`Дисциплина: Методы и технологии машинного обучения`
`Уровень подготовки: бакалавриат`
`Направление подготовки: 01.03.02 Прикладная математика и информатика`
`Семестр: осень 2022/2023`

Лабораторная работа №4: Методы снижения размерности. Регуляризация логистической регрессии.

В практических примерах ниже показано:

- как снижать размерность пространства признаков методами главных компонент (PCR), частных наименьшах квадратов (PLS)
- как строить логистическую регрессию с регуляризацией параметров (методы ридж и лассо)

Точность всех моделей оценивается методом перекрёстной проверки по 10 блокам.

Модели: множественная линейная регрессия *Данные*: `Wines` (источник: репозиторий к книге С.Рашки Python и машинное обучение, глава 4)

In [1]:
```
# настройка ширины страницы блокнота .....................................
from IPython.core.display import display, HTML
display(HTML("<style>.container { width:80% !important; }</style>"))
```

Указания к выполнению

Загружаем пакеты

In [2]:
```
# загрузка пакетов: инструменты ----------------------------------------------
#  работа с массивами
import numpy as np
#  фреймы данных
import pandas as pd
#  распределение Стьюдента для проверки значимости
from scipy.stats import t
# подсчёт частот внутри массива
from collections import Counter
#  графики
import matplotlib as mpl
#  стили и шаблоны графиков на основе matplotlib
import seaborn as sns

# загрузка пакетов: данные ---------------------------------------------------
from sklearn import datasets
```

```
# загрузка пакетов: модели ------------------------------------------------
#  стандартизация показателей
from sklearn.preprocessing import StandardScaler
#  метод главных компонент
from sklearn.decomposition import PCA
# метод частных наименьших квадратов
from sklearn.cross_decomposition import PLSRegression
#  логистическая регрессия (ММП)
from sklearn.linear_model import LogisticRegression, LogisticRegressionCV
#  перекрёстная проверка по k блокам
from sklearn.model_selection import KFold, cross_val_score
#  расчёт Acc и сводка по точности классификации
from sklearn.metrics import accuracy_score, classification_report
```

In [3]:

```
# константы
#  ядро для генератора случайных чисел
my_seed = 9212
#  создаём псевдоним для короткого обращения к графикам
plt = mpl.pyplot
# настройка стиля и отображения графиков
#  примеры стилей и шаблонов графиков:
#  http://tonysyu.github.io/raw_content/matplotlib-style-gallery/gallery.html
mpl.style.use('seaborn-whitegrid')
sns.set_palette("Set2")
# раскомментируйте следующую строку, чтобы посмотреть палитру
# sns.color_palette("Set2")
```

In [4]:

```
# функция, которая строит график сжатия коэффициентов в ридж и лассо
#  из репозитория к книге С.Рашки Python и машинное обучение,
#  слегка переработанная
def plot_coeffs_traces (X, y, class_number, penalty_name, C_opt, col_names,
                        C_min_pow=-4, C_max_pow=3.) :
    fig = plt.figure()
    ax = plt.subplot(111)

    # палитра
    colors = sns.color_palette("Spectral", len(col_names)-1)

    weights, params = [], []
    for c in np.arange(C_min_pow, C_max_pow+1):
        lr = LogisticRegression(penalty=penalty_name,
                                C=10.**c, solver='liblinear',
                                multi_class='ovr', random_state=my_seed)
        lr.fit(X, y)
        weights.append(lr.coef_[class_number])
        params.append(10**c)

    weights = np.array(weights)

    for column, color in zip(range(weights.shape[1]), colors):
        plt.plot(params, weights[:, column],
                 label=col_names[column],
                 color=color)

    # отсечки по оптимальным C
    plt.axvline(x=C_opt[class_number], color='magenta',
                linestyle='--', linewidth=1)
```

```
    plt.axhline(0, color='black', linestyle='--', linewidth=1)
    plt.xlim([10**(C_min_pow), 10**C_max_pow])
    plt.ylabel('weight coefficient')
    plt.xlabel('C')
    plt.xscale('log')
    plt.legend(loc='upper left')
    ax.legend(loc='upper center',
              bbox_to_anchor=(1.38, 1.03),
              ncol=1, fancybox=True)
    plt.show()
```

Загружаем данные

Набор данных `wine` можно загрузить напрямую из пакета `sklearn` (набор впервые выложен на сайте Калифорнийского университета в Ирвине). Таблица содержит результаты химического анализа вин, выращенных в одном регионе Италии тремя разными производителями. Большинство столбцов таблицы отражают содержание в вине различных веществ:

- `alcohol` – алкоголь, процентное содержание;
- `malic_acid` – яблочная кислота (разновидность кислоты с сильной кислотностью и ароматом яблока);
- `ash` – зола (неорганическая соль, оказывающая влияние на вкус и придающая вину ощущение свежести);
- `alcalinity_of_ash` – щелочность золы;
- `magnesium` – магний (важный для организма слабощелочной элемент, способствующий энергетическому обмену);
- `total_phenols` – всего фенола (молекулы, содержащие полифенольные вещества; имеют горький вкус, также влияют на цвет, относятся к питательным веществам в вине);
- `flavanoids` – флаваноиды (полезный антиоксидант, даёт богатый аромат и горький вкус);
- `nonflavanoid_phenols` – фенолы нефлаваноидные (специальный ароматический газ, устойчивый к окислению);
- `proanthocyanins` – проантоцианы (биофлавоноидное соединение, которое также является природным антиоксидантом с легким горьковатым запахом);
- `color_intensity` – интенсивность цвета;
- `hue` – оттенок (мера яркости цвета, используется в т.ч. для измерения возраста вина);
- `od280/od315_of_diluted_wines` – OD280/OD315 разбавленных вин (метод определения концентрации белка);
- `proline` – пролин (основная аминокислота в красном вине, важный фактор вкуса и аромата);
- `target` – целевая переменная: класс вина.

Загружаем данные во фрейм и выясняем их размерность.

In [5]:
```
# загружаем таблицу и превращаем её во фрейм
DF_all = datasets.load_wine(as_frame=True)['frame']
```

```
# выясняем размерность фрейма
print('Число строк и столбцов в наборе данных:\n', DF_all.shape)
```

```
Число строк и столбцов в наборе данных:
 (178, 14)
```

Отложим 10% наблюдений для прогноза.

In [6]:
```
# наблюдения для моделирования
DF = DF_all.sample(frac = 0.9, random_state = my_seed)
# отложенные наблюдения
DF_predict = DF_all.drop(DF.index)
```

In [7]:
```
# первые 5 строк фрейма у первых 7 столбцов
DF.iloc[:, :7].head(5)
```

Out[7]:

	alcohol	malic_acid	ash	alcalinity_of_ash	magnesium	total_phenols	flavanoids
142	13.52	3.17	2.72	23.5	97.0	1.55	0.52
92	12.69	1.53	2.26	20.7	80.0	1.38	1.46
163	12.96	3.45	2.35	18.5	106.0	1.39	0.70
40	13.56	1.71	2.31	16.2	117.0	3.15	3.29
117	12.42	1.61	2.19	22.5	108.0	2.00	2.09

In [8]:
```
# первые 5 строк фрейма у столбцов 8-11
DF.iloc[:, 7:11].head(5)
```

Out[8]:

	nonflavanoid_phenols	proanthocyanins	color_intensity	hue
142	0.50	0.55	4.35	0.89
92	0.58	1.62	3.05	0.96
163	0.40	0.94	5.28	0.68
40	0.34	2.34	6.13	0.95
117	0.34	1.61	2.06	1.06

In [9]:
```
# первые 5 строк фрейма у столбцов 12-14
DF.iloc[:, 11:].head(5)
```

Out[9]:

	od280/od315_of_diluted_wines	proline	target
142	2.06	520.0	2
92	2.06	495.0	1
163	1.75	675.0	2
40	3.38	795.0	0
117	2.96	345.0	1

In [10]:
```
# типы столбцов фрейма
DF.dtypes
```

Out[10]:
```
alcohol                         float64
malic_acid                      float64
ash                             float64
alcalinity_of_ash               float64
magnesium                       float64
total_phenols                   float64
flavanoids                      float64
nonflavanoid_phenols            float64
proanthocyanins                 float64
color_intensity                 float64
hue                             float64
od280/od315_of_diluted_wines    float64
proline                         float64
target                            int32
dtype: object
```

Проверим, нет ли в таблице пропусков.

In [11]:
```
# считаем пропуски в каждом столбце
DF.isna().sum()
```

Out[11]:
```
alcohol                         0
malic_acid                      0
ash                             0
alcalinity_of_ash               0
magnesium                       0
total_phenols                   0
flavanoids                      0
nonflavanoid_phenols            0
proanthocyanins                 0
color_intensity                 0
hue                             0
od280/od315_of_diluted_wines    0
proline                         0
target                          0
dtype: int64
```

Пропусков не обнаружено.

Предварительный анализ данных

Описательные статистики

Считаем доли классов целевой переменной `target`.

In [12]:
```
# метки классов
DF.target.unique()
```

Out[12]:
```
array([2, 1, 0])
```

In [13]:
```
# доли классов
```

```
np.around(DF.target.value_counts() / len(DF.index), 3)
```

Out[13]:
```
1    0.394
0    0.338
2    0.269
Name: target, dtype: float64
```

Итак, всего целевых классов три, и их доли примерно одинаковы, с перевесом в пользу класса '0'. Все объясняющие переменные набора данных непрерывные. Рассчитаем для них описательные статистики.

In [14]:
```
# описательные статистики
DF.iloc[:, :6].describe()
```

Out[14]:

	alcohol	malic_acid	ash	alcalinity_of_ash	magnesium	total_phenols
count	160.000000	160.000000	160.000000	160.000000	160.000000	160.000000
mean	13.001000	2.311250	2.366250	19.490625	100.106250	2.288000
std	0.823077	1.092852	0.279871	3.295142	14.479479	0.629285
min	11.030000	0.740000	1.360000	10.600000	70.000000	0.980000
25%	12.355000	1.607500	2.210000	17.175000	89.000000	1.735000
50%	13.050000	1.830000	2.360000	19.500000	98.000000	2.265000
75%	13.682500	3.030000	2.560000	21.500000	107.250000	2.800000
max	14.830000	5.650000	3.230000	30.000000	162.000000	3.880000

In [15]:
```
# описательные статистики
DF.iloc[:, 6:11].describe()
```

Out[15]:

	flavanoids	nonflavanoid_phenols	proanthocyanins	color_intensity	hue
count	160.000000	160.000000	160.000000	160.000000	160.000000
mean	2.003062	0.370750	1.590250	5.027937	0.959288
std	1.014983	0.126404	0.590739	2.257101	0.222782
min	0.340000	0.130000	0.410000	1.280000	0.540000
25%	1.090000	0.270000	1.235000	3.200000	0.797500
50%	2.065000	0.345000	1.545000	4.800000	0.960000
75%	2.892500	0.470000	1.952500	6.200000	1.120000
max	5.080000	0.660000	3.580000	13.000000	1.710000

In [16]:
```
# описательные статистики
DF.iloc[:, 11:13].describe()
```

Out[16]:

	od280/od315_of_diluted_wines	proline

	od280/od315_of_diluted_wines	proline
count	160.000000	160.000000
mean	2.601062	758.043750
std	0.699801	314.274513
min	1.270000	278.000000
25%	1.952500	510.000000
50%	2.780000	679.000000
75%	3.170000	986.250000
max	3.920000	1680.000000

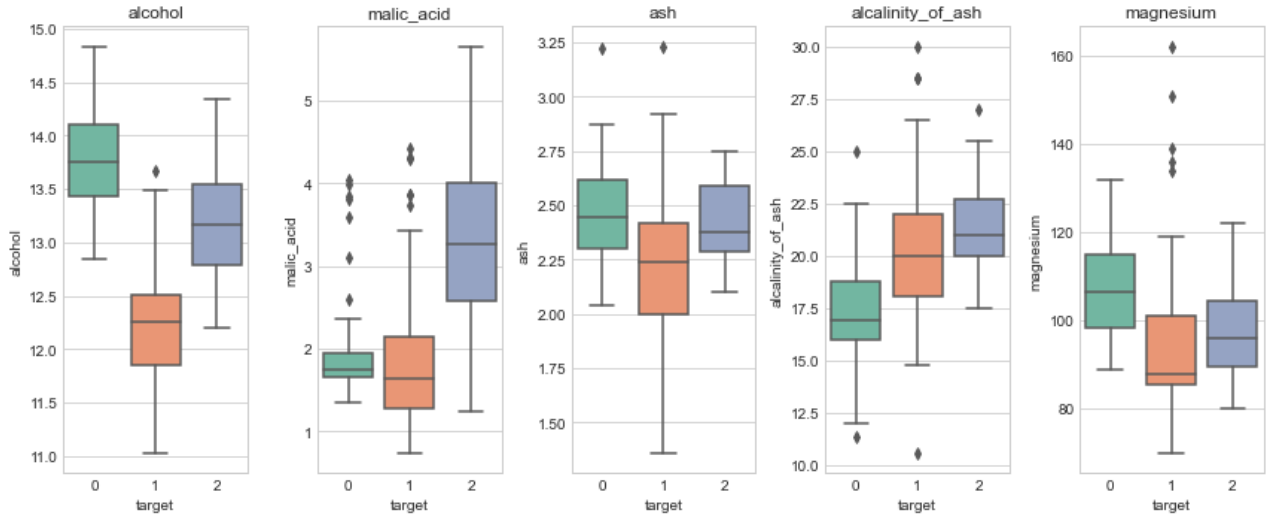
Выводы по описательным статистикам: значения объясняющих переменных положительные, масштабы измерения отличаются. Для работы с методами снижения размерности и регуляризации понадобится стандартизация значений.

Визуализация разброса переменных внутри классов

Поскольку в наборе данных 13 объясняющих переменных, и все они непрерывные, анализ матричного графика разброса будет затруднительным. Построим коробчатые диаграммы для объясняющих переменных, чтобы сравнить средние уровни и разброс по классам.

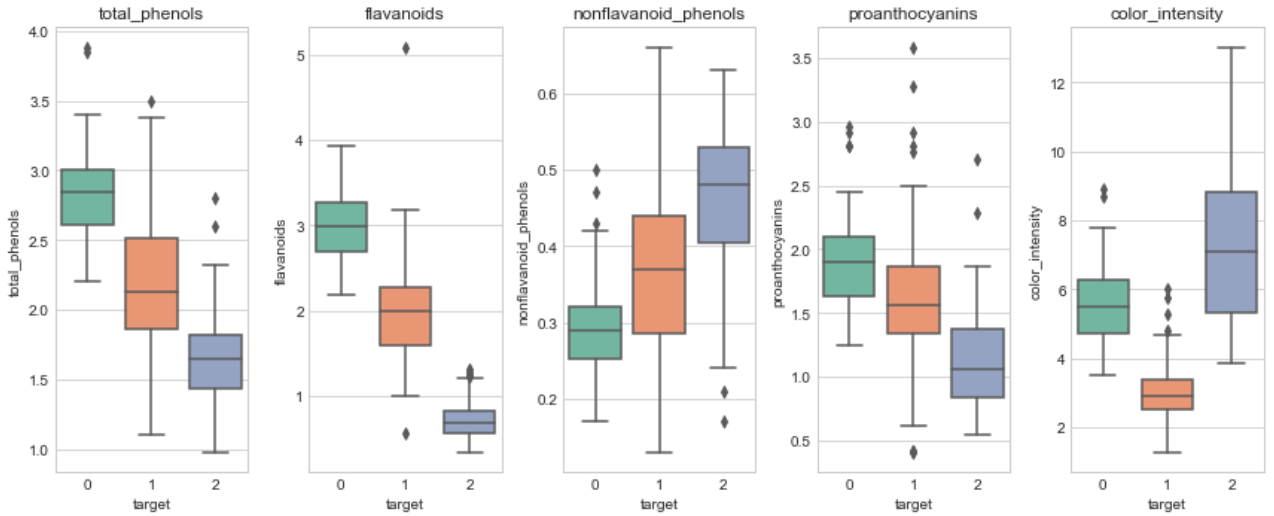
In [17]:

```
# создаём полотно и делим его на четыре части
fig = plt.figure(figsize=(12, 5))
gs = mpl.gridspec.GridSpec(1, 5)
ax1 = plt.subplot(gs[0, 0])
ax2 = plt.subplot(gs[0, 1])
ax3 = plt.subplot(gs[0, 2])
ax4 = plt.subplot(gs[0, 3])
ax5 = plt.subplot(gs[0, 4])

axs = [ax1, ax2, ax3, ax4, ax5]

cols_loop = list(DF.columns[:5].values)
for col_name in cols_loop :
    i = cols_loop.index(col_name)
    sns.boxplot(x='target', y=col_name, data=DF, ax=axs[i])
    axs[i].set_ylabel(col_name)
    axs[i].set_title(col_name)

# корректируем расположение графиков на полотне
gs.tight_layout(plt.gcf())
plt.show()
```



In [18]:

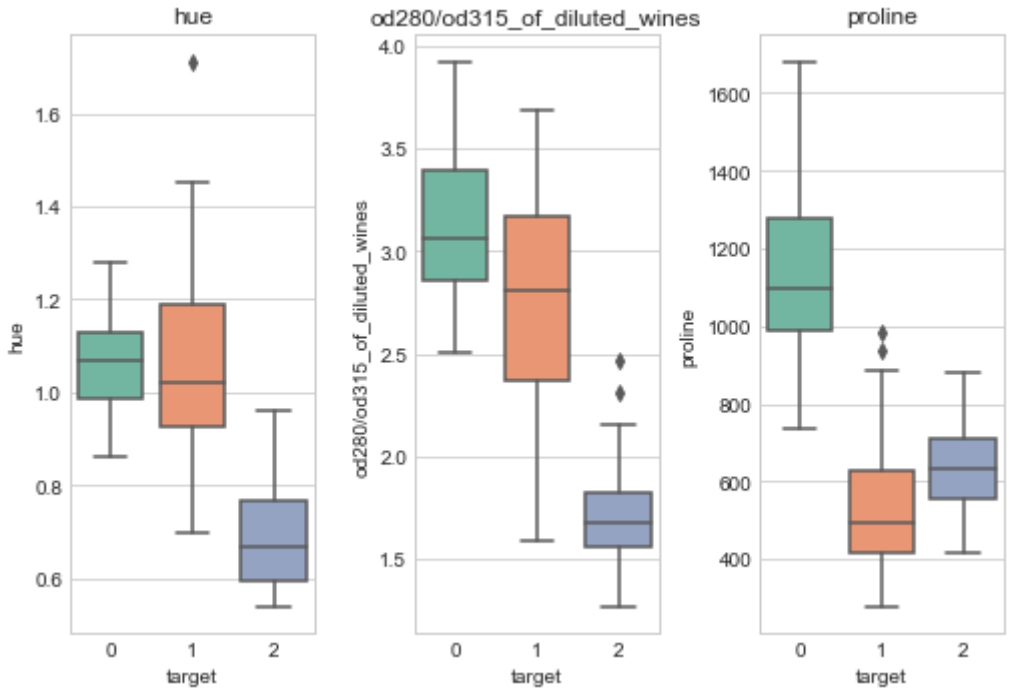
```
# создаём полотно и делим его на четыре части
fig = plt.figure(figsize=(12, 5))
gs = mpl.gridspec.GridSpec(1, 5)
ax1 = plt.subplot(gs[0, 0])
ax2 = plt.subplot(gs[0, 1])
ax3 = plt.subplot(gs[0, 2])
ax4 = plt.subplot(gs[0, 3])
ax5 = plt.subplot(gs[0, 4])

axs = [ax1, ax2, ax3, ax4, ax5]

cols_loop = list(DF.columns[5:10].values)
for col_name in cols_loop :
    i = cols_loop.index(col_name)
    sns.boxplot(x='target', y=col_name, data=DF, ax=axs[i])
    axs[i].set_ylabel(col_name)
    axs[i].set_title(col_name)

# корректируем расположение графиков на полотне
gs.tight_layout(plt.gcf())
plt.show()
```



In [19]:

```
# создаём полотно и делим его на четыре части
fig = plt.figure(figsize=(7.2, 5))
```

```
gs = mpl.gridspec.GridSpec(1, 3)
ax1 = plt.subplot(gs[0, 0])
ax2 = plt.subplot(gs[0, 1])
ax3 = plt.subplot(gs[0, 2])

axs = [ax1, ax2, ax3]

cols_loop = list(DF.columns[10:13].values)
for col_name in cols_loop :
    i = cols_loop.index(col_name)
    sns.boxplot(x='target', y=col_name, data=DF, ax=axs[i])
    axs[i].set_ylabel(col_name)
    axs[i].set_title(col_name)

# корректируем расположение графиков на полотне
gs.tight_layout(plt.gcf())
plt.show()
```



На графиках отличие в медианах и разбросе между классами прослеживается практически по всем объясняющим переменным. Меньше всего различаются коробчатые диаграммы по переменной `ash`. Это говорит о том, классы по зависимой переменной `target` неплохо разделяются по всем объясняющим переменным.

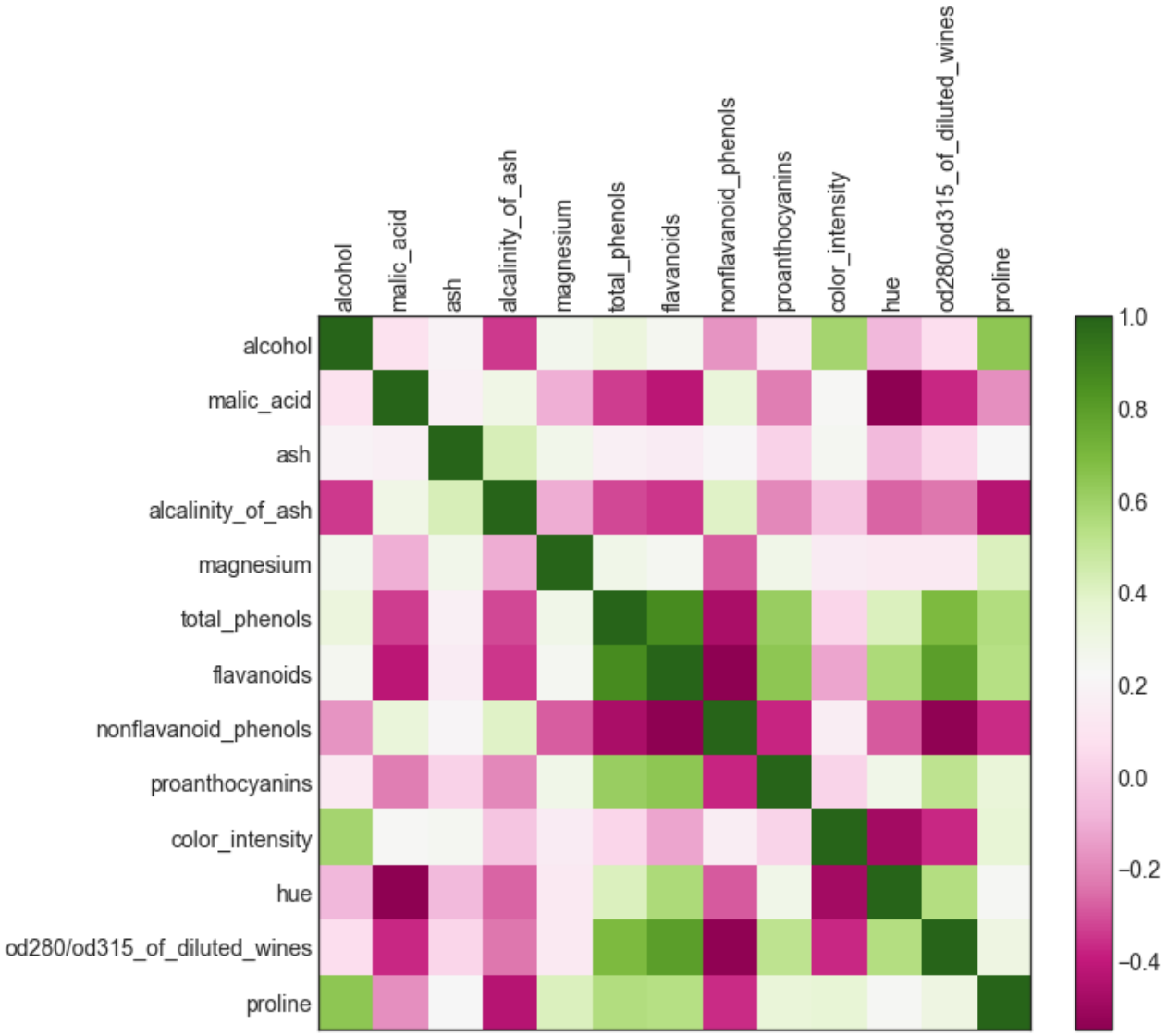
Корреляционный анализ

Теперь посмотрим на взаимодействие объясняющих переменных.

In [20]:

```
# рассчитываем корреляционную матрицу
corr_mat = DF.drop('target', axis=1).corr()
col_names = DF.drop('target', axis=1).columns

# переключаем стиль оформления, чтобы убрать сетку с тепловой карты
mpl.style.use('seaborn-white')

# рисуем корреляционную матрицу
```

```
f = plt.figure(figsize=(10, 8))
plt.matshow(corr_mat, fignum=f.number, cmap='PiYG')
# координаты для названий строк и столбцов
tics_coords = np.arange(0, len(col_names))
# рисуем подписи
plt.xticks(tics_coords, col_names, fontsize=14, rotation=90)
plt.yticks(tics_coords, col_names, fontsize=14)
# настраиваем легенду справа от тепловой карты
cb = plt.colorbar()
cb.ax.tick_params(labelsize=14)
cb.ax.tick_params(labelsize=14)
plt.show()
```



Между объясняющими переменными обнаруживаются как прямые, так и обратные линейные взаимосвязи. Выведем все значимые коэффициенты в одной таблице и определим минимальный / максимальный из них.

In [21]:

```
# делаем фрейм из корреляционной матрицы и стираем диагональные значения
#  и нижний треугольник матрицы
df = corr_mat
df = df.where(np.triu(np.ones(df.shape), k=1).astype(bool))
# меняем размерность с матрицы на таблицу: показатель 1, показатель 2,
#  корреляция
```

```
df = df.stack().reset_index()
df.columns = ['Показатель_1', 'Показатель_2', 'Корреляция']
# считаем двусторонние p-значения для проверки значимости
t_stat = np.sqrt((len(DF.index) - 2) / (1 - df.Корреляция.values ** 2))
df['P_значение'] = 2*(1 - t.cdf(abs(t_stat), len(DF.index) - 2))
# получили все корреляционные коэффициенты без 1 и без повторов
#  выводим все значимые с сортировкой
df.loc[df['P_значение'] < 0.05].sort_values('Корреляция')
```

Out[21]:

	Показатель_1	Показатель_2	Корреляция	P_значение
57	flavanoids	nonflavanoid_phenols	-0.548931	0.0
20	malic_acid	hue	-0.547041	0.0
66	nonflavanoid_phenols	od280/od315_of_diluted_wines	-0.538438	0.0
72	color_intensity	hue	-0.482839	0.0
51	total_phenols	nonflavanoid_phenols	-0.464909	0.0
...	...	...	...	...
11	alcohol	proline	0.647394	0.0
58	flavanoids	proanthocyanins	0.647605	0.0
55	total_phenols	od280/od315_of_diluted_wines	0.693227	0.0
61	flavanoids	od280/od315_of_diluted_wines	0.796361	0.0
50	total_phenols	flavanoids	0.865223	0.0

78 rows × 4 columns

Методы снижения резмерности

Посмотрим, как работают методы снижения размерности:

- регрессия на главные компоненты (PCR)
- частный метод наименьших квадратов (PLS)

Оба метода требуют предварительной стандартизации переменных.

In [22]:

```
# стандартизация
sc = StandardScaler()
X_train_std = sc.fit_transform(DF.iloc[:, :13].values)

# проверяем средние и стандартные отклонения после стандартизации
for i_col in range(X_train_std.shape[1]) :
    print('Столбец ', i_col, ': среднее = ',
          np.round(np.mean(X_train_std[:, i_col]), 2),
         '   Станд. отклонение = ',
          np.round(np.std(X_train_std[:, i_col]), 2), sep='')
```

```
Столбец 0: среднее = 0.0   Станд. отклонение = 1.0
Столбец 1: среднее = -0.0   Станд. отклонение = 1.0
Столбец 2: среднее = 0.0   Станд. отклонение = 1.0
```

```
Столбец 3: среднее = -0.0   Станд. отклонение = 1.0
Столбец 4: среднее = -0.0   Станд. отклонение = 1.0
Столбец 5: среднее = 0.0   Станд. отклонение = 1.0
Столбец 6: среднее = 0.0   Станд. отклонение = 1.0
Столбец 7: среднее = -0.0   Станд. отклонение = 1.0
Столбец 8: среднее = 0.0   Станд. отклонение = 1.0
Столбец 9: среднее = -0.0   Станд. отклонение = 1.0
Столбец 10: среднее = 0.0   Станд. отклонение = 1.0
Столбец 11: среднее = 0.0   Станд. отклонение = 1.0
Столбец 12: среднее = -0.0   Станд. отклонение = 1.0
```

Регрессия на главные компоненты (PCR)

Пересчитаем объясняющие показатели в главные компоненты.

In [23]:

```
# функция с методом главных компонент
pca = PCA()
# пересчитываем в главные компоненты (ГК)
X_train_pca = pca.fit_transform(X_train_std)

# считаем доли объяснённой дисперсии
frac_var_expl = pca.explained_variance_ratio_
print('Доли объяснённой дисперсии по компонентам в PLS:\n',
      np.around(frac_var_expl, 3),
      '\nОбщая сумма долей:', np.around(sum(frac_var_expl), 3))
```

```
Доли объяснённой дисперсии по компонентам в PLS:
 [0.368 0.19  0.115 0.069 0.063 0.05  0.041 0.026 0.023 0.018 0.017 0.013
 0.007]
Общая сумма долей: 1.0
```

Главные компоненты взаимно ортогональны, убедимся в этом.

In [24]:

```
# ГК ортогональны - убедимся в этом, рассчитыв корреляционную матрицу
corr_mat = pd.DataFrame(X_train_pca).corr()
np.around(corr_mat, 2)
```

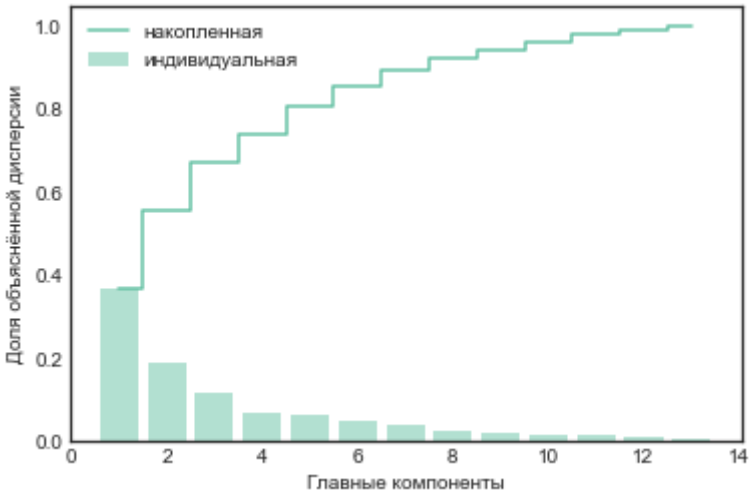
Out[24]:

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	1.0	-0.0	0.0	-0.0	0.0	0.0	-0.0	0.0	0.0	0.0	-0.0	0.0	-0.0
1	-0.0	1.0	-0.0	-0.0	-0.0	-0.0	-0.0	-0.0	-0.0	-0.0	0.0	0.0	-0.0
2	0.0	-0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-0.0	-0.0	0.0
3	-0.0	-0.0	0.0	1.0	0.0	-0.0	0.0	0.0	0.0	-0.0	-0.0	0.0	0.0
4	0.0	-0.0	0.0	0.0	1.0	0.0	-0.0	0.0	-0.0	-0.0	0.0	-0.0	-0.0
5	0.0	-0.0	0.0	-0.0	0.0	1.0	-0.0	0.0	0.0	0.0	-0.0	-0.0	0.0
6	-0.0	-0.0	0.0	0.0	-0.0	-0.0	1.0	-0.0	-0.0	-0.0	-0.0	0.0	-0.0
7	0.0	-0.0	0.0	0.0	0.0	0.0	-0.0	1.0	0.0	0.0	-0.0	-0.0	0.0
8	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
9	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
10	-0.0	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	1	2	3	4	5	6	7	8	9	10	11	12
11	0.0	0.0	-0.0	0.0	-0.0	-0.0	0.0	-0.0	0.0	0.0	-0.0	1.0	0.0
12	-0.0	-0.0	0.0	0.0	-0.0	0.0	-0.0	0.0	0.0	-0.0	0.0	0.0	1.0

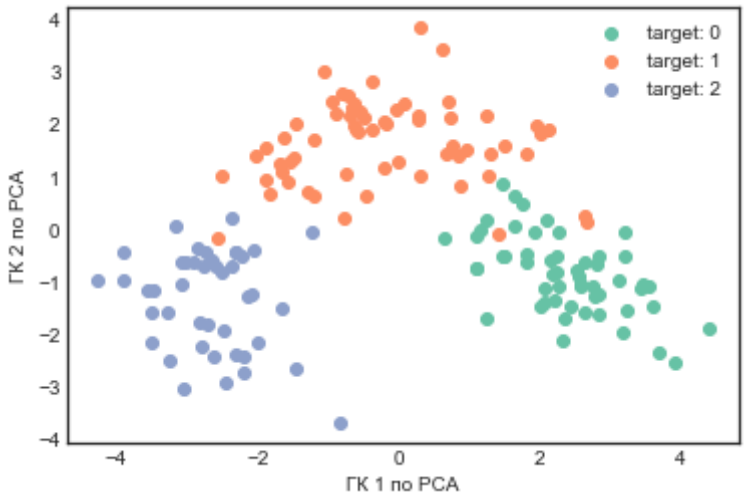
Построим график объяснённой дисперсии.

In [25]:
```
# график объяснённой дисперсии
plt.bar(range(1, 14), pca.explained_variance_ratio_, alpha=0.5,
        align='center', label='индивидуальная')
plt.step(range(1, 14), np.cumsum(pca.explained_variance_ratio_),
         where='mid', label='накопленная')
plt.ylabel('Доля объяснённой дисперсии')
plt.xlabel('Главные компоненты')
plt.legend()
plt.show()
```



Столбцы на графике показывают долю исходной дисперсии исходных переменных, которую объясняет главная компонента. Линией показана накопленная доля. Так, видно, что первые 5 компонент объясняют 80% исходной дисперсии X.
Чтобы увидеть, как классы выглядят в координатах ГК на графике, придётся сократить пространство для двух компонент, которые объясняют 56% разброса объясняющих переменных.

In [26]:
```
# пересчитываем X в 2 ГК
pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train_std)

# график классов в пространстве ГК
plt.scatter(X_train_pca[DF['target'] == 0][:, 0],
            X_train_pca[DF['target'] == 0][:, 1], label='target: 0')
plt.scatter(X_train_pca[DF['target'] == 1][:, 0],
            X_train_pca[DF['target'] == 1][:, 1], label='target: 1')
plt.scatter(X_train_pca[DF['target'] == 2][:, 0],
            X_train_pca[DF['target'] == 2][:, 1], label='target: 2')
plt.xlabel('ГК 1 по PCA')
plt.ylabel('ГК 2 по PCA')
plt.legend()
plt.show()
```



Судя по графику, классы неплохо разделяются в пространстве двух главных компонент. Построим логистическую регрессию и оценим её точность с помощью перекрёстной проверки.

In [27]:
```
# функция оценки логистической регрессии
logit = LogisticRegression()
# функция разбиения на блоки для перекрёстной проверки
kf_10 = KFold(n_splits=10, random_state=my_seed, shuffle=True)
# считаем точность модели (Acc) с перекрёстной проверкой по блокам
score = list()
acc = cross_val_score(logit, X_train_pca, DF.target.values, cv=kf_10,
                      scoring='accuracy').mean()
score.append(np.around(acc, 3))
score_models = list()
score_models.append('logit_PC2')
print('Модель ', score_models[0], ', перекрёстная проверка по 10 блокам',
      '\nAcc = ', np.around(score[0], 2), sep='')
```

```
Модель logit_PC2, перекрёстная проверка по 10 блокам
Acc = 0.96
```

Метод частных наименьших квадратов

Сначала посмотрим, как работает метод на всех наблюдениях обучающего набора.

In [28]:
```
# функция для оценки модели, берём все компоненты, по числу столбцов X
pls = PLSRegression(n_components=13)
# значения зависимой переменной превращаем в фиктивные по классам
Y_train = pd.get_dummies(DF[['target']].astype(str))
# оцениваем
pls.fit(X_train_std, Y_train)

# считаем долю объяснённой дисперсии
frac_var_expl = np.var(pls.x_scores_, axis = 0) / np.sum(np.var(X_train_std, axi
print('Доли объяснённой дисперсии по компонентам в PLS:\n',
     np.around(frac_var_expl, 3),
     '\nОбщая сумма долей:', np.around(sum(frac_var_expl), 3))
```

```
Доли объяснённой дисперсии по компонентам в PLS:
```

```
 [0.361 0.186 0.054 0.073 0.038 0.025 0.024 0.027 0.03  0.038 0.023 0.017
 0.014]
Общая сумма долей: 0.911
```

Из-за того, что при вычислении компонент метдом PLS мы учитываем корреляцию с Y, компоненты, во-первых, не ортогональны, а во-вторых сумма объяснённых долей дисперсии уже не равняется 1.

In [29]:
```
# сокращаем пространство компонент до 2
pls = PLSRegression(n_components=2)
# перестраиваем модель
pls.fit(X_train_std, Y_train)
# пересчитываем X
X_train_pls = pls.transform(X_train_std)
# предсказываем принадлежности классов для обучающего набора
Y_train_pred = pls.predict(X_train_std)
pd.DataFrame(Y_train_pred)
```

Out[29]:

	0	1	2
0	-0.053349	0.273117	0.780232
1	-0.168385	0.809259	0.359126
2	0.031179	0.180927	0.787894
3	0.828136	0.216990	-0.045126
4	0.026077	0.985468	-0.011545
...	...	...	...
155	-0.206711	0.121626	1.085085
156	-0.074481	0.781661	0.292820
157	1.312968	-0.367024	0.054056
158	-0.070547	0.178859	0.891688
159	-0.079943	1.106192	-0.026249

160 rows × 3 columns

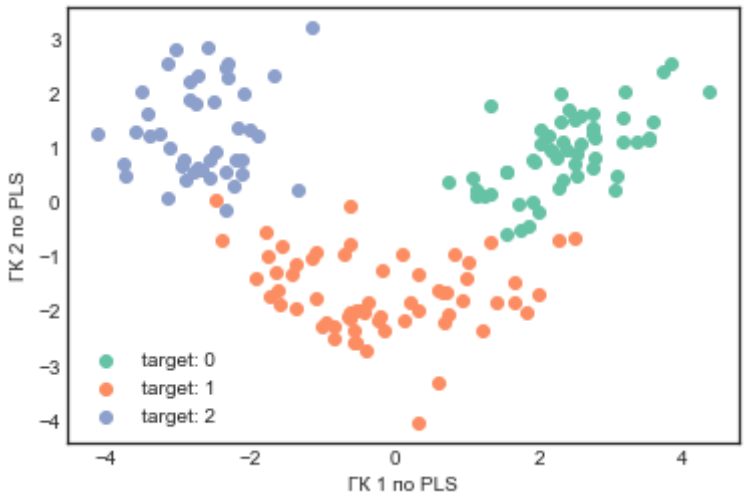
In [30]:
```
# вычисляем классы
Y_train_hat = list()
for y_i in Y_train_pred :
    Y_train_hat.append([i for i in range(len(y_i)) if y_i[i] == max(y_i)][0])

# сколько наблюдений попали в каждый класс по модели
Counter(Y_train_hat).items()
```

Out[30]:
```
dict_items([(2, 44), (1, 61), (0, 55)])
```

Рисуем классы на графике в координатах 2 главных компонент по PLS.

In [31]:
```
# график классов в пространстве ГК
plt.scatter(X_train_pls[DF['target'] == 0][:, 0],
            X_train_pls[DF['target'] == 0][:, 1], label='target: 0')
```

```
plt.scatter(X_train_pls[DF['target'] == 1][:, 0],
            X_train_pls[DF['target'] == 1][:, 1], label='target: 1')
plt.scatter(X_train_pls[DF['target'] == 2][:, 0],
            X_train_pls[DF['target'] == 2][:, 1], label='target: 2')
plt.xlabel('ГК 1 по PLS')
plt.ylabel('ГК 2 по PLS')
plt.legend()
plt.show()
```



Видно, что в координатах двух компонент, рассчитанных методом частных наименьших квадратов, классы также оказываются хорошо разделимы.
Теперь оценим точность модели с перекрёстной проверкой.

In [32]:

```
# функция разбиения на блоки для перекрёстной проверки
#  для чистоты эксперимента возьмём другое ядро генератора случайных чисел
kf_10 = KFold(n_splits=10, random_state=my_seed+1, shuffle=True)
# считаем точность модели (Acc) с перекрёстной проверкой по блокам
#  функция cross_val_score не сработает, т.к. у нас мультиклассовая
#  классификация, поэтому делаем вручную
# значения Y как метки классов
Y_train = DF.target.values
# значения Y как фиктивные переменные
Y_train_dummy = pd.get_dummies(Y_train.astype(str))
# модель внутри блока
pls_cv = PLSRegression(n_components=2)
# для записи Acc по блокам
acc_blocks = list()
# цикл по блокам
for train_index, test_index in kf_10.split(X_train_std, DF.target.values) :
    # данные для модели внутри блока
    X_i_train = X_train_std[train_index]
    Y_i_train = Y_train_dummy.iloc[train_index, :]

    # данные для прогноза вне блока
    X_i_test = X_train_std[test_index]
    Y_i_test = Y_train[test_index]

    # оцениваем модель на блоке
    pls_cv.fit(X_i_train, Y_i_train)
    # делаем прогноз у вне блока
    Y_pred = pls.predict(X_i_test)
    Y_hat = list()
```

```
    for y_i in Y_pred :
        Y_hat.append([i for i in range(len(y_i)) if y_i[i] == max(y_i)][0])
    # считаем точность
    acc = accuracy_score(Y_i_test, Y_hat)
    acc_blocks.append(acc)

score.append(np.around(np.mean(acc_blocks), 3))
score_models.append('logit_PLS')
print('Модель ', score_models[1], ', перекрёстная проверка по 10 блокам',
      '\nAcc = ', np.around(score[1], 2), sep='')
```

```
Модель logit_PLS, перекрёстная проверка по 10 блокам
Acc = 0.98
```

Методы сжатия

Ридж-регрессия

Функция `LogisticRegression()` умеет работать с мультиклассовой классификацией, используя при оценке параметров подход **один класс против остальных**. Построим ридж на наших данных.

In [33]:
```
# функция для построения модели
logit_ridge = LogisticRegression(penalty='l2', solver='liblinear')
# оцениваем параметры
logit_ridge.fit(X_train_std, Y_train)
# выводим параметры
print('Константы моделей для классов:\n', np.around(logit_ridge.intercept_, 3),
      '\nКоэффициенты моделей для классов:\n', np.around(logit_ridge.coef_, 3))
```

```
Константы моделей для классов:
 [-1.389 -1.236 -2.173]
Коэффициенты моделей для классов:
 [[ 1.418  0.331  0.966 -1.309  0.012  0.273  0.909 -0.246 -0.116  0.026
   0.12   0.867  1.777]
 [-1.552 -0.671 -1.144  0.704 -0.099 -0.016  0.476  0.386  0.231 -1.722
   0.982  0.109 -1.75 ]
 [ 0.414  0.408  0.478  0.336  0.094 -0.149 -1.442 -0.164 -0.362  1.413
  -1.027 -0.899 -0.   ]]
```

Подбираем гиперпараметр регуляризации λ с помощью перекрёстной проверки. В функции `LogisticRegression()` есть аргумент C – это инверсия гиперпараметра λ.

In [34]:
```
# поиск оптимального значения C:
#  подбираем C по наибольшей точности с перекрёстной проверкой
ridge_cv = LogisticRegressionCV(cv=10, random_state=my_seed+2,
                                penalty='l2', solver='liblinear')
ridge_cv.fit(X_train_std, Y_train)
# значения параметра C (инверсия лямбды), которые дают наилучшую
#  точность для каждого класса
ridge_cv.C_
```

Out[34]:
```
array([2.7825594 , 0.04641589, 2.7825594 ])
```

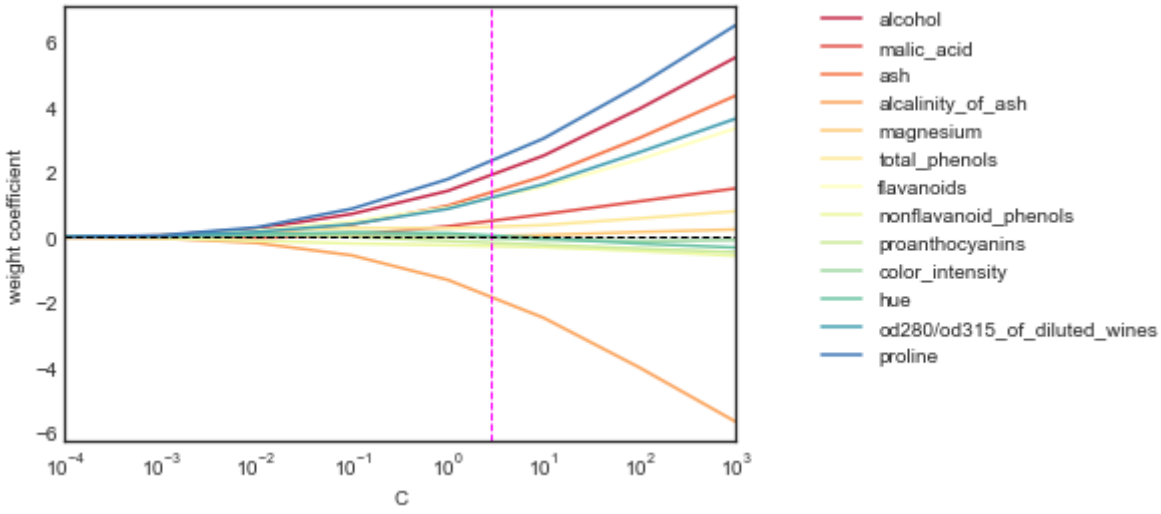
In [35]:
```
# сохраняем и выводим Acc для модели
```

```
score.append(np.around(ridge_cv.score(X_train_std, Y_train), 3))
score_models.append('logit_ridge')
print('Модель ', score_models[2], ', перекрёстная проверка по 10 блокам',
      '\nAcc = ', score[2], sep='')
```

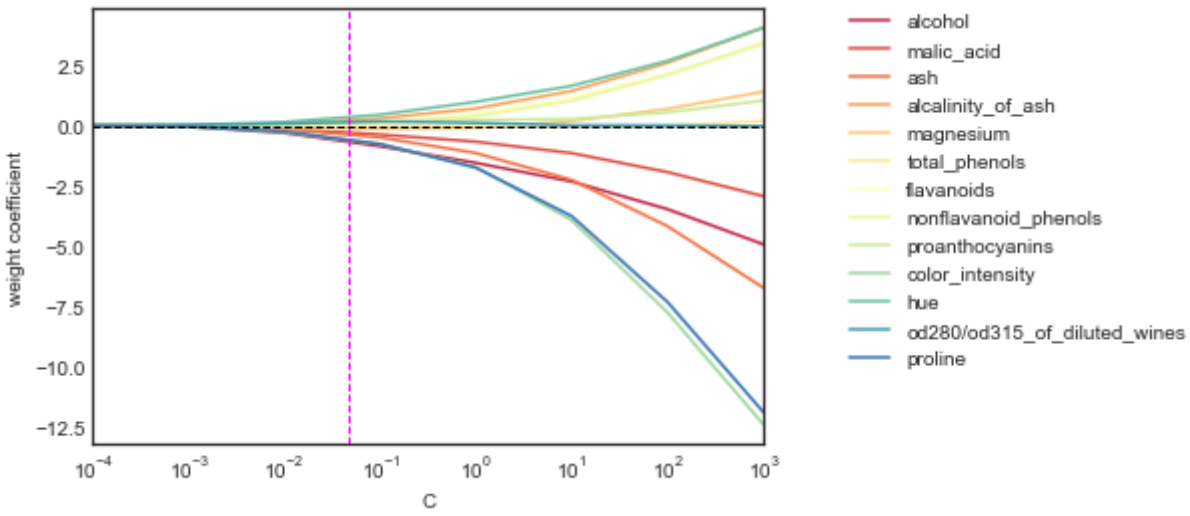
```
Модель logit_ridge, перекрёстная проверка по 10 блокам
Acc = 1.0
```

Изобразим изменение коэффициентов ридж-регрессии на графике и сделаем отсечку на уровне оптимального параметра C.

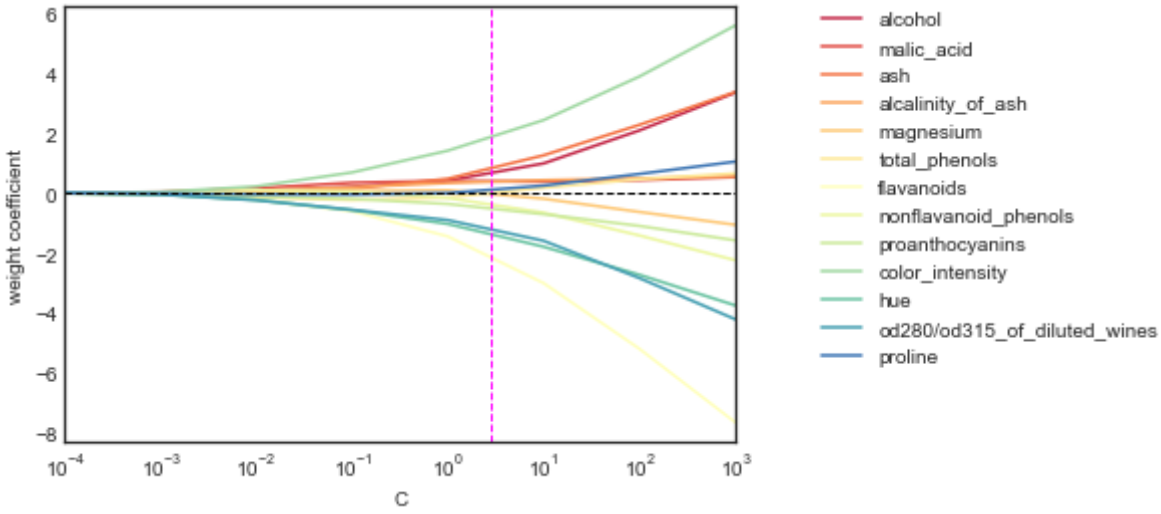
In [36]:
```
# график динамики коэффициентов в ридж-регрессии
#  модель для класса 0
plot_coeffs_traces(X_train_std, Y_train, 0, 'l2', ridge_cv.C_, DF.columns)
```



In [37]:
```
# график динамики коэффициентов в ридж-регрессии
#  модель для класса 1
plot_coeffs_traces(X_train_std, Y_train, 1, 'l2', ridge_cv.C_, DF.columns)
```



In [38]:
```
# график динамики коэффициентов в ридж-регрессии
#  модель для класса 2
plot_coeffs_traces(X_train_std, Y_train, 2, 'l2', ridge_cv.C_, DF.columns)
```



Лассо-регрессия

Технически реализация лассо-регрессии отличается от ридж единственным аргументом `penalty='l1'` в функции `LogisticRegression`.

In [39]:
```
# функция для построения модели
logit_lasso = LogisticRegression(penalty='l1', solver='liblinear')
# оцениваем параметры
logit_lasso.fit(X_train_std, Y_train)
# выводим параметры
print('Константы моделей для классов:\n', np.around(logit_lasso.intercept_, 3),
      '\nКоэффициенты моделей для классов:\n', np.around(logit_lasso.coef_, 3))
```

```
Константы моделей для классов:
 [-1.506 -1.553 -2.444]
Коэффициенты моделей для классов:
 [[ 1.512  0.     0.96  -1.3    0.     0.     0.899  0.     0.     0.
   0.     1.055  2.407]
 [-1.635 -0.535 -1.089  0.502  0.     0.     0.497  0.223  0.089 -2.199
   1.18   0.    -2.275]
 [ 0.05   0.248  0.507  0.     0.     0.    -2.42  -0.035  0.     1.87
  -1.111 -0.596  0.   ]]
```

Отметим, что в векторе коэффициентов появились нулевые значения: метод лассо позволяет обнулять коэффициенты, тем самым отбрасывая слабые объясняющие переменные.

In [40]:
```
# поиск оптимального значения C:
#  подбираем C по наибольшей точности с перекрёстной проверкой
lasso_cv = LogisticRegressionCV(cv=10, random_state=my_seed+3,
                                penalty='l1', solver='liblinear')
lasso_cv.fit(X_train_std, Y_train)
# значения параметра C (инверсия лямбды), которые дают наилучшую
#  точность для каждого класса
lasso_cv.C_
```

Out[40]:
```
array([2.7825594 , 0.35938137, 0.35938137])
```

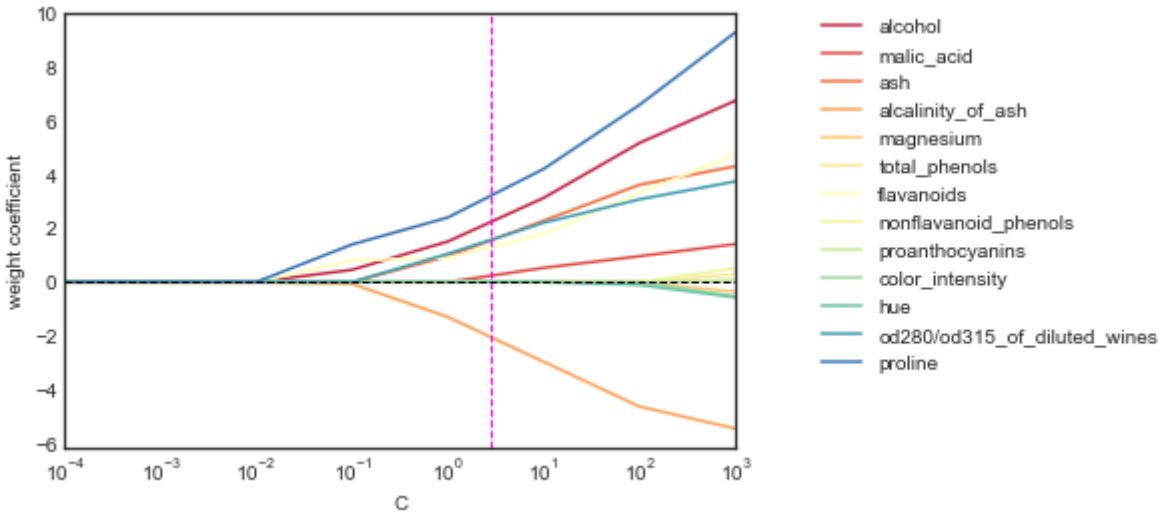
In [41]:
```
# сохраняем и выводим Acc для модели
score.append(np.around(lasso_cv.score(X_train_std, Y_train), 3))
```

```
score_models.append('logit_lasso')
print('Модель ', score_models[3], ', перекрёстная проверка по 10 блокам',
      '\nAcc = ', score[3], sep='')
```

```
Модель logit_lasso, перекрёстная проверка по 10 блокам
Acc = 0.994
```

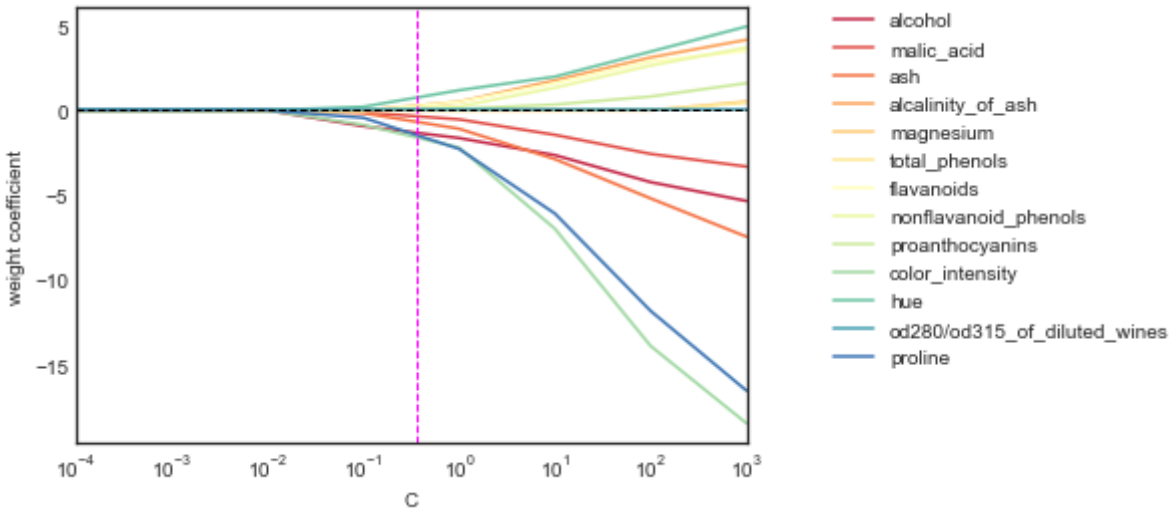
In [42]:

```
# график динамики коэффициентов в лассо-регрессии
#  модель для класса 0
plot_coeffs_traces(X_train_std, Y_train, 0, 'l1', lasso_cv.C_, DF.columns)
```



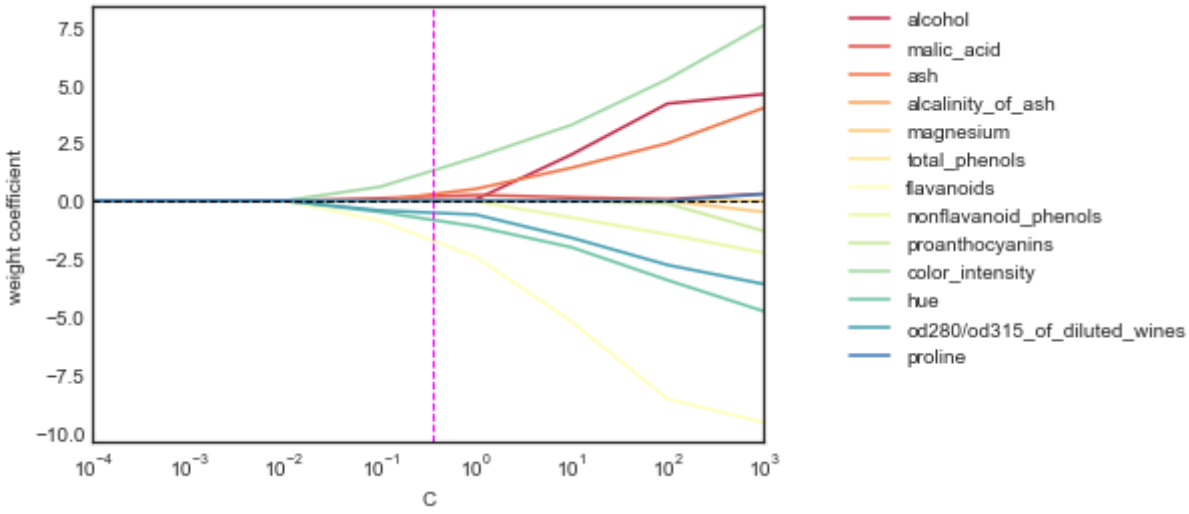
In [43]:

```
# график динамики коэффициентов в лассо-регрессии
#  модель для класса 1
plot_coeffs_traces(X_train_std, Y_train, 1, 'l1', lasso_cv.C_, DF.columns)
```



In [44]:

```
# график динамики коэффициентов в лассо-регрессии
#  модель для класса 2
plot_coeffs_traces(X_train_std, Y_train, 2, 'l1', lasso_cv.C_, DF.columns)
```



Итак, судя по графикам, для значения гиперпараметра, дающего самую точную модель, ни один коэффициент при объясняющих переменных не обнуляется. Это подтверждает наблюдение, сделанное нами ещё на этапе предварительного анализа: все объясняющие переменные неплохо разделяют классы.

Прогноз на отложенные наблюдения по лучшей модели

Ещё раз посмотрим на точность построенных моделей.

In [45]:
```
# сводка по точности моделей
pd.DataFrame({'Модель' : score_models, 'Acc' : score})
```

Out[45]:

	Модель	Acc
0	logit_PC2	0.956
1	logit_PLS	0.975
2	logit_ridge	1.000
3	logit_lasso	0.994

Все модели показывают высокую точность по показателю Acc, при этом самой точной оказывается ридж-регрессия. Сделаем прогноз на отложенные наблюдения.

In [46]:
```
# формируем объекты с данными отложенной выборки
X_pred_std = sc.fit_transform(DF_predict.iloc[:, :13].values)
Y_pred = DF_predict.target
Y_hat = logit_ridge.predict(X_pred_std)
# отчёт по точности на отложенных наблюдениях
print(classification_report(Y_pred, Y_hat))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         5
           1       1.00      1.00      1.00         8
```

```
           2       1.00      1.00      1.00         5

    accuracy                           1.00        18
   macro avg       1.00      1.00      1.00        18
weighted avg       1.00      1.00      1.00        18
```

Итак, методом логистической регрессии со сжатием коэффициенты с L2-регуляризацией мы получили идеально точную модель классификации трёх видов красных вин.