0.5, 0.5],
       [0. , 1. ],
       [1. , 0. ],
       [0. , 1. ],
       [0. , 1. ],
       [0. , 1. ],
       [0.5, 0.5],
       [0. , 1. ],
       [0.5, 0.5],
       [1. , 0. ],
       [0. , 1. ],
       [0.5, 0.5],
       [0.5, 0.5],
       [1. , 0. ],
       [1. , 0. ],
       [1. , 0. ],
       [0.5, 0.5],
       [0.5, 0.5],
       [0. , 1. ]])
```

Оставим только один столбец с вероятностями единичного (истинного) класса.

In [43]:

```
true_proba = proba[:,1]
```

In [44]:

```
true_proba.shape
```

Out[44]:

```
(91,)
```

In [45]:

```
from sklearn.metrics import roc_curve, roc_auc_score
```

In [46]:

```
roc_auc_score(heart_y_test, true_proba)
```

Out[46]:

```
0.7965853658536585
```

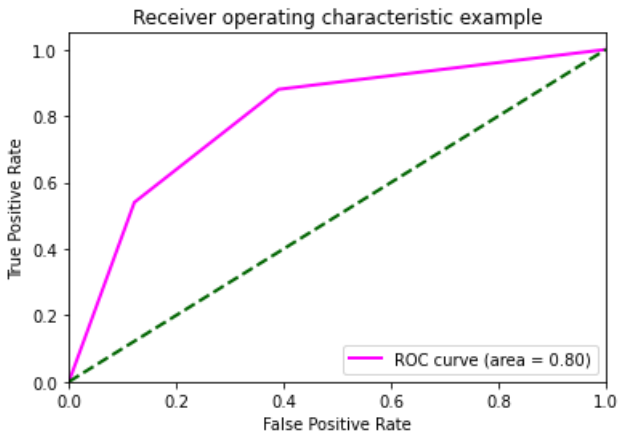
Воспользуемся написанной Юрием Евгеньевичем функцией для отрисовки ROC-кривой (https://nbviewer.jupyter.org/github/ugapanyuk/ml_course_2021/blob/main/common/notebooks/metrics/metrics.ipynb)

In [47]:

```
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='magenta',
             lw=lw, label='ROC curve (area = %0.2f)' % roc_auc_value)
    plt.plot([0, 1], [0, 1], color='darkgreen', 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 example')
    plt.legend(loc="lower right")
    plt.show()
```

In [48]:

```
draw_roc_curve(heart_y_test, true_proba, pos_label=1, average='micro')
```



Итак, соберём в одном месте метрики для нашей начальной модели с произвольно выбранным гиперпараметром k = 2.

$$ accuracy = 0.69 $$$$ precision = 0.84 $$$$ recall = 0.54 $$$$ F-measure = 0.66 $$$$ ROC-AUC = 0.8 $$

Оценка качества модели (решающее дерево)

In [49]:

```
acc, prec, rec, f1 = accuracy_score(y_predicted, tree_predicted), \
precision_score(heart_y_test,tree_predicted), \
recall_score(heart_y_test,tree_predicted), \
f1_score(heart_y_test,tree_predicted)
```

In [50]:

```
acc_uns, prec_uns, rec_uns, f1_uns = accuracy_score(y_predicted, unscaled_tree_predicted), \
precision_score(heart_y_test,unscaled_tree_predicted), \
recall_score(heart_y_test,unscaled_tree_predicted), \
f1_score(heart_y_test,unscaled_tree_predicted)
```

Сравним метрики качества на отмасштабированных и неотмасштабированных данных.

In [51]:

```
print("{:<12} {:<8} {:<8}".format('metrics','scaled','unscaled'))
print()
metrics_dict = {'accuracy':[round(acc,2), round(acc_unscaled,2)],
                'precision':[round(prec,2), round(prec_unscaled,2)],
                'recall':[round(rec,2), round(rec_unscaled,2)],
                'f1':[round(f1,2), round(f1_unscaled,2)]}
for key, value in metrics_dict.items():
    scaled, unscaled = value
    print ("{:<12} {:<8} {:<8}".format(key, scaled, unscaled))
```

```
metrics      scaled   unscaled

accuracy     0.81     0.62
precision    0.82     0.74
recall       0.64     0.46
f1           0.72     0.57
```

Подбор гиперпараметра k и кросс-валидация

In [52]:

```
from sklearn.model_selection import cross_val_score, ShuffleSplit, RepeatedKFold, LeaveOneOut
```

In [53]:

```
from sklearn.model_selection import GridSearchCV
```

Для подбора гиперпараметра k будем использовать GridSearchCV, перебирая значения от 1 до 100 с шагом = 2.

Примечание: я попробовала сначала прогнать бОльший интервал через RandomizedSearchCV, чтобы сначала получить квазиоптимальные значения, а потом уже с помощью GridSearchCV в более узком диапазоне в окрестности того значения, которое нашёл RandomizedSearchCV, искать как раз оптимальное значение параметра k. Но при каждом перезапуске ноутбука RandomizedSearchCV выдавал очень отличающиеся друг от друга значения - 8, 16, 26, 32, один раз - даже 68! Поэтому я решила отказаться от этого инструмента в пользу увеличения шага от 1 до 2 в диапазоне для и увеличить быстродействие таким образом (+ выбрала для начала простой K-Fold).

In [54]:

```
n_range = np.array(range(1,100,2))
parameters = [{'n_neighbors':n_range}]
```

Я хочу посмотреть, как с оптимальным значением гиперпараметра изменятся все те метрики, которые я измерила для выбранного наугад k, поэтому сделаем словарь со всеми пятью метриками.

In [55]:

```
scoring = {'accuracy':'accuracy',
           'precision':'precision',
           'recall':'recall',
           'F-measure':'f1',
           'AUC':'roc_auc'}
```

In [56]:

```
from sklearn.model_selection import GridSearchCV
```

Обучим 5 образцов GridSearchCV подряд, меняя у них параметр refit, который указывает на метрику, в зависимости от которой выбирается лучшее значение гиперпараметров в каждом случае.

In [57]:

```
%%time
grid_search = GridSearchCV(KNeighborsClassifier(), parameters, cv=5, refit='accuracy', scoring=scoring)

# ВНИМАНИЕ
# ВОПРОС: Юрий Евгеньевич упомянул, что поиск гиперпараметров стоит выполнять на ВСЁМ блоке данных
# при этом фолды кросс-валидации, насколько я понимаю, должны выбираться только из обучающей выборки
# тогда вопрос: что передавать в метод fit() GridSearch'а?
grid_search.fit(X, y)
```

```
Wall time: 4.26 s
```

Out[57]:

```
GridSearchCV(cv=5, estimator=KNeighborsClassifier(),
             param_grid=[{'n_neighbors': array([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31
, 33,
       35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67,
       69, 71, 73, 75, 77, 79, 81, 83, 85, 87, 89, 91, 93, 95, 97, 99])}],
             refit='accuracy',
             scoring={'AUC': 'roc_auc', 'F-measure': 'f1',
                      'accuracy': 'accuracy', 'precision': 'precision',
                      'recall': 'recall'})
```

In [58]:

```
# accuracy
grid_search.best_score_
```

Out[58]:

```
0.8383060109289617
```

In [59]:

```
grid_search.best_params_
```

Out[59]:

```
{'n_neighbors': 5}
```

In [60]:

```
grid_search.best_estimator_
```

Out[60]:

```
KNeighborsClassifier()
```

Мы получили оптимальное значение гиперпараметра k = 5!

In [61]:

```
%%time
grid_search = GridSearchCV(KNeighborsClassifier(), parameters, cv=5, refit='precision', scoring=scoring)
grid_search.fit(X, y)
```

```
Wall time: 4.5 s
```

Out[61]:

```
GridSearchCV(cv=5, estimator=KNeighborsClassifier(),
             param_grid=[{'n_neighbors': array([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31
, 33,
       35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67,
       69, 71, 73, 75, 77, 79, 81, 83, 85, 87, 89, 91, 93, 95, 97, 99])}],
             refit='precision',
             scoring={'AUC': 'roc_auc', 'F-measure': 'f1',
                      'accuracy': 'accuracy', 'precision': 'precision',
                      'recall': 'recall'})
```

In [62]:

```
# precision
grid_search.best_score_
```

Out[62]:

```
0.8243617987735634
```

В этом и последующих случаях не будем выводить оптимальный параметр, т.к. очевидно, что его значение не будет меняться.

grid_search = GridSearchCV(KNeighborsClassifier(), parameters, cv=5, refit='recall', scoring=scoring) grid_search.fit(X, y)

In [63]:

```
# recall
grid_search.best_score_
```

Out[63]:

```
0.8243617987735634
```

In [64]:

```
grid_search = GridSearchCV(KNeighborsClassifier(), parameters, cv=5, refit='F-measure', scoring=scoring)
grid_search.fit(X, y)
```

Out[64]:

```
GridSearchCV(cv=5, estimator=KNeighborsClassifier(),
             param_grid=[{'n_neighbors': array([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31
, 33,
       35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67,
       69, 71, 73, 75, 77, 79, 81, 83, 85, 87, 89, 91, 93, 95, 97, 99])}],
             refit='F-measure',
             scoring={'AUC': 'roc_auc', 'F-measure': 'f1',
                      'accuracy': 'accuracy', 'precision': 'precision',
                      'recall': 'recall'})
```

In [65]:

```
# F-мера
grid_search.best_score_
```

Out[65]:

```
0.8583165443646278
```

In [66]:

```
grid_search = GridSearchCV(KNeighborsClassifier(), parameters, cv=5, refit='AUC', scoring=scoring)
grid_search.fit(X, y)
```

Out[66]:

```
GridSearchCV(cv=5, estimator=KNeighborsClassifier(),
             param_grid=[{'n_neighbors': array([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31
, 33,
       35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67,
       69, 71, 73, 75, 77, 79, 81, 83, 85, 87, 89, 91, 93, 95, 97, 99])}],
             refit='AUC',
             scoring={'AUC': 'roc_auc', 'F-measure': 'f1',
                      'accuracy': 'accuracy', 'precision': 'precision',
                      'recall': 'recall'})
```

In [67]:

```
# ROC AUC
grid_search.best_score_
```

Out[67]:

```
0.912694404361071
```

Итак, теперь мы можем собрать в одном месте метрики для модели с отмасштабированными данными и выбранным наугад гиперпараметром k = 2. $$ accuracy = 0.69 $$ $$ precision = 0.84 $$ $$ recall = 0.54 $$ $$ F-measure = 0.66 $$ $$ ROC-AUC = 0.8 $$

И с отмасштабированными данными и оптимальным гиперпараметром k = 5.

$$ accuracy = 0.84 $$$$ precision = 0.82 $$$$ recall = 0.94 $$$$ F-measure = 0.86 $$$$ ROC-AUC = 0.91 $$

Показатели всех метрик качества (кроме precision) выросли.

И также попробуем использовать другие стратегии кросс-валидации.

In [68]:

```
# RepeatedKFold
grid_search_rep_K = GridSearchCV(KNeighborsClassifier(), parameters, cv=RepeatedKFold(n_splits=5, n_repeats=2,
grid_search_rep_K.fit(X, y)
```

Out[68]:

```
GridSearchCV(cv=RepeatedKFold(n_repeats=2, n_splits=5, random_state=42),
             estimator=KNeighborsClassifier(),
             param_grid=[{'n_neighbors': array([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31
, 33,
       35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67,
       69, 71, 73, 75, 77, 79, 81, 83, 85, 87, 89, 91, 93, 95, 97, 99])}],
             scoring='accuracy')
```

In [69]:

```
grid_search_rep_K.best_score_
```

Out[69]:

```
0.836639344262295
```

In [70]:

In [70]:
```
grid_search_rep_K.best_params_
```

Out[70]:
```
{'n_neighbors': 15}
```

Для ресурсоёмкой стратегии LeaveOneOut() зададим другой диапазон и шаг значений, чтобы сократить время выполнения.

In [71]:
```
n_range = np.array(range(4,20,4))
LOO_parameters = [{'n_neighbors':n_range}]
```

In [72]:
```
grid_search_LOO = GridSearchCV(KNeighborsClassifier(), LOO_parameters, cv=LeaveOneOut(), scoring='accuracy')
grid_search_LOO.fit(X, y)
```

Out[72]:
```
GridSearchCV(cv=LeaveOneOut(), estimator=KNeighborsClassifier(),
             param_grid=[{'n_neighbors': array([ 4,  8, 12, 16])}],
             scoring='accuracy')
```

In [73]:
```
grid_search_LOO.best_score_
```

Out[73]:
```
0.8481848184818482
```

In [74]:
```
grid_search_LOO.best_params_
```

Out[74]:
```
{'n_neighbors': 8}
```

Примечательно, что оптимальные значения гиперпараметра k, полученные с помощью стратегий LeaveOneOut() и RepeatedKFold, не совпадают ни друг с другом, ни с результатом, полученным с помощью KFold-a.

Сравнение метрик качества модели с оптимальным гиперпараметром k с отмасштабированными и неотмасштабированными данными

In [75]:
```
grid_search_unscaled = GridSearchCV(KNeighborsClassifier(), parameters, cv=5, scoring='accuracy')
grid_search_unscaled.fit(X_unscaled, y_unscaled)
```

Out[75]:
```
GridSearchCV(cv=5, estimator=KNeighborsClassifier(),
             param_grid=[{'n_neighbors': array([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31
, 33,
       35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67,
       69, 71, 73, 75, 77, 79, 81, 83, 85, 87, 89, 91, 93, 95, 97, 99])}],
             scoring='accuracy')
```

In [76]:
```
grid_search.best_params_
```

Out[76]:
```
{'n_neighbors': 43}
```

In [77]:
```
grid_search.best_score_
```

Out[77]:
```
0.912694404361071
```

А также проверим, какая модель оказалась лучшей!

In [78]:
```
n_range = np.array(range(1,100,2))
parameters = [{'max_depth':n_range}]
grid_search_tree = GridSearchCV(tree.DecisionTreeClassifier(), parameters, cv=10, scoring='accuracy')
grid_search_tree.fit(X, y)
grid_search_tree.best_params_
```

Out[78]:
```
{'max_depth': 3}
```

In [79]:
```
print('accuracy = ', grid_search_tree.best_score_)
```

```
accuracy =  0.8080645161290322
```

In [80]:
```
grid_search_tree = GridSearchCV(tree.DecisionTreeClassifier(), parameters, cv=10, scoring='precision')
grid_search_tree.fit(X, y)
print('precision = ', grid_search_tree.best_score_)
```

```
precision =  0.813143274853801
```

In [81]:

```
grid_search_tree = GridSearchCV(tree.DecisionTreeClassifier(), parameters, cv=10, scoring='recall')
grid_search_tree.fit(X, y)
print('recall = ', grid_search_tree.best_score_)
```

```
recall =  0.8540441176470587
```

In [82]:

```
n_range = np.array(range(1,100,2))
parameters = [{'max_depth':n_range}]
grid_search_tree = GridSearchCV(tree.DecisionTreeClassifier(), parameters, cv=10, scoring='f1')
grid_search_tree.fit(X, y)
print('F-measure = ', grid_search_tree.best_score_)
```

```
F-measure =  0.8221014369952501
```

In [83]:

```
n_range = np.array(range(1,100,2))
parameters = [{'max_depth':n_range}]
grid_search_tree = GridSearchCV(tree.DecisionTreeClassifier(), parameters, cv=10, scoring='roc_auc')
grid_search_tree.fit(X, y)
print('ROC AUC = ', grid_search_tree.best_score_)
```

```
ROC AUC =  0.8382322640594699
```

In [ ]: