Практическая работа реализована в среде Jupyter Notebook на языке программирования Python.

# Практическая работа №3

# КЛАССИФИКАЦИЯ ДАННЫХ НА ОСНОВЕ SVM-АЛГОРИТМА

**Вариант 18**

Вариант 19 не удалось подключить. Возникли проблемы с файлом исходных данных.

### 1) Импортируем необходимые для работы библиотеки

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn import svm
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV
```

### 2) Импортируем датасет

In [2]:
```
path = "http://archive.ics.uci.edu/ml/machine-learning-databases/spambase/spambase.data"
```

In [3]:
```
headernames = list(range(1, 58)) + ['Class']
```

In [4]:
```
dataset = pd.read_csv(path, names=headernames)
dataset
```

Out[4]:

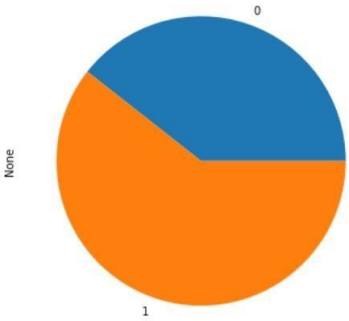
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | ... | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | Class |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0.00 | 0.64 | 0.64 | 0.0 | 0.32 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | ... | 0.000 | 0.000 | 0.0 | 0.778 | 0.000 | 0.000 | 3.756 | 61 | 278 | 1 |
| **1** | 0.21 | 0.28 | 0.50 | 0.0 | 0.14 | 0.28 | 0.21 | 0.07 | 0.00 | 0.94 | ... | 0.000 | 0.132 | 0.0 | 0.372 | 0.180 | 0.048 | 5.114 | 101 | 1028 | 1 |
| **2** | 0.06 | 0.00 | 0.71 | 0.0 | 1.23 | 0.19 | 0.19 | 0.12 | 0.64 | 0.25 | ... | 0.010 | 0.143 | 0.0 | 0.276 | 0.184 | 0.010 | 9.821 | 485 | 2259 | 1 |
| **3** | 0.00 | 0.00 | 0.00 | 0.0 | 0.63 | 0.00 | 0.31 | 0.63 | 0.31 | 0.63 | ... | 0.000 | 0.137 | 0.0 | 0.137 | 0.000 | 0.000 | 3.537 | 40 | 191 | 1 |
| **4** | 0.00 | 0.00 | 0.00 | 0.0 | 0.63 | 0.00 | 0.31 | 0.63 | 0.31 | 0.63 | ... | 0.000 | 0.135 | 0.0 | 0.135 | 0.000 | 0.000 | 3.537 | 40 | 191 | 1 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **4596** | 0.31 | 0.00 | 0.62 | 0.0 | 0.00 | 0.31 | 0.00 | 0.00 | 0.00 | 0.00 | ... | 0.000 | 0.232 | 0.0 | 0.000 | 0.000 | 0.000 | 1.142 | 3 | 88 | 0 |
| **4597** | 0.00 | 0.00 | 0.00 | 0.0 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | ... | 0.000 | 0.000 | 0.0 | 0.353 | 0.000 | 0.000 | 1.555 | 4 | 14 | 0 |
| **4598** | 0.30 | 0.00 | 0.30 | 0.0 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | ... | 0.102 | 0.718 | 0.0 | 0.000 | 0.000 | 0.000 | 1.404 | 6 | 118 | 0 |
| **4599** | 0.96 | 0.00 | 0.00 | 0.0 | 0.32 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | ... | 0.000 | 0.057 | 0.0 | 0.000 | 0.000 | 0.000 | 1.147 | 5 | 78 | 0 |
| **4600** | 0.00 | 0.00 | 0.65 | 0.0 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | ... | 0.000 | 0.000 | 0.0 | 0.125 | 0.000 | 0.000 | 1.250 | 5 | 40 | 0 |

4601 rows × 58 columns

### 3) Первичный анализ данных

In [5]:
```
ps = pd.Series([dataset.loc[dataset['Class'] == 1].Class.count(), dataset.loc[dataset['Class'] == 0].Class.count()])
ps.plot.pie(figsize=(6, 6))
```

Out[5]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x228aa793308>
```




In [6]:
```
print(ps)
```

```
0    1813
1    2788
dtype: int64
```

Как мы можем заметить распределение классов в датасете неравномерное, но не критическое. Теперь посмотрим, от каких коэффициентов больше всего зависит класс объекта. Определим корреляцию для каждого из параметров.

In [7]:
```
dataset.corr()['Class'][:-1]
```

Out[7]:
```
1     0.126208
2    -0.030224
3     0.196988
4     0.057371
5     0.241920
6     0.232604
7     0.332117
8     0.206808
9     0.231551
10    0.138962
11    0.234529
12    0.007741
13    0.132927
14    0.060027
15    0.195902
16    0.263215
17    0.263204
18    0.204208
19    0.273651
20    0.189761
21    0.383234
22    0.091860
23    0.334787
24    0.216111
25   -0.256723
26   -0.232968
27   -0.183404
28   -0.158800
29   -0.133523
30   -0.171095
31   -0.126912
32   -0.114214
33   -0.119931
34   -0.112754
35   -0.149225
36   -0.136134
37   -0.178045
38   -0.031035
39   -0.122831
40   -0.064801
41   -0.097375
42   -0.136615
43   -0.135664
44   -0.094594
45   -0.140408
46   -0.146138
```

```
47   -0.044679
48   -0.084020
49   -0.059630
50   -0.089672
51   -0.064709
52    0.241888
53    0.323629
54    0.065067
55    0.109999
56    0.216097
57    0.249164
Name: Class, dtype: float64
```

Как мы можем заметить, нет параметров, которые сильно коррелируют со значением класса объекта.

### 4) Подготовка данных для обучения модели

Отделяем параметры от значения класса

In [8]:
```
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, -1].values
```

Теперь разбиваем выборку на тренировочную и тестовую в соотношении 60/40

In [9]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.4)
```

Регулируем масштаб значений параметров так, чтобы каждый параметр имел одинаковый вес.

In [10]:
```
scaler = StandardScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

### 5) Обучение SVM модели

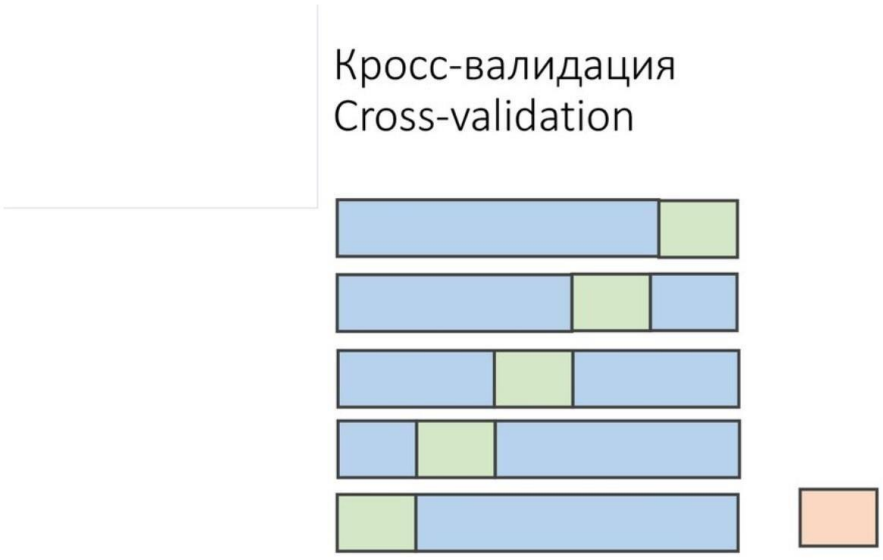
In [11]:
```
clf = svm.SVC(C=1, kernel='linear')
clf.fit(X_train, y_train)
```

Out[11]:
```
SVC(C=1, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='scale', kernel='linear',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

In [12]:
```
clf.score(X_test, y_test)
```

Out[12]:
```
0.928843020097773
```

**6) Улучшаем результат. Подбираем оптимальное ядро и параметр регуляризации.**




Для поиска оптимальных параметров будем использовать кросвалидацию. Для этого разделм тренировочную выборку на 3 части: 2 - тренировочные и 1 - валидационная. Причём каждая из часей будет использована и как тренировочная, и как валидационная.

In [14]:
```
svc = svm.SVC()
param = {'kernel': ['linear', 'poly', 'rbf', 'sigmoid'], 'C': [i * 0.1 for i in range(1, 16)]}
gscv =  GridSearchCV(svc, param, cv=3, n_jobs=-1, verbose=1)
gscv.fit(X_train, y_train)
```

```
Fitting 3 folds for each of 60 candidates, totalling 180 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  42 tasks      | elapsed:   21.1s
[Parallel(n_jobs=-1)]: Done 180 out of 180 | elapsed:  1.2min finished
```

Out[14]:
```
GridSearchCV(cv=3, error_score=nan,
             estimator=SVC(C=1.0, break_ties=False, cache_size=200,
                           class_weight=None, coef0=0.0,
                           decision_function_shape='ovr', degree=3,
                           gamma='scale', kernel='rbf', max_iter=-1,
                           probability=False, random_state=None, shrinking=True,
                           tol=0.001, verbose=False),
             iid='deprecated', n_jobs=-1,
             param_grid={'C': [0.1, 0.2, 0.30000000000000004, 0.4, 0.5,
                               0.6000000000000001, 0.7000000000000001, 0.8, 0.9,
                               1.0, 1.1, 1.2000000000000002, 1.3,
                               1.4000000000000001, 1.5],
                         'kernel': ['linear', 'poly', 'rbf', 'sigmoid']},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=1)
```

In [16]:
```
gscv.best_params_
```

Out[16]:
```
{'C': 1.2000000000000002, 'kernel': 'linear'}
```

## 7) Итоговая оценка результата обучения

In [17]:
```
best_c = gscv.best_estimator_
```

In [18]:
```
y_pred = best_c.predict(X_test)
```

In [19]:
```
result = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(result)
result1 = classification_report(y_test, y_pred)
print("Classification Report:",)
print (result1)
result2 = accuracy_score(y_test,y_pred)
print("Accuracy:",result2)
```

```
Confusion Matrix:
[[1047   56]
 [  74  664]]
Classification Report:
              precision    recall  f1-score   support

           0       0.93      0.95      0.94      1103
           1       0.92      0.90      0.91       738

    accuracy                           0.93      1841
   macro avg       0.93      0.92      0.93      1841
weighted avg       0.93      0.93      0.93      1841

Accuracy: 0.9293862031504617
```

**Итоговая точность: 92,94%**

Вывод:

Алгоритм SVM может эффективно справляться с задачей классификации. Очень важно подбирать правильные параметры для работы с этим алгоритмом и проводить качественную предобработку данных перед обучением. Это напрямую влияет на результативность полученной модели.

# КЛАССИФИКАЦИЯ ДАННЫХ НА ОСНОВЕ KNN-АЛГОРИТМА

Налало такое же как и в 3 практической работе. Импортируем библиотеки и подготовим данные для обучения.

In [1]:
```
import numpy as np
import pandas as pd
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
```

In [2]:
```
path = "http://archive.ics.uci.edu/ml/machine-learning-databases/spambase/spambase.data"
headernames = list(range(1, 58)) + ['Class']
dataset = pd.read_csv(path, names=headernames)
dataset
```

Out[2]:

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | ... | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | Class |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0.00 | 0.64 | 0.64 | 0.0 | 0.32 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | ... | 0.000 | 0.000 | 0.0 | 0.778 | 0.000 | 0.000 | 3.756 | 61 | 278 | 1 |
| **1** | 0.21 | 0.28 | 0.50 | 0.0 | 0.14 | 0.28 | 0.21 | 0.07 | 0.00 | 0.94 | ... | 0.000 | 0.132 | 0.0 | 0.372 | 0.180 | 0.048 | 5.114 | 101 | 1028 | 1 |
| **2** | 0.06 | 0.00 | 0.71 | 0.0 | 1.23 | 0.19 | 0.19 | 0.12 | 0.64 | 0.25 | ... | 0.010 | 0.143 | 0.0 | 0.276 | 0.184 | 0.010 | 9.821 | 485 | 2259 | 1 |
| **3** | 0.00 | 0.00 | 0.00 | 0.0 | 0.63 | 0.00 | 0.31 | 0.63 | 0.31 | 0.63 | ... | 0.000 | 0.137 | 0.0 | 0.137 | 0.000 | 0.000 | 3.537 | 40 | 191 | 1 |
| **4** | 0.00 | 0.00 | 0.00 | 0.0 | 0.63 | 0.00 | 0.31 | 0.63 | 0.31 | 0.63 | ... | 0.000 | 0.135 | 0.0 | 0.135 | 0.000 | 0.000 | 3.537 | 40 | 191 | 1 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **4596** | 0.31 | 0.00 | 0.62 | 0.0 | 0.00 | 0.31 | 0.00 | 0.00 | 0.00 | 0.00 | ... | 0.000 | 0.232 | 0.0 | 0.000 | 0.000 | 0.000 | 1.142 | 3 | 88 | 0 |
| **4597** | 0.00 | 0.00 | 0.00 | 0.0 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | ... | 0.000 | 0.000 | 0.0 | 0.353 | 0.000 | 0.000 | 1.555 | 4 | 14 | 0 |
| **4598** | 0.30 | 0.00 | 0.30 | 0.0 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | ... | 0.102 | 0.718 | 0.0 | 0.000 | 0.000 | 0.000 | 1.404 | 6 | 118 | 0 |
| **4599** | 0.96 | 0.00 | 0.00 | 0.0 | 0.32 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | ... | 0.000 | 0.057 | 0.0 | 0.000 | 0.000 | 0.000 | 1.147 | 5 | 78 | 0 |
| **4600** | 0.00 | 0.00 | 0.65 | 0.0 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | ... | 0.000 | 0.000 | 0.0 | 0.125 | 0.000 | 0.000 | 1.250 | 5 | 40 | 0 |

4601 rows × 58 columns

In [3]:
```
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, -1].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.4)
```

Проведём первые тесты KNN классификатора на исходных данных.

In [4]:
```
clf1 = KNeighborsClassifier()
clf1.fit(X_train, y_train)
clf1.score(X_test, y_test)
```

Out[4]:
```
0.7935904399782727
```

Правильное регулирование масштбов значений данных повышает эффективность работы модели. Особенно это ощущается при использовании алгоритма K ближайших соседей.

In [5]:
```
scaler = StandardScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

In [6]:
```
clf2 = KNeighborsClassifier()
clf2.fit(X_train, y_train)
clf2.score(X_test, y_test)
```

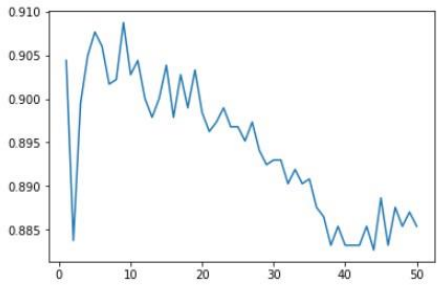
Out[6]:
```
0.9076588810429115
```

Начнём подбор более оптимальных параметров. Начнём с подбора числа соседей. Я написал свою функцию подбора определённого параметра модели машинного обучения.

In [7]:
```
def search_param(model, param, X_train, y_train, X_val, y_val, area=range(1, 11), msg=True, plot=True):
    import matplotlib.pyplot as plt
    import time
    score_list = []
    if msg:
        print('#     accuracy  time')
    for i in area:
        start = time.time()
        if str(type(i)) == "<class 'str'>":
            mod = eval(model + "(" + param + "='" + i + "')")
        else:
            mod = eval(model + '(' + param + '=' + str(i) + ')')
        mod.fit(X_train, y_train)
        s = mod.score(X_val, y_val)
        end = time.time()
        score_list.append(s)
        if msg:
            print("%3s %10f  %7f" % (str(i), s, end - start))
    if plot:
        plt.plot(list(area), score_list)
    return list(area)[score_list.index(max(score_list))]
```

In [8]:
```
search_param('KNeighborsClassifier', 'n_neighbors', X_train, y_train, X_test, y_test, area=range(1, 51), msg=False)
```

Out[8]: 9




Как мы можем убедиться,при использовании классической метрики увеличение числа соседей приводит к понижению эффективности.

Подберём оптимальную метрку для данного классификатора. Будем использовать всё ту же функцию *search_param*.

In [9]:
```
search_param('KNeighborsClassifier', 'metric', X_train, y_train, X_test, y_test, area=['euclidean', 'manhattan', 'chebyshe
```

```
#     accuracy  time
euclidean   0.907659  1.025857
manhattan   0.912548  0.951035
chebyshev   0.875068  1.069546
minkowski   0.907659  1.019098
```

Out[9]: 'manhattan'

Использование манхэтенской метрики является наиболее оптимальным решением. Точность повсилась на 0,5% по сравнению с классической евклидовой метрикой.

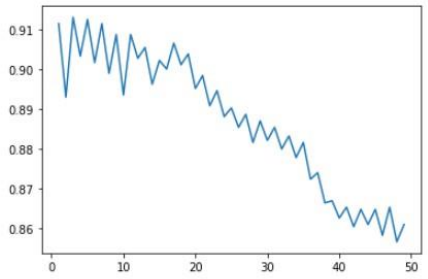
In [10]:
```
clf3 = KNeighborsClassifier(n_neighbors=3, metric = 'manhattan')
clf3.fit(X_train, y_train)
clf3.score(X_test, y_test)
```

Out[10]: 0.913090711569799

Подберём оптимальное число соседий для найденной оптимальной метрики.

In [11]:
```
def search_param2(model, param, X_train, y_train, X_val, y_val, area=range(1, 11), msg=True, plot=True):
    import matplotlib.pyplot as plt
    import time
    score_list = []
    if msg:
        print('#     accuracy  time')
    for i in area:
        start = time.time()
        if str(type(i)) == "<class 'str'>":
            mod = eval("KNeighborsClassifier(metric = 'manhattan', " + param + "='" + i + "')")
        else:
            mod = eval("KNeighborsClassifier(metric = 'manhattan', " + param + '=' + str(i) + ')')
        mod.fit(X_train, y_train)
        s = mod.score(X_val, y_val)
        end = time.time()
        score_list.append(s)
        if msg:
            print("%3s %10f  %7f" % (str(i), s, end - start))
    if plot:
        plt.plot(list(area), score_list)
    return list(area)[score_list.index(max(score_list))]
```

In [12]:
```
k = search_param2('KNeighborsClassifier', 'n_neighbors', X_train, y_train, X_test, y_test, area=range(1, 50), plot=True, m
```




In [13]:
```
clf4 = KNeighborsClassifier(n_neighbors=k, metric = 'manhattan')
clf4.fit(X_train, y_train)
clf4.score(X_test, y_test)
```

Out[13]:
```
0.913090711569799
```

Итоговый результат:

In [14]:
```
y_pred = clf4.predict(X_test)
result = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(result)
result1 = classification_report(y_test, y_pred)
print("Classification Report:",)
print (result1)
result2 = accuracy_score(y_test,y_pred)
print("Accuracy:",result2)
```

```
Confusion Matrix:
[[1041   62]
 [  98  640]]
Classification Report:
              precision    recall  f1-score   support

           0       0.91      0.94      0.93      1103
           1       0.91      0.87      0.89       738

    accuracy                           0.91      1841
   macro avg       0.91      0.91      0.91      1841
weighted avg       0.91      0.91      0.91      1841

Accuracy: 0.913090711569799
```

## Для ирисов Фишера

In [15]:
```
headernames = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'Class']
dataset = pd.read_csv(path, names = headernames)
```

In [16]:
```
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, 4].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.40)
scaler = StandardScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

In [17]:
```
classifier = KNeighborsClassifier(n_neighbors = k, metric = 'manhattan')
classifier.fit(X_train, y_train)
```

Out[17]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='manhattan',
                     metric_params=None, n_jobs=None, n_neighbors=3, p=2,
                     weights='uniform')
```

In [18]:
```
y_pred = classifier.predict(X_test)
result = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(result)
result1 = classification_report(y_test, y_pred)
print("Classification Report:",)
print (result1)
result2 = accuracy_score(y_test,y_pred)
print("Accuracy:",result2)
```

```
Confusion Matrix:
[[916 190]
 [226 509]]
Classification Report:
              precision    recall  f1-score   support

           0       0.80      0.83      0.81      1106
           1       0.73      0.69      0.71       735

    accuracy                           0.77      1841
   macro avg       0.77      0.76      0.76      1841
weighted avg       0.77      0.77      0.77      1841

Accuracy: 0.7740358500814775
```

Полученный результат показывает нам,что для разных денных необходимо побирать уникальные параметры модели. Эффективная модель для одних данных может быть совершенно неэффективна для других.