Лабораторная работа №4 по курсу "Технологии машинного обучения"

Выполнила Попова Дарья, студентка группы РТ5-61Б

Линейные модели, SVM и деревья решений

Задача бинарной классификации

Для задачи бинарной классификации я выбрала датасет: https://www.kaggle.com/ruslankl/mice-protein-expression

Датасет состоит из 77 колонок с уровнями выделения различных белков у мышей, которые разделены на 2 группы: контрольную и трисомическую.

В описании датасета упомянуто, что для измерений использовали 38 мышей в контрольной группе и 34 мыши в трисомической (таким образом, всего 72 мыши). Однако сказано, что каждую строку можно рассматривать как отдельный самостоятельный образец. Мы так и поступим.

In [1]:

```
import pandas as pd
import numpy as np
df = pd.read_csv('C:\\Users\\Дасупс\\Downloads\\Data_Cortex_Nuclear.csv')
```

In [2]:

```
df.head()
```

Out[2]:

	MouseID	DYRK1A_N	ITSN1_N	BDNF_N	NR1_N	NR2A_N	pAKT_N	pBRAF_N	pCAMKII_N	pCREB_N	...	pCFOS_N	SYP_N	H3AcK18_N	EG
0	309_1	0.503644	0.747193	0.430175	2.816329	5.990152	0.218830	0.177565	2.373744	0.232224	...	0.108336	0.427099	0.114783	0.1
1	309_2	0.514617	0.689064	0.411770	2.789514	5.685038	0.211636	0.172817	2.292150	0.226972	...	0.104315	0.441581	0.111974	0.1
2	309_3	0.509183	0.730247	0.418309	2.687201	5.622059	0.209011	0.175722	2.283337	0.230247	...	0.106219	0.435777	0.111883	0.1
3	309_4	0.442107	0.617076	0.358626	2.466947	4.979503	0.222886	0.176463	2.152301	0.207004	...	0.111262	0.391691	0.130405	0.1
4	309_5	0.434940	0.617430	0.358802	2.365785	4.718679	0.213106	0.173627	2.134014	0.192158	...	0.110694	0.434154	0.118481	0.1

5 rows × 82 columns

In [3]:

```
df.shape
```

Out[3]:

```
(1080, 82)
```

In [4]:

```
df['MouseID'].nunique()
```

Out[4]:

```
1080
```

In [5]:

```
df['Genotype'].unique()
```

Out[5]:

```
array(['Control', 'Ts65Dn'], dtype=object)
```

In [6]:

```
df['class'].unique()
```

Out[6]:

```
array(['c-CS-m', 'c-SC-m', 'c-CS-s', 'c-SC-s', 't-CS-m', 't-SC-m',
       't-CS-s', 't-SC-s'], dtype=object)
```

Как можно заметить, классификация изначально многоклассовая. Класс состоит из трёх параметров и формируется по следующему принципу:

- c/t - control/trisomic - мышь из контрольной группы или с трисомией (синдромом Дауна)
- CS/SC (control shock/shock control) - поведенческий показатель, отображающий способность мыши к обучению
- m/s (memantine/saline) - некоторым мышкам вводили препарат мемантин для стимуляции способности к обучению, а некоторым - физраствор (saline).

Но стоит обратить внимание на то, что параметр CS/SC формируется полностью на основе столбца Behavior, а m/s - на основе столбца Treatment. В свою очередь то, относится мышь к контрольной группе или к группе с трисомией, определяет признак Genotype.

Я удалю столбцы Genotype, Treatment и Behavior и сделаю целевой признак бинарным. Задача, таким образом, будет сводиться к задаче бинарной классификации и будет состоять в предсказании наличия у мыши трисомии (1 - есть, 0 - нет) на основании 77 колонок с показателями выделения белков корой головного мозга.

In [7]:

```
df.loc[df['class'] == 'c-CS-m'].Behavior.unique()
```

Out[7]:

```
array(['C/S'], dtype=object)
```

In [8]:

```
df.loc[df['class'] == 'c-CS-m'].Treatment.unique()
```

Out[8]:

```
array(['Memantine'], dtype=object)
```

Предобработка данных

Перекодируем столбец с целевым признаком и заменим все образцы контрольной группы нулями, а трихомической - единицами.

In [9]:

```
df['class'] = df['class'].replace(['c-CS-m', 'c-SC-m', 'c-CS-s', 'c-SC-s'], 0)
```

In [10]:

```
df['class'] = df['class'].replace(['t-CS-m', 't-SC-m', 't-CS-s', 't-SC-s'], 1)
```

In [11]:

```
df
```

Out[11]:

	MouseID	DYRK1A_N	ITSN1_N	BDNF_N	NR1_N	NR2A_N	pAKT_N	pBRAF_N	pCAMKII_N	pCREB_N	...	pCFOS_N	SYP_N	H3AcK18_N
0	309_1	0.503644	0.747193	0.430175	2.816329	5.990152	0.218830	0.177565	2.373744	0.232224	...	0.108336	0.427099	0.114783
1	309_2	0.514617	0.689064	0.411770	2.789514	5.685038	0.211636	0.172817	2.292150	0.226972	...	0.104315	0.441581	0.111974
2	309_3	0.509183	0.730247	0.418309	2.687201	5.622059	0.209011	0.175722	2.283337	0.230247	...	0.106219	0.435777	0.111883
3	309_4	0.442107	0.617076	0.358626	2.466947	4.979503	0.222886	0.176463	2.152301	0.207004	...	0.111262	0.391691	0.130405
4	309_5	0.434940	0.617430	0.358802	2.365785	4.718679	0.213106	0.173627	2.134014	0.192158	...	0.110694	0.434154	0.118481
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1075	J3295_11	0.254860	0.463591	0.254860	2.092082	2.600035	0.211736	0.171262	2.483740	0.207317	...	0.183324	0.374088	0.318782
1076	J3295_12	0.272198	0.474163	0.251638	2.161390	2.801492	0.251274	0.182496	2.512737	0.216339	...	0.175674	0.375259	0.325639
1077	J3295_13	0.228700	0.395179	0.234118	1.733184	2.220852	0.220665	0.161435	1.989723	0.185164	...	0.158296	0.422121	0.321306
1078	J3295_14	0.221242	0.412894	0.243974	1.876347	2.384088	0.208897	0.173623	2.086028	0.192044	...	0.196296	0.397676	0.335936
1079	J3295_15	0.302626	0.461059	0.256564	2.092790	2.594348	0.251001	0.191811	2.361816	0.223632	...	0.187556	0.420347	0.335062

1080 rows × 82 columns

In [12]:

```
# посмотрим, сколько образцов каждого класса содержится в наборе данных
labels, counters = np.unique(df['class'], return_counts = True)
labels = labels.tolist()
counters = counters.tolist()
for i in range(df['class'].nunique()):
    print('Количество образцов класса {} = {} ({}%)'.format(
    labels[i], counters[i], round(100 * counters[i] / df.shape[0], 2)))
```

```
Количество образцов класса 0 = 570 (52.78%)
Количество образцов класса 1 = 510 (47.22%)
```

Классы поделены почти пополам, поэтому нам не придётся сталкиваться с негативными последствиями несбалансированности исходной выборки.

Разделим данные на независимые фичи и целевой признак. Для столбцов-предсказателей к тому же удалим столбец с ID каждой мыши, а также столбцы Genotype, Treatment и Behavior.

In [13]:

```
X = df.drop(['MouseID', 'Genotype', 'Treatment', 'Behavior', 'class'], axis=1)
```

In [14]:

```
y = df['class']
```

Обработка пропусков

In [15]:

```
# убедимся, что в целевой функции у нас нет пропусков
df['class'].isnull().any()
```

Out[15]:

```
False
```

In [16]:

```
# сначала убедимся, что все 77 фичей являются числовыми
num = 0
for column in X.columns:
    if X[column].dtype == 'float64' or X[column].dtype == 'int':
        num += 1
num
```

Out[16]:

```
77
```

In [17]:

```
i = 1
for column in X.columns:
    null_count = X[column].isnull().sum()
    if null_count == 0:
        print(f'{i}. В колонке {column} нет пропусков.')
        i += 1
```

```
1. В колонке NUMB_N нет пропусков.
2. В колонке P70S6_N нет пропусков.
3. В колонке pGSK3B_N нет пропусков.
4. В колонке pPKCG_N нет пропусков.
5. В колонке CDK5_N нет пропусков.
6. В колонке S6_N нет пропусков.
7. В колонке ADARB1_N нет пропусков.
8. В колонке AcetylH3K9_N нет пропусков.
9. В колонке RRP1_N нет пропусков.
10. В колонке BAX_N нет пропусков.
11. В колонке ARC_N нет пропусков.
12. В колонке ERBB4_N нет пропусков.
13. В колонке nNOS_N нет пропусков.
14. В колонке Tau_N нет пропусков.
15. В колонке GFAP_N нет пропусков.
16. В колонке GluR3_N нет пропусков.
17. В колонке GluR4_N нет пропусков.
18. В колонке IL1B_N нет пропусков.
19. В колонке P3525_N нет пропусков.
20. В колонке pCASP9_N нет пропусков.
21. В колонке PSD95_N нет пропусков.
22. В колонке SNCA_N нет пропусков.
23. В колонке Ubiquitin_N нет пропусков.
24. В колонке pGSK3B_Tyr216_N нет пропусков.
25. В колонке SHH_N нет пропусков.
26. В колонке pS6_N нет пропусков.
27. В колонке SYP_N нет пропусков.
28. В колонке CaNA_N нет пропусков.
```

In [18]:

```
i = 1
for column in X.columns:
    null_count = X[column].isnull().sum()
    if null_count > 0:
        print(f'{i}. В колонке {column} {null_count} пропусков, {round((null_count / X.shape[0]) * 100.0, 2)}%
        i += 1
```

```
1. В колонке DYRK1A_N 3 пропусков, 0.28%.
2. В колонке ITSN1_N 3 пропусков, 0.28%.
3. В колонке BDNF_N 3 пропусков, 0.28%.
4. В колонке NR1_N 3 пропусков, 0.28%.
5. В колонке NR2A_N 3 пропусков, 0.28%.
6. В колонке pAKT_N 3 пропусков, 0.28%.
7. В колонке pBRAF_N 3 пропусков, 0.28%.
8. В колонке pCAMKII_N 3 пропусков, 0.28%.
9. В колонке pCREB_N 3 пропусков, 0.28%.
10. В колонке pELK_N 3 пропусков, 0.28%.
11. В колонке pERK_N 3 пропусков, 0.28%.
12. В колонке pJNK_N 3 пропусков, 0.28%.
13. В колонке PKCA_N 3 пропусков, 0.28%.
14. В колонке pMEK_N 3 пропусков, 0.28%.
15. В колонке pNR1_N 3 пропусков, 0.28%.
16. В колонке pNR2A_N 3 пропусков, 0.28%.
17. В колонке pNR2B_N 3 пропусков, 0.28%.
18. В колонке pPKCAB_N 3 пропусков, 0.28%.
19. В колонке pRSK_N 3 пропусков, 0.28%.
20. В колонке AKT_N 3 пропусков, 0.28%.
21. В колонке BRAF_N 3 пропусков, 0.28%.
22. В колонке CAMKII_N 3 пропусков, 0.28%.
23. В колонке CREB_N 3 пропусков, 0.28%.
24. В колонке ELK_N 18 пропусков, 1.67%.
25. В колонке ERK_N 3 пропусков, 0.28%.
26. В колонке GSK3B_N 3 пропусков, 0.28%.
27. В колонке JNK_N 3 пропусков, 0.28%.
28. В колонке MEK_N 7 пропусков, 0.65%.
29. В колонке TRKA_N 3 пропусков, 0.28%.
30. В колонке RSK_N 3 пропусков, 0.28%.
31. В колонке APP_N 3 пропусков, 0.28%.
32. В колонке Bcatenin_N 18 пропусков, 1.67%.
33. В колонке SOD1_N 3 пропусков, 0.28%.
34. В колонке MTOR_N 3 пропусков, 0.28%.
35. В колонке P38_N 3 пропусков, 0.28%.
36. В колонке pMTOR_N 3 пропусков, 0.28%.
37. В колонке DSCR1_N 3 пропусков, 0.28%.
38. В колонке AMPKA_N 3 пропусков, 0.28%.
39. В колонке NR2B_N 3 пропусков, 0.28%.
40. В колонке pNUMB_N 3 пропусков, 0.28%.
41. В колонке RAPTOR_N 3 пропусков, 0.28%.
42. В колонке TIAM1_N 3 пропусков, 0.28%.
43. В колонке pP70S6_N 3 пропусков, 0.28%.
44. В колонке BAD_N 213 пропусков, 19.72%.
45. В колонке BCL2_N 285 пропусков, 26.39%.
46. В колонке pCFOS_N 75 пропусков, 6.94%.
47. В колонке H3AcK18_N 180 пропусков, 16.67%.
48. В колонке EGR1_N 210 пропусков, 19.44%.
49. В колонке H3MeK4_N 270 пропусков, 25.0%.
```

Как можно заметить, в большинстве столбцов пропусков меньше одного процента. В некоторых колонках процент пропущенных значений достигает 25-26%, но это не так критично, поэтому оставим все признаки.

In [19]:

```
from sklearn.impute import SimpleImputer
```

In [20]:

```
# заполним пропуски во всех колонках
for column in X.columns:
    null_count = X[column].isnull().sum()
    if null_count > 0:
        imputer = SimpleImputer(missing_values=np.nan, strategy='median')
        X[column] = imputer.fit_transform(X[[column]])
```

In [21]:

```
# убедимся, что пропусков не осталось
null_sum = 0
for column in X.columns:
    null_count = X[column].isnull().sum()
    null_sum += null_count
null_sum
```

Out[21]:

```
0
```

Масштабирование признаков

Проверим, нужно ли будет масштабировать признаки.

In [22]:

```
for column in X.columns:
    print(f'В колонке {column} данные распределены от {X[column].min()} до {X[column].max()}')
```

```
В колонке DYRK1A_N данные распределены от 0.145326504 до 2.516367377
В колонке ITSN1_N данные распределены от 0.245358515 до 2.602662135
В колонке BDNF_N данные распределены от 0.11518140199999999 до 0.497159859
В колонке NR1_N данные распределены от 1.330830671 до 3.7576413310000003
В колонке NR2A_N данные распределены от 1.737539936 до 8.482553422
В колонке pAKT_N данные распределены от 0.063236006 до 0.5390501320000001
В колонке pBRAF_N данные распределены от 0.064042588 до 0.317065589
В колонке pCAMKII_N данные распределены от 1.343998185 до 7.4640702139999995
В колонке pCREB_N данные распределены от 0.112811791 до 0.306247231
В колонке pELK_N данные распределены от 0.42903225799999994 до 6.113347458
В колонке pERK_N данные распределены от 0.149155227 до 3.566685372
В колонке pJNK_N данные распределены от 0.05211039 до 0.493425858
В колонке PKCA_N данные распределены от 0.191430693 до 0.473992025
В колонке pMEK_N данные распределены от 0.056818182 до 0.45800055100000003
В колонке pNR1_N данные распределены от 0.500159744 до 1.4081687859999998
В колонке pNR2A_N данные распределены от 0.281284812 до 1.412750248
В колонке pNR2B_N данные распределены от 0.301608579 до 2.723965377
В колонке pPKCAB_N данные распределены от 0.567840497 до 3.0613871460000004
В колонке pRSK_N данные распределены от 0.09594155800000001 до 0.6509618070000001
В колонке AKT_N данные распределены от 0.06442119 до 1.182174736
В колонке BRAF_N данные распределены от 0.143893591 до 2.133415747
В колонке CAMKII_N данные распределены от 0.212959525 до 0.586244541
В колонке CREB_N данные распределены от 0.113636364 до 0.319558247
В колонке ELK_N данные распределены от 0.49769500299999997 до 2.802948336
В колонке ERK_N данные распределены от 1.131795717 до 5.198404111
В колонке GSK3B_N данные распределены от 0.151124339 до 2.475751236
В колонке JNK_N данные распределены от 0.04629779 до 0.387190684
В колонке MEK_N данные распределены от 0.14720149300000002 до 0.415407855
В колонке TRKA_N данные распределены от 0.19874338600000002 до 1.001622938
В колонке RSK_N данные распределены от 0.10739436599999999 до 0.305135952
В колонке APP_N данные распределены от 0.23559539100000002 до 0.6326627220000001
В колонке Bcatenin_N данные распределены от 1.134886146 до 3.680551799
В колонке SOD1_N данные распределены от 0.21712018100000002 до 1.872898533
В колонке MTOR_N данные распределены от 0.20114336 до 0.6767479670000001
В колонке P38_N данные распределены от 0.227880387 до 0.933256284
В колонке pMTOR_N данные распределены от 0.166578716 до 1.124883359
В колонке DSCR1_N данные распределены от 0.15532102 до 0.916429495
В колонке AMPKA_N данные распределены от 0.22640869 до 0.700838518
В колонке NR2B_N данные распределены от 0.184784521 до 0.9720197509999999
В колонке pNUMB_N данные распределены от 0.185597624 до 0.631052204
В колонке RAPTOR_N данные распределены от 0.194824478 до 0.526681402
В колонке TIAM1_N данные распределены от 0.237777071 до 0.722121604
В колонке pP70S6_N данные распределены от 0.13111979199999999 до 1.1291714609999999
В колонке NUMB_N данные распределены от 0.117998506 до 0.31657534800000003
В колонке P70S6_N данные распределены от 0.344119778 до 1.6799531559999998
В колонке pGSK3B_N данные распределены от 0.09997585099999999 до 0.253210088
В колонке pPKCG_N данные распределены от 0.5987665520000001 до 3.381976285
В колонке CDK5_N данные распределены от 0.181157025 до 0.8174018129999999
В колонке S6_N данные распределены от 0.130206306 до 0.8226108
В колонке ADARB1_N данные распределены от 0.529107819 до 2.539889642
В колонке AcetylH3K9_N данные распределены от 0.05252842 до 1.459386852
В колонке RRP1_N данные распределены от -0.062007874000000004 до 0.612377029
В колонке BAX_N данные распределены от 0.072325525 до 0.241141052
В колонке ARC_N данные распределены от 0.067254286 до 0.158747815
В колонке ERBB4_N данные распределены от 0.100217339 до 0.208697506
В колонке nNOS_N данные распределены от 0.099734364 до 0.26073863399999997
В колонке Tau_N данные распределены от 0.09623279400000001 до 0.602768056
В колонке GFAP_N данные распределены от 0.08611418 до 0.213620644
В колонке GluR3_N данные распределены от 0.111382114 до 0.331015923
В колонке GluR4_N данные распределены от 0.072579682 до 0.537004104
В колонке IL1B_N данные распределены от 0.284001296 до 0.889735099
В колонке P3525_N данные распределены от 0.207437817 до 0.443735036
В колонке pCASP9_N данные распределены от 0.8531755609999999 до 2.586215947
В колонке PSD95_N данные распределены от 1.206097755 до 2.877873418
В колонке SNCA_N данные распределены от 0.10123315199999999 до 0.257615894
В колонке Ubiquitin_N данные распределены от 0.750664091 до 1.897202342
В колонке pGSK3B_Tyr216_N данные распределены от 0.577396764 до 1.2045980809999999
В колонке SHH_N данные распределены от 0.15586929300000002 до 0.35828877
В колонке BAD_N данные распределены от 0.088304624 до 0.282016349
В колонке BCL2_N данные распределены от 0.08065684599999999 до 0.26150572
В колонке pS6_N данные распределены от 0.067254286 до 0.158747815
В колонке pCFOS_N данные распределены от 0.08541015 до 0.256528926
```

```
В колонке pCR0S_N данные распределены от 0.08541915 до 0.256528926
В колонке SYP_N данные распределены от 0.258625833 до 0.7595884390000001
В колонке H3AcK18_N данные распределены от 0.079690896 до 0.47976326799999996
В колонке EGR1_N данные распределены от 0.10553720400000001 до 0.360692103
В колонке H3MeK4_N данные распределены от 0.10178700800000001 до 0.413902681
В колонке CaNA_N данные распределены от 0.586478779 до 2.129791061
```

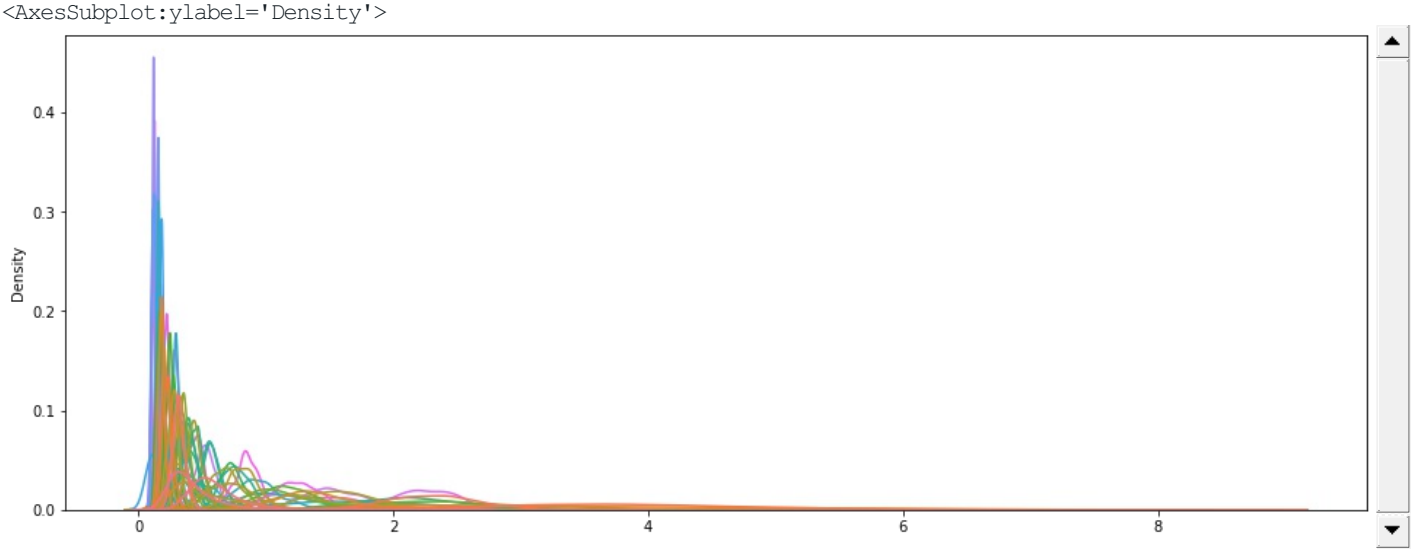
Практически все данные распределены в промежутке [0, 3], так что масштабирование можно не проводить.

In [23]:

```
import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(16, 6))
sns.kdeplot(data=X, legend=False)
```

Out[23]:

```
<AxesSubplot:ylabel='Density'>
```



Разделение датасета на обучающую и тестовую выборку

In [24]:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)
```

Построение и обучение моделей

Логистическая регрессия

In [25]:

```
from sklearn.linear_model import LogisticRegression
log_regr = LogisticRegression(max_iter=1000)
```

In [26]:

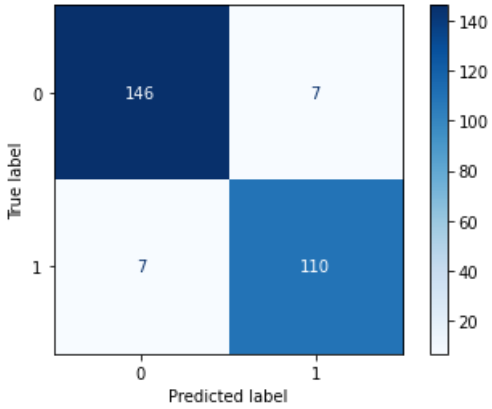
```
log_regr.fit(X_train, y_train)
log_predicted = log_regr.predict(X_test)
```

In [27]:

```
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

plot_confusion_matrix(log_regr, X_test, y_test, cmap=plt.cm.Blues)
```

Out[27]:

```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x20c71ede310>
```



In [28]:

```
def print_metrics(y_true, y_predicted):
    print("{:<15} {:<15}".format('метрика','значение'))
    print()
    print("{:<15} {:<15}".format('accuracy', round(accuracy_score(y_true, y_predicted),4)))
    print("{:<15} {:<15}".format('precision', round(precision_score(y_true, y_predicted),4)))
    print("{:<15} {:<15}".format('recall', round(recall_score(y_true, y_predicted),4)))
    print("{:<15} {:<15}".format('f1_score', round(f1_score(y_true, y_predicted),4)))
```

In [29]:

```
print_metrics(y_test, log_predicted)
```

```
метрика         значение

accuracy        0.9481
precision       0.9402
recall          0.9402
f1_score        0.9402
```

In [30]:

```
from sklearn.metrics import roc_curve, roc_auc_score
proba = log_regr.predict_proba(X_test)
true_proba = proba[:,1]
roc_auc_score(y_test, true_proba)
```

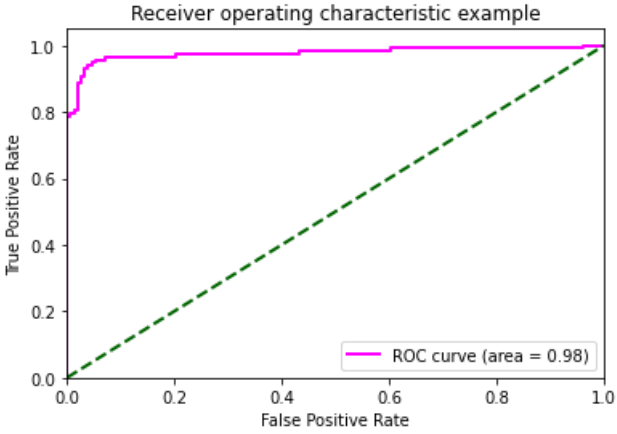
Out[30]:

```
0.9763141723926038
```

In [31]:

```
# воспользуемся написанной Юрием Евгеньевичем функцией для отрисовки ROC-кривой
# https://nbviewer.jupyter.org/github/ugapanyuk/ml_course_2021/blob/main/common/notebooks/metrics/metrics.ipyn
```

```
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()

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



In [32]:

```
# воспользуемся блоком кода, приведённым Юрием Евгеньевичем в лекции
# https://nbviewer.jupyter.org/github/ugapanyuk/ml_course_2021/blob/main/common/notebooks/linear/linear.ipynb

from typing import Dict
def accuracy_score_for_classes(y_true: np.ndarray, y_pred: np.ndarray) -> Dict[int, float]:
    d = {'t': y_true, 'p': y_pred}
    df = pd.DataFrame(data=d)
    # Метки классов
    classes = np.unique(y_true)
    # Результирующий словарь
    res = dict()
    # Перебор меток классов
    for c in classes:
        # отфильтруем данные, которые соответствуют
        # текущей метке класса в истинных значениях
        temp_data_flt = df[df['t']==c]
        # расчет accuracy для заданной метки класса
        temp_acc = accuracy_score(
            temp_data_flt['t'].values,
            temp_data_flt['p'].values)
        # сохранение результата в словарь
        res[c] = temp_acc
    return res

def print_accuracy_score_for_classes(
    y_true: np.ndarray,
    y_pred: np.ndarray):
    """
    Вывод метрики accuracy для каждого класса
    """
    accs = accuracy_score_for_classes(y_true, y_pred)
    if len(accs)>0:
        print('Метка \t Accuracy')
    for i in accs:
        print('{} \t {}'.format(i, accs[i]))

print_accuracy_score_for_classes(y_test, log_predicted)
```

```
Метка   Accuracy
0   0.954248366013072
1   0.9401709401709402
```

Решающее дерево

In [33]:

```
from sklearn.tree import DecisionTreeClassifier
```

In [34]:

```
tree_cl = DecisionTreeClassifier(criterion='entropy', max_depth=10, random_state=42)
```

In [35]:

```
tree_cl.fit(X_train, y_train)
tree_cl_predicted = tree_cl.predict(X_test)
```

In [36]:

```
print_metrics(y_test, tree_cl_predicted)
```

```
метрика         значение

accuracy        0.8926
precision       0.8929
recall          0.8547
f1_score        0.8734
```

Найдём оптимальный гиперпараметр для дерева.

In [37]:

```
from sklearn.model_selection import GridSearchCV

params = {'max_depth' : range(1,30,1)}

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

grid = GridSearchCV(DecisionTreeClassifier(), params, scoring=scoring, refit='accuracy',n_jobs = -1)
grid.fit(X_train, y_train)
```

Out[37]:

```
GridSearchCV(estimator=DecisionTreeClassifier(), n_jobs=-1,
             param_grid={'max_depth': range(1, 30)}, refit='accuracy',
             scoring={'AUC': 'roc_auc', 'F-measure': 'f1',
                      'accuracy': 'accuracy', 'precision': 'precision',
                      'recall': 'recall'})
```

In [38]:

```
pruned_tree_predicted = grid.predict(X_test)
```

In [39]:

```
print_metrics(y_test, pruned_tree_predicted)
```

```
метрика         значение

accuracy        0.9148
precision       0.9273
recall          0.8718
f1_score        0.8987
```

In [40]:

```
tree_cl = DecisionTreeClassifier(criterion='entropy',max_depth=3, random_state=42)
tree_cl.fit(X_train, y_train)
tree_cl_predicted = tree_cl.predict(X_test)
accuracy_score(y_test, tree_cl_predicted)
```

Out[40]:

```
0.837037037037037
```

In [41]:

```
tree_cl = DecisionTreeClassifier(criterion='entropy',max_depth=6, random_state=42)
tree_cl.fit(X_train, y_train)
tree_cl_predicted = tree_cl.predict(X_test)
accuracy_score(y_test, tree_cl_predicted)
```

Out[41]:

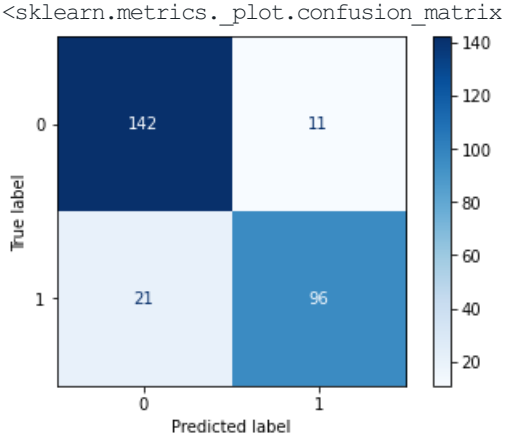
```
0.8814814814814815
```

In [42]:

```
plot_confusion_matrix(tree_cl, X_test, y_test, cmap=plt.cm.Blues)
```

Out[42]:

```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x20c720a1ca0>
```



In [43]:

```
proba = tree_cl.predict_proba(X_test)
true_proba = proba[:,1]
roc_auc_score(y_test, true_proba)
```

Out[43]:

```
0.8973521032344561
```

Показатели чуть пониже, чем у логистической регрессии, но всё равно очень достойные.

In [44]:

```
type(tree_cl.feature_importances_)
```

Out[44]:

```
numpy.ndarray
```

In [45]:

```
feature_importance = tree_cl.feature_importances_.tolist()
```

In [46]:

```
# важность признаков
# я проверила на всех 77 и поняла, что удобнее выводить только те признаки, чья значимость превышает ноль
sum = 0
for i in range(len(feature_importance)):
    if feature_importance[i] != 0.0:
        print("{:<10} {:<8}".format(X.columns.values[i], feature_importance[i]))
        sum += feature_importance[i]
print('\nСумма = ', sum)
```

```
ITSN1_N    0.015930686316212094
NR2A_N     0.008932943898668786
pERK_N     0.03618482920331274
PKCA_N     0.019550406180847104
pNR1_N     0.0120058401062572
BRAF_N     0.013547151673691867
ELK_N      0.015228063560231065
JNK_N      0.016850601187793798
RSK_N      0.005192635694112281
APP_N      0.3182461625954617
SOD1_N     0.017455777264733515
AMPKA_N    0.12994876505466504
NR2B_N     0.10456724041820686
pNUMB_N    0.027608638202502936
ADARB1_N   0.05582845687413816
RRP1_N     0.009211149185146697
BAX_N      0.01781565399641388
ERBB4_N    0.014086824933884376
Tau_N      0.021304766954710862
GluR3_N    0.03817914556930272
GluR4_N    0.007418265303870218
P3525_N    0.020723322900698105
pCASP9_N   0.013747883472990857
pCFOS_N    0.011257367147487325
SYP_N      0.016238816941875605
H3AcK18_N  0.020011377659880265
CaNA_N     0.012927227702904104

Сумма =  1.0
```

SVM

LinearSVC

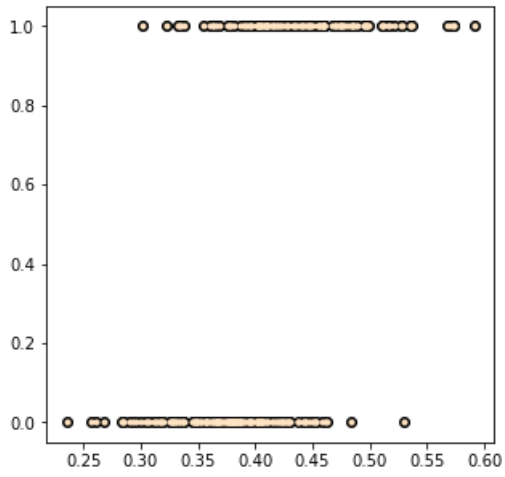
In [47]:

```
from sklearn.svm import LinearSVC
```

In [48]:

```
lin_svc = LinearSVC(C=1.0, penalty='l2',max_iter=100000)
lin_svc.fit(X_train, y_train)
lin_svc_predicted = lin_svc.predict(X_test)

# на графике будем выводить точки с признаком, по мнению решающего дерева, наиболее влияющего и важного

fig, ax = plt.subplots(figsize=(5,5))
ax.scatter(X_test['APP_N'], y_test, color='black')
ax.plot(X_test['APP_N'], lin_svc_predicted, 'b.', color='#ffe4c4')
```

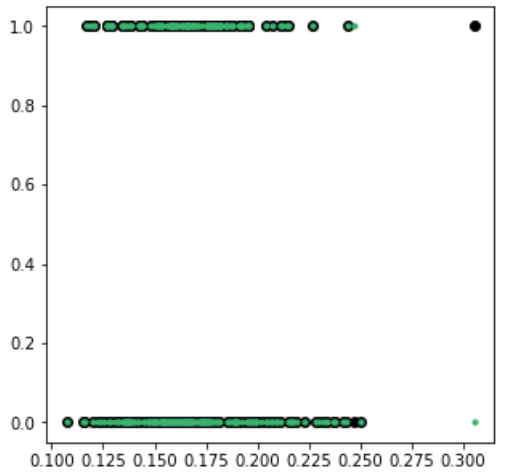
```
plt.show()
```



На графике видно, как светлые (предсказанные) точки полностью перекрывают чёрные (истинные) значения.

In [49]:

```
# также взглянем на график с точками признака, для которого дерево решений показало не такой уровень значимос

fig, ax = plt.subplots(figsize=(5,5))
plt.scatter(X_test['RSK_N'], y_test, color='black')
plt.plot(X_test['RSK_N'], lin_svc_predicted, 'b.', color='#3cb371')
plt.show()
```



In [50]:

```
print_metrics(y_test, lin_svc_predicted)
```

```
метрика         значение

accuracy        0.9815
precision       0.9912
recall          0.9658
f1_score        0.9784
```

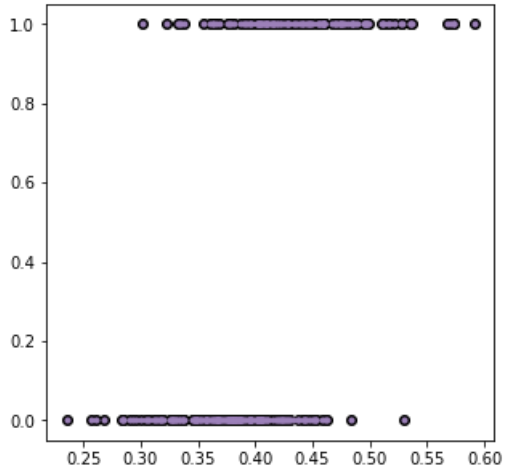
NuSVC

In [51]:

```
from sklearn.svm import NuSVC
```

In [52]:

```
params = {'nu':np.arange(0.1,0.9,0.1)}

nu_grid = GridSearchCV(NuSVC(kernel='linear'), params)

nu_grid.fit(X_train, y_train)
nu_predicted = nu_grid.predict(X_test)

fig, ax = plt.subplots(figsize=(5,5))
plt.scatter(X_test['APP_N'], y_test, color='black')
plt.plot(X_test['APP_N'], nu_predicted, 'b.', color='#9d81ba')
plt.show()
```



In [53]:

```
print_metrics(y_test, nu_predicted)
```

```
метрика         значение

accuracy        0.9704
precision       0.9739
recall          0.9573
f1_score        0.9655
```

SVC (kernel trick)

Радиально-базисная функция

In [54]:

```
from sklearn.svm import SVC
```

In [55]:

```
rbf_svc = SVC(kernel='rbf', gamma=0.2, C=100.0)
```

In [56]:

```
rbf_svc.fit(X_train, y_train)
rbf_predicted = rbf_svc.predict(X_test)
print_metrics(y_test, rbf_predicted)
```

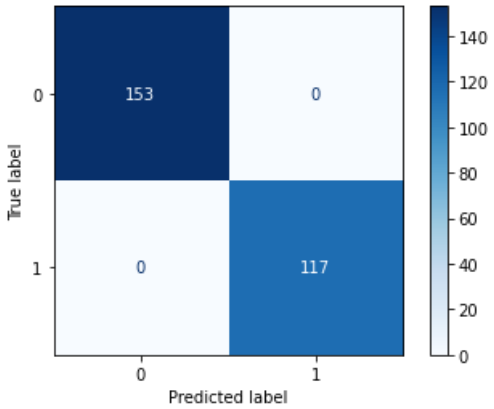
```
метрика         значение

accuracy        1.0
precision       1.0
recall          1.0
f1_score        1.0
```

In [57]:

```
plot_confusion_matrix(rbf_svc, X_test, y_test, cmap=plt.cm.Blues)
```

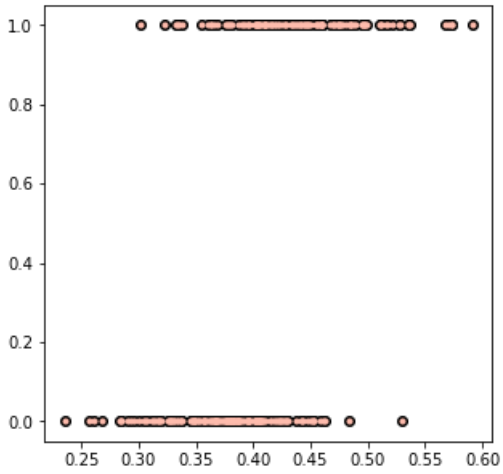
Out[57]:

```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x20c71fc7dc0>
```



In [58]:

```
fig, ax = plt.subplots(figsize=(5,5))
plt.scatter(X_test['APP_N'], y_test, color='black')
plt.plot(X_test['APP_N'], rbf_predicted, 'b.', color='#ffbcad')
plt.show()
```



Полиномиальная функция

Сначала обучим SVC с полиномиальным ядром и произвольными параметрами.

In [59]:

```
poly_svc = SVC(kernel='poly', degree = 4, gamma=0.4, C=1.0)
poly_svc.fit(X_train, y_train)
poly_predicted = poly_svc.predict(X_test)
print_metrics(y_test, poly_predicted)
```

```
метрика         значение

accuracy        0.9852
precision       0.9829
recall          0.9829
f1_score        0.9829
```

Теперь найдём оптимальные значения.

In [60]:

```
params = [{'degree':range(2,15,1)}]

poly_grid = GridSearchCV(SVC(kernel='poly'), params, scoring='accuracy', n_jobs=-1)
poly_grid.fit(X_train, y_train)
poly_predicted = poly_grid.predict(X_test)
poly_grid.best_params_
```

Out[60]:

```
{'degree': 11}
```

In [61]:

```
params = [{'gamma':np.arange(0.05,0.95,0.05)}]

poly_grid = GridSearchCV(SVC(kernel='poly'), params, scoring='accuracy', n_jobs=-1)
poly_grid.fit(X_train, y_train)
poly_predicted = poly_grid.predict(X_test)
poly_grid.best_params_
```

Out[61]:

```
{'gamma': 0.15000000000000002}
```

In [62]:

```
params = [{'C':np.geomspace(0.05,1000,30)}]

poly_grid = GridSearchCV(SVC(kernel='poly'), params, scoring='accuracy', n_jobs=-1)
poly_grid.fit(X_train, y_train)
poly_predicted = poly_grid.predict(X_test)
poly_grid.best_params_
```

Out[62]:

```
{'C': 255.12586098293818}
```

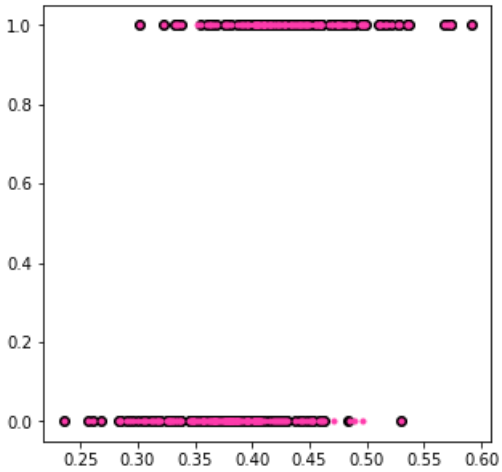
In [63]:

```
poly_svc = SVC(kernel='poly', degree = 11, gamma=0.14, C=255.0)
poly_svc.fit(X_train, y_train)
poly_predicted = poly_svc.predict(X_test)
print_metrics(y_test, poly_predicted)
```

```
метрика         значение

accuracy        0.9444
precision       0.9322
recall          0.9402
f1_score        0.9362
```

In [64]:

```
fig, ax = plt.subplots(figsize=(5,5))
plt.scatter(X_test['APP_N'], y_test, color='black')
plt.plot(X_test['APP_N'], poly_predicted, 'b.', color='#ff3cad')
plt.show()
```



Выводы

На выбранном мной датасете наилучшим образом повела себя модель опорных векторов с радиально-базисным ядром.