`Дисциплина: Методы и технологии машинного обучения`
`Уровень подготовки: бакалавриат`
`Направление подготовки: 01.03.02 Прикладная математика и информатика`
`Семестр: осень 2021/2022`

Лабораторная работа №2: Параметрические классификаторы: логистическая регрессия, LDA, QDA

В практических примерах ниже показано:

- как импортировать данные из .csv
- как рассчитать матрицу неточностей
- как считать показатели качества модели по матрице неточностей (метод проверочной выборки)
- как пользоваться моделями логистической регрессии, линейного и квадратичного дискриминантного анализа

Модели: логистическая регрессия, LDA, QDA.
Данные: `Default`.

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
#  графики
import matplotlib as mpl
#  стили и шаблоны графиков на основе matplotlib
import seaborn as sns
#  тест Шапиро-Уилка на нормальность распределения
from scipy.stats import shapiro
#  тест Лиллиефорса на нормальность распределения
from statsmodels.stats.diagnostic import lilliefors

# загрузка пакетов: модели ---------------------------
#  логистическая регрессия (ММП)
from sklearn.linear_model import LogisticRegression
```

```
#  линейный дискриминантный анализ (LDA)
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
#  квадратичный дискриминантный анализ (QDA)
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
#  матрица неточностей
from sklearn.metrics import classification_report, confusion_matrix
#  визуализация матрицы неточностей
from sklearn.metrics import plot_confusion_matrix
#  PPV (TP / (TP + FP))
from sklearn.metrics import precision_score
#  расчёт TPR, SPC, F1
from sklearn.metrics import precision_recall_fscore_support
#  ROC-кривая
from sklearn.metrics import plot_roc_curve, roc_curve, auc
#  подготовка матрицы X для модели регрессии
from statsmodels.api import add_constant
#  модель логистической регрессии
from statsmodels.formula.api import logit
```

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")
```

Загружаем данные

Набор данных `Default` в формате .csv доступен по адресу:
https://raw.githubusercontent.com/aksyuk/MTML/main/Labs/data/Default.csv.

In [4]:

```
# читаем таблицу из файла .csv во фрейм
fileURL = 'https://raw.githubusercontent.com/aksyuk/MTML/main/Labs/data/Default.
DF = pd.read_csv(fileURL)

# делаем из категориальных (в нашем случае бинарных) переменных факторы,
#  т.е. перенумеровываем уникальные значения
DF['defaultYes'] = DF.default.factorize()[0]
DF['studentYes'] = DF.student.factorize()[0]

# первые 5 строк фрейма
DF.head(5)
```

Out[4]:

	default	student	balance	income	defaultYes	studentYes
0	No	No	729.526495	44361.625074	0	0
1	No	Yes	817.180407	12106.134700	0	1
2	No	No	1073.549164	31767.138947	0	0

	default	student	balance	income	defaultYes	studentYes
3	No	No	529.250605	35704.493935	0	0
4	No	No	785.655883	38463.495879	0	0

In [5]:
```
# типы столбцов фрейма
DF.dtypes
```

Out[5]:
```
default        object
student        object
balance       float64
income        float64
defaultYes      int64
studentYes      int64
dtype: object
```

In [6]:
```
# сколько наблюдений во фрейме
print("Число наблюдений во фрейме DF:\n", len(DF.index))
```

```
Число наблюдений во фрейме DF:
 10000
```

🗄 Подробнее о функции `factorize()`

В предыдущем блоке мы создали новые столбцы и назвали их `defaultYes` и `studentYes`, подразумевая, что значения `Yes` закодируются как 1, а значения `No` – как 0. Но на деле запустили функцию factorize() с аргументами по умолчанию, что предполагает, что уникальные элементы массива (столбца фрейма) будут закодированы по порядку, начиная с 0, *в порядке их появления*. Проверим:

In [7]:
```
# кодируем массив, который начинается с Yes, по умолчанию
labels1, uniques1 = pd.factorize(['Yes', 'No', 'Yes', 'Yes', 'No', 'No'])

print("Пронумеровенные значения: \n", labels1)
print("Уникальные коды: \n", uniques1)
```

```
Пронумеровенные значения:
 [0 1 0 0 1 1]
Уникальные коды:
 ['Yes' 'No']
```

Обратите внимание: функция возвращает два массива, первый это закодированные значения (непосредственно фактор), второй это уникальные коды значений (уровни фактора). И в этом случае нам не повезло, поскольку исходный показатель начинался с `Yes`. Если воспользоваться аргументом `sort`, функция `factorize()` сначала отсортирует уникальные коды, а уже потом их пронумерует:

In [8]:
```
# кодируем массив, который начинается с Yes, с сортировкой кодов
#  (уникальных значений) по алфавиту
labels2, uniques2 = pd.factorize(['Yes', 'No', 'Yes', 'Yes', 'No', 'No'],
                                 sort=True)
```

```
print("Пронумеровенные значения: \n", labels2)
print("Уникальные коды: \n", uniques2)
```

```
Пронумеровенные значения:
 [1 0 1 1 0 0]
Уникальные коды:
 ['No' 'Yes']
```

Однако не всегда такой подход даёт контролируемый результат, поскольку порядок уникальных значений фактора необязательно совпадает с алфавитным. Чтобы избежать сюрпризов, стоит создать словарь с порядком следования категорий, а затем применить его. Этот способ применим к объектам типа `Series` из библиотеки `pandas` (к этому типу относятся столбцы фрейма данных):

In [9]:
```
# сначала создаём словарь
x_dict = {'Yes' : 1,
          'No' : 0}
# теперь определяем x_to_factorize как столбец фрейма df_temp
df_tmp = pd.DataFrame({'x_to_factorize': ['Yes', 'No', 'Yes',
                                          'Yes', 'No', 'No']})
# создаём столбец с кодами под названием x_factor с помощью map()
df_tmp['x_factor'] = df_tmp.x_to_factorize.map(x_dict)

df_tmp
```

Out[9]:

	x_to_factorize	x_factor
0	Yes	1
1	No	0
2	Yes	1
3	Yes	1
4	No	0
5	No	0

Предварительный анализ данных

В этой лабораторной для оценки точности моделей мы используем метод проверочной выборки. Создадим фреймы с обучающей и тестовой выборками (`DF_train` и `DF_test` соответственно), распределив наблюдения случайным образом в соотношении 80% и 20%.

In [10]:
```
# обучающая выборка
DF_train = DF.sample(frac = 0.8, random_state = my_seed)
# тестовая выборка (методом исключения)
DF_test = DF.drop(DF_train.index)

# сколько наблюдений в обучающей выборке + подсчёт частот классов
print("Число наблюдений во фрейме DF_train:\n", len(DF_train.index),
      "\n\nЧастоты классов defaultYes:\n",
      np.around(DF_train.defaultYes.value_counts() / len(DF_train.index), 3),
       sep='')
```

```
Число наблюдений во фрейме DF_train:
8000

Частоты классов defaultYes:
0    0.966
1    0.034
Name: defaultYes, dtype: float64
```

In [11]:

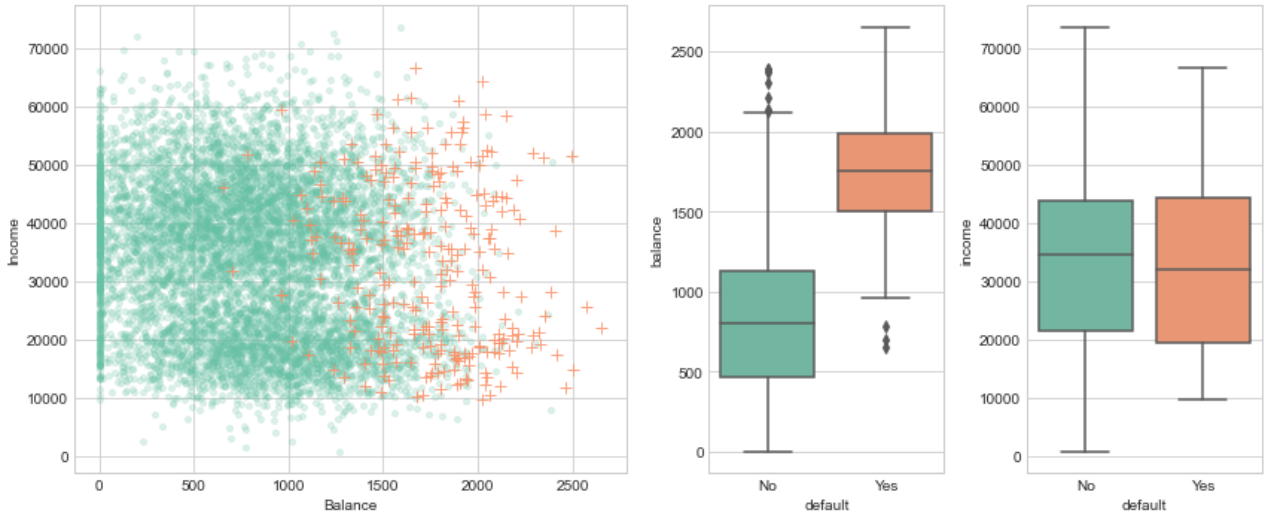
```
# сколько наблюдений в тестовой выборке + подсчёт частот классов
print("Число наблюдений во фрейме DF_test:\n", len(DF_test.index),
     "\n\nЧастоты классов defaultYes:\n",
     np.around(DF_test.defaultYes.value_counts() / len(DF_test.index), 3),
      sep='')
```

```
Число наблюдений во фрейме DF_test:
2000

Частоты классов defaultYes:
0    0.968
1    0.032
Name: defaultYes, dtype: float64
```

Посмотрим на разброс значений переменных и взаимосвязи между ними в обучающей выборке.

In [12]:

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

# график разброса
ax1.scatter(DF_train[DF_train.default == 'No'].balance,
            DF_train[DF_train.default == 'No'].income, marker='.',
            linewidths=1, alpha=.2, s=60)
ax1.scatter(DF_train[DF_train.default == 'Yes'].balance,
            DF_train[DF_train.default == 'Yes'].income,
            marker='+', linewidths=1, alpha=.8, s=60)
# подписи осей для графика разброса
ax1.set_xlabel('Balance')
ax1.set_ylabel('Income')

# строим коробчатые
sns.boxplot(x='default', y='balance', data=DF_train, orient='v', ax=ax2)
sns.boxplot(x='default', y='income', data=DF_train, orient='v', ax=ax3)

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



Логистическая регрессия

Классифицируем наблюдения по классам из `defaultYes` с помощью логистической регрессии. В качестве объясняющей переменной возьмём `balance`.

Строим модель с помощью пакета `scikit-learn`

Воспользуемся функцией `LogisticRegression()`.

In [13]:
```
# данные для логистической выборки в формате,
#  понятном функции LogisticRegression()
X_train = DF_train.balance.values.reshape(-1, 1)
y_train = DF_train.defaultYes

# строим модель на обучающей
fit_LR_1 = LogisticRegression(solver='newton-cg').fit(X=X_train, y=y_train)

# коэффициенты модели
print('Коэффициенты при объясняющих переменных:',
      np.around(fit_LR_1.coef_, 4),
      '\nКонстанта:', np.around(fit_LR_1.intercept_, 4))
```

```
Коэффициенты при объясняющих переменных: [[0.0053]]
Константа: [-10.3733]
```

Строим модель с помощью пакета `statmodels`

Воспользуемся функцией `logit()`.

In [14]:
```
fit_logit_1 = logit(str('defaultYes ~ balance'),
                    DF_train).fit(solver='newton-cg')
fit_logit_1.summary().tables[1]
```

```
Optimization terminated successfully.
         Current function value: 0.082238
         Iterations 10
```

Out[14]:

	coef	std err	z	P>|z|	[0.025	0.975]

Intercept	-10.3733	0.388	-26.770	0.000	-11.133	-9.614
balance	0.0053	0.000	22.451	0.000	0.005	0.006

Можно убедиться, что две функции строят на обучающей выборке одинаковые модели. При этом отчёт по функции `logit()` содержит P-значения (столбец P>|z|) для проверки гипотез о значимости параметров модели.
Изобразим модельные кривые на графике.

График фактических и модельных вероятностей

In [15]:
```
# данные тестовой выборки
X_test = DF_test.balance.values.reshape(-1, 1)
y_test = DF_test.defaultYes

# равноотстоящие координаты balance по возрастанию (для графика модельных значен
x_line_train = np.linspace(DF_train.balance.min(),
                           DF_train.balance.max(), 2000).reshape(-1, 1)
x_line_test = np.linspace(DF_test.balance.min(),
                          DF_test.balance.max(), 2000).reshape(-1, 1)

# прогноз вероятностей для обучающей
y_line_train = fit_LR_1.predict_proba(x_line_train)
```

Посмотрим на содержимое объекта `y_line_train` с прогнозными значениями. Из таблицы ниже очевидно, что столбцы содержат прогнозы вероятностей классов. Это прогноз для 2000 равноотстоящих координат модельной кривой по горизонтальной оси, поэтому можно видеть, как при движении по строкам фрейма сверху вниз плавно меняются принадлежности классов от `No` (значения из столбца 0 превышают значения из столбца 1) до `Yes` (наоборот).

In [16]:
```
pd.DataFrame(y_line_train)
```

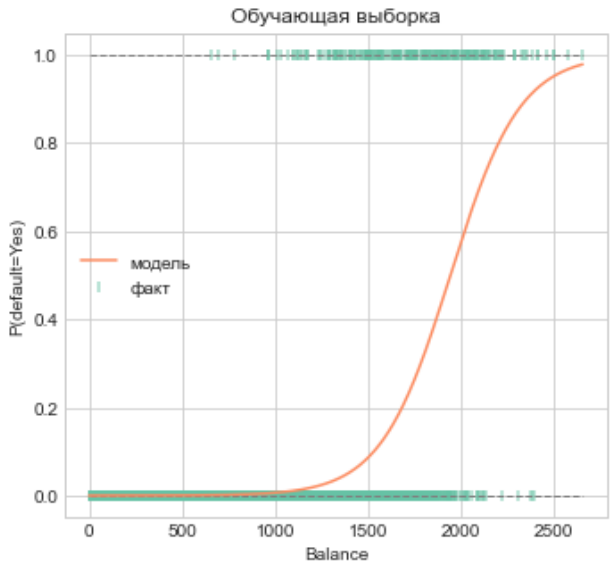
Out[16]:

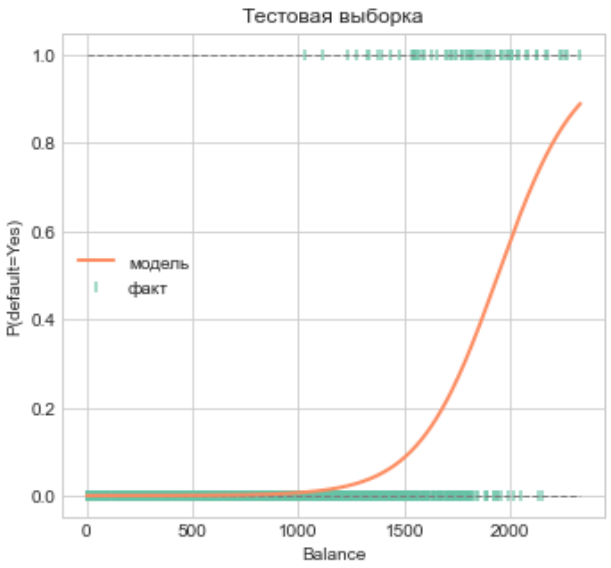
	0	1
0	0.999969	0.000031
1	0.999969	0.000031
2	0.999968	0.000032
3	0.999968	0.000032
4	0.999968	0.000032
...	...	...
1995	0.022566	0.977434
1996	0.022410	0.977590
1997	0.022255	0.977745
1998	0.022102	0.977898
1999	0.021949	0.978051

2000 rows × 2 columns

In [17]:

```
# прогноз вероятностей для тестовой
y_line_test = fit_LR_1.predict_proba(x_line_test)

# график логистической регрессии
fig, (ax1, ax2) = plt.subplots(1,2, figsize=(12,5))

# палитра
clrs = sns.color_palette('Set2')

# график для обучающей выборки
#  фактические наблюдения
ax1.scatter(X_train, y_train, marker='|', color=clrs[0],
            alpha=0.6, label='факт')
#  модельная кривая
ax1.plot(x_line_train, y_line_train[:, 1], linestyle='solid',
         color=clrs[1], label='модель')
#  заголовок
ax1.set_title('Обучающая выборка')

# график для тестовой выборки
#  фактические наблюдения
ax2.scatter(X_test, y_test, marker='|', color=clrs[0], lw=2,
            alpha=0.6, label='факт')
#  модельная кривая
ax2.plot(x_line_test, y_line_test[:, 1], linestyle='solid', lw=2,
         color=clrs[1], label='модель')
#  заголовок
ax2.set_title('Тестовая выборка')

# дополнительные настройки графиков
for ax in fig.axes:
    #  линиии вероятностей P=0 и P=1
    ax.hlines(1, xmin=ax.xaxis.get_data_interval()[0],
              xmax=ax.xaxis.get_data_interval()[1], linestyles='dashed',
              lw=1, color='grey')
    ax.hlines(0, xmin=ax.xaxis.get_data_interval()[0],
              xmax=ax.xaxis.get_data_interval()[1], linestyles='dashed',
              lw=1, color='grey')
    #  подписи осей
    ax.set_ylabel('P(default=Yes)')
    ax.set_xlabel('Balance')
    #  легенда
    ax.legend(loc='center left')
```





Выбор лучшей модели множественной логистической регрессии

Теперь построим множественную логистическую регрессию.

In [18]:
```
fit_logit_2 = logit(str('defaultYes ~ studentYes + balance + income'),
                    DF_train).fit(solver='newton-cg')
fit_logit_2.summary().tables[1]
```

```
Optimization terminated successfully.
         Current function value: 0.080803
         Iterations 10
```

Out[18]:

	coef	std err	z	P>|z|	[0.025	0.975]
Intercept	-10.5180	0.528	-19.930	0.000	-11.552	-9.484
studentYes	-0.7228	0.257	-2.814	0.005	-1.226	-0.219
balance	0.0056	0.000	22.336	0.000	0.005	0.006
income	1.855e-06	8.84e-06	0.210	0.834	-1.55e-05	1.92e-05

Исключим незначимую объясняющую `income`.

In [19]:
```
fit_logit_3 = logit(str('defaultYes ~ studentYes + balance'),
                    DF_train).fit(solver='newton-cg')
fit_logit_3.summary().tables[1]
```

```
Optimization terminated successfully.
         Current function value: 0.080806
         Iterations 10
```

Out[19]:

	coef	std err	z	P>|z|	[0.025	0.975]
Intercept	-10.4450	0.395	-26.423	0.000	-11.220	-9.670
studentYes	-0.7641	0.165	-4.641	0.000	-1.087	-0.441
balance	0.0056	0.000	22.345	0.000	0.005	0.006

Пробуем модель со взаимодействием `student` и `balance`.

In [20]:
```
fit_logit_4 = logit(str('defaultYes ~ studentYes + studentYes * balance'),
                    DF_train).fit(solver='newton-cg')
fit_logit_4.summary().tables[1]
```

```
Optimization terminated successfully.
         Current function value: 0.080799
         Iterations 10
```

Out[20]:

	coef	std err	z	P>|z|	[0.025	0.975]
Intercept	-10.3533	0.476	-21.737	0.000	-11.287	-9.420
studentYes	-1.0594	0.897	-1.181	0.238	-2.817	0.699
balance	0.0055	0.000	18.150	0.000	0.005	0.006
studentYes:balance	0.0002	0.001	0.335	0.737	-0.001	0.001

In [21]:
```
# сводим в таблицу характеристики качества моделей
#  пустые массивы для будущих столбцов
mdls = ["" for x in range(4)]
aics = np.zeros(4)
tprs = np.zeros(4)
spcs = np.zeros(4)

#  цикл по построенным моделям
fits_loop = [fit_logit_1, fit_logit_2, fit_logit_3, fit_logit_4]
for fit in fits_loop :
    #  номер текущей модели в списке
    i = fits_loop.index(fit)
    #  объясняющие переменные модели
    mdls[i] = ' '.join(fit.pvalues.index[1:])
    #  значения AIC
    aics[i] = np.around(fit.aic, 2)
    #  делаем прогноз на тестовую
    y_hat_test = fit.predict(DF_test)
    y_hat_test = (y_hat_test > 0.5).astype(int)
    #  значения TPR на тестовой
    tprs[i] = np.around(precision_score(y_test, y_hat_test), 3)
    #  значения SPC на тестовой
    spcs[i] = np.around(precision_recall_fscore_support(y_test, y_hat_test,
                                                        average='binary')[1], 3)

df_summary = pd.DataFrame({'Объясняющие переменные': mdls, 'AIC': aics,
                           'TPR_test': tprs, 'SPC_test': spcs})
df_summary
```

Out[21]:

	Объясняющие переменные	AIC	TPR_test	SPC_test
0	balance	1319.80	0.769	0.317
1	studentYes balance income	1300.85	0.800	0.317
2	studentYes balance	1298.89	0.800	0.317
3	studentYes balance studentYes:balance	1300.78	0.808	0.333

Проанализируем таблицу. Наилучшей моделью по информационному критерию Акаике

является модель зависимости `defaultYes` от `studentYes` и `balance` (модель №3, наименьшее значение AIC). Значение чувствительности на тестовой выборке (TPR_{test}) у неё также наилучшее. По специфичности (SPC_{test}) модель №4 (с переменной константой при `balance`) немного лучше остальных, однако отличие несущественное.
В итоге стоит остановиться на третьей модели. Перестроим её с помощью `LogisticRegression()`.

Перестраиваем лучшую модель с `LogisticRegression()`

In [22]:
```
# строим модель на обучающей
X_train_LR_2 = DF_train[['studentYes', 'balance']]
fit_LR_2 = LogisticRegression(solver='newton-cg').fit(X=X_train_LR_2,
                                                      y=y_train)
```

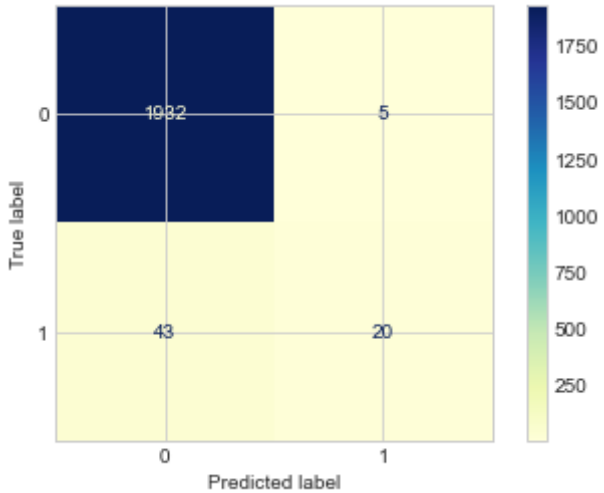
In [23]:
```
# коэффициенты
fit_LR_2.coef_
```

Out[23]:
```
array([[-0.7439704 ,  0.00555927]])
```

In [24]:
```
# константа
fit_LR_2.intercept_
```

Out[24]:
```
array([-10.43950886])
```

In [25]:
```
# прогноз
X_test_LR_2 = DF_test[['studentYes', 'balance']]
# визуализация матрицы неточностей
plot_confusion_matrix(fit_LR_2, X_test_LR_2, y_test, cmap='YlGnBu')
plt.show()
```



In [26]:
```
# отчёт по точности на тестовой
y_prob_test_LR_2 = fit_LR_2.predict_proba(X_test_LR_2).reshape(2, -1)[1]
y_hat_test = (y_prob_test_LR_2 > 0.5).astype(int)
print('Модель логистической регрессии от studentYes, balance ',
```

```
      'с порогом 0.5 : \n',
      classification_report(y_test, y_hat_test))
```

```
Модель логистической регрессии от studentYes, balance  с порогом 0.5 :
              precision    recall  f1-score   support

           0       0.97      0.50      0.66      1937
           1       0.03      0.54      0.06        63

    accuracy                           0.50      2000
   macro avg       0.50      0.52      0.36      2000
weighted avg       0.94      0.50      0.64      2000
```

В последнем отчёте метрики качества рассчитаны для каждого класса. Для бинарной классификации:

- в столбце `recall` для класса 1 стоит чувствительность, а для класса 0 – специфичность.
- в столбце `precision` для класса 1 стоит ценность положительного прогноза, а для класса 0 – отрицательного.

Подробнее о прогнозе по модели логистической регрессии

Выше при заполнении таблицы со сводными характеристиками моделей мы воспользовались методом `logit.predict()`. В учебных целях воспроизведём вручную вычисления, чтобы проверить, что именно выдаёт нам эта функция.
Для начала на примере первой модели (`fit_logit_1`) убедимся, что прогнозы методом `logit.predict()` совпадают с прогнозами по формуле для вероятности логистической регрессии:

$$P(X) = \frac{e^{X \cdot \hat{\beta}}}{(1 + e^{X \cdot \hat{\beta}})}$$

Здесь X – матрица объясняющих переменных для модели с константой (n строк и $p + 1$ столбцов), $\hat{\beta}$ – вектор-столбец оценок параметров модели ($p + 1$ строк, 1 столбец); n – число наблюдений, p – количество объясняющих переменных модели.
Метод `logit.predict()` без аргументов возвращает прогнозы для наблюдений обучающей выборки. Чтобы сделать прогноз на тестовую, передадим ему в качестве аргумента фрейм с тестовой выборкой (`.predict(DF_test)`). Чтобы всё сработало, имена столбцов фреймов с обучающими и тестовыми наблюдениями должны совпадать.

In [27]:
```
# прогноз с помощью logit.predict()
y_hat_test_1 = fit_logit_1.predict(DF_test)
```

Теперь сделаем прогноз вручную, с помощью матричного умножения (функция `matmul()` из библиотеки `numpy`). Затем пересчитаем результат в вероятности.

In [28]:
```
# снова создаём матрицу объясняющих
#  для модели зависимости defaultYes от balance
X_test = DF_test.balance.values.reshape(-1, 1)
```

```
# прогноз вручную: y = X * beta, где X - матрица 2000 на 2,
#  а beta - вектор-столбец 2 на 1
y_hat_test = np.matmul(add_constant(X_test).reshape(-1, 2),
                       fit_logit_1.params.to_numpy().reshape(-1, 1))
# пересчитываем в вероятности
y_hat_test = np.exp(y_hat_test) / (1 + np.exp(y_hat_test))
```

Совместим результаты прогноза разными методами в одном фрейме.

In [29]:
```
pd.DataFrame({'Прогноз P(X) функцией predict': y_hat_test_1,
              'Прогноз P(X) вручную': y_hat_test.reshape(-1)})
```

Out[29]:

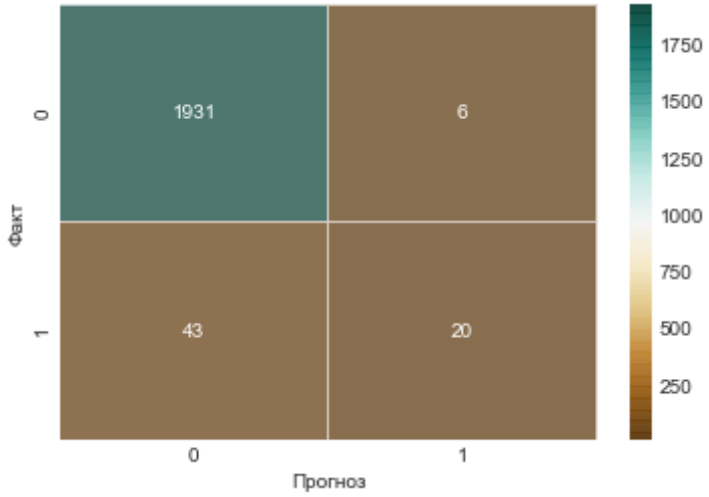
	Прогноз P(X) функцией predict	Прогноз P(X) вручную
1	0.002446	0.002446
4	0.002068	0.002068
6	0.002557	0.002557
9	0.000031	0.000031
12	0.000111	0.000111
...	...	...
9970	0.030396	0.030396
9971	0.000082	0.000082
9981	0.001616	0.001616
9983	0.000467	0.000467
9991	0.001052	0.001052

2000 rows × 2 columns

Как видно, прогнозы вероятностей полностью совпадают. Теперь получим из вероятностей метки классов, используя стандартный порог отсечения 0.5, и найдём прогнозные и фактические частоты классов в тестовой выборке.

In [30]:
```
# перекодируем в 0 и 1, граница отсечения 0.5
y_hat_test = (y_hat_test > 0.5).astype(int)
```

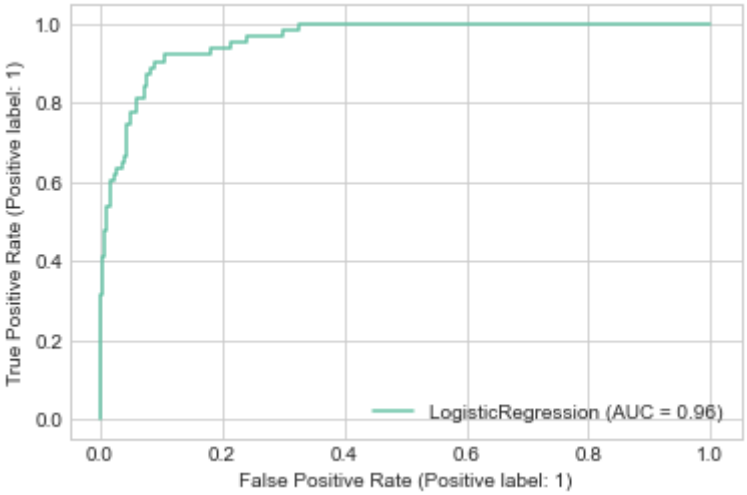
In [31]:
```
# считаем матрицу неточностей
cm = confusion_matrix(y_test, y_hat_test)
# рисуем матрицу в виде тепловой карты
sns.heatmap(cm, annot=True,
            fmt='g', linewidths=0.5, cmap='BrBG', alpha=0.7)
plt.ylabel('Факт')
plt.xlabel('Прогноз')
plt.show()
```



ROC-кривая и подбор порога отсечения

Чтобы настроить порог отсечения классов, построим для данной модели ROC-кривую.

In [32]:
```
# рисуем ROC-кривую
plot_roc_curve(fit_LR_2, X_test_LR_2, y_test)
plt.show()
```



Чтобы выбрать оптимальный порог отсечения формально, получим координаты ROC-кривой с помощью функции `roc_curve()`, а затем свернём их с помощью J-статистики Юдена, которая рассчитывается по формуле:

$$J = TPR + SPC - 1 = TPR - FPR$$

Когда чувствительность и специфичность модели максимальны, J-статистика также принимает максимальное значение.

In [33]:
```
# рассчитываем координаты ROC-кривой
fpr, tpr, thresholds = roc_curve(y_test, y_prob_test_LR_2)

# считаем J-статистику Юдена = TPR - FPR
```

```
j_scores = tpr - fpr

df_roc = pd.DataFrame({'TPR': tpr, 'SPC': 1-fpr,
                       'Порог отсечения': thresholds})
# находим оптимум по максимуму J-статистики
df_roc[j_scores == np.max(j_scores)]
```

Out[33]:

	TPR	SPC	Порог отсечения
93	0.777778	0.358286	0.009817

In [34]:

```
thr = df_roc[j_scores == np.max(j_scores)]['Порог отсечения'].to_numpy()
# прогноз с новой границей отсечения
y_hat_test = (y_prob_test_LR_2 > thr).astype(int)
# отчёт по точности на тестовой
print('Модель логистической регрессии от studentYes, balance с порогом',
      np.around(thr, 4), ': \n',
      classification_report(y_test, y_hat_test))
```

```
Модель логистической регрессии от studentYes, balance с порогом [0.0098] :
              precision    recall  f1-score   support

           0       0.98      0.36      0.52      1937
           1       0.04      0.76      0.07        63

    accuracy                           0.37      2000
   macro avg       0.51      0.56      0.30      2000
weighted avg       0.95      0.37      0.51      2000
```

Если сравнить с отчётом по модели с порогом отсечения 0.5, который был получен выше, можно увидеть, что чуда не произошло: ценность прогнозов (столбец `precision`) слегка повысилась, в то время как баланс между специфичностью и чувствительностью (столбец `recall`) сместился в пользу последней. Подбор порога отсечения позволяет вытянуть TPR в ущерб SPC или наоборот, но не то и другое одновременно.

Второй способ настройки порога отсечения – взять в качестве порога априорную долю наименьшего класса в обучающей выборке.

In [35]:

```
thr = np.min(DF_train.defaultYes.value_counts() / len(DF_train.index))
# прогноз с новой границей отсечения
y_hat_test = (y_prob_test_LR_2 > thr).astype(int)
# отчёт по точности на тестовой
print('Модель логистической регрессии от studentYes, balance с порогом',
      np.around(thr, 4), ': \n',
      classification_report(y_test, y_hat_test))
```

```
Модель логистической регрессии от studentYes, balance с порогом 0.0338 :
              precision    recall  f1-score   support

           0       0.98      0.42      0.59      1937
           1       0.04      0.67      0.07        63

    accuracy                           0.43      2000
   macro avg       0.51      0.55      0.33      2000
weighted avg       0.95      0.43      0.57      2000
```

Линейный дискриминантный анализ (LDA)

Для начала посмотрим на гистограммы распределения непрерывных объясняющих переменных и проверим их на нормальность.

In [36]:
```
# статистический тест Шапиро-Уилка на нормальность
#  для объясняющих переменных внутри классов
for col in ['balance', 'income']:
    stat, p = shapiro(DF_train[DF_train.defaultYes == 1][col])
    print(col, '| defaultYes : 1\n',
          'Statistics=%.2f, p=%.4f' % (stat, p))
    # интерпретация
    alpha = 0.05
    if p > alpha:
        print('Распределение нормально (H0 не отклоняется)\n')
    else:
        print('Распределение не нормально (H0 отклоняется)\n')

    stat, p = shapiro(DF_train[DF_train.defaultYes == 0][col])
    print(col, '| defaultYes : 0\n',
          'Statistics=%.2f, p=%.4f' % (stat, p))
    # интерпретация
    alpha = 0.05
    if p > alpha:
        print('Распределение нормально (H0 не отклоняется)\n')
    else:
        print('Распределение не нормально (H0 отклоняется)\n')
```

```
balance | defaultYes : 1
 Statistics=0.99, p=0.3875
Распределение нормально (H0 не отклоняется)

balance | defaultYes : 0
 Statistics=0.99, p=0.0000
Распределение не нормально (H0 отклоняется)

income | defaultYes : 1
 Statistics=0.95, p=0.0000
Распределение не нормально (H0 отклоняется)

income | defaultYes : 0
 Statistics=0.98, p=0.0000
Распределение не нормально (H0 отклоняется)
```

```
C:\Users\user\anaconda3\lib\site-packages\scipy\stats\morestats.py:1681: UserWar
ning: p-value may not be accurate for N > 5000.
  warnings.warn("p-value may not be accurate for N > 5000.")
C:\Users\user\anaconda3\lib\site-packages\scipy\stats\morestats.py:1681: UserWar
ning: p-value may not be accurate for N > 5000.
  warnings.warn("p-value may not be accurate for N > 5000.")
```

Применив функцию `shapiro()` к столбцам фрейма с обучающей выборкой, мы получили предупреждение о том, что рассчитанное p-значение может быть ненадёжно, поскольку наблюдений больше 5000. При этом p-значения говорят нам, что нормально распределён только показатель `balance` в классе `defaultYes = 1`. Проверим гипотезы о нормальности с помощью теста Лиллиефорса. Его мощность ниже, чем теста Шапиро, но у него нет ограничения на количество наблюдений.

In [37]:
```
# тест Лиллиефорса на нормальность
for col in ['balance', 'income']:
    stat, p = lilliefors(DF_train[DF_train.defaultYes == 1][col])
    print(col, '| defaultYes : 1\n',
          'Statistics=%.2f, p=%.4f' % (stat, p))
    # интерпретация
    alpha = 0.05
    if p > alpha:
        print('Распределение нормально (H0 не отклоняется)\n')
    else:
        print('Распределение не нормально (H0 отклоняется)\n')

    stat, p = lilliefors(DF_train[DF_train.defaultYes == 0][col])
    print(col, '| defaultYes : 0\n',
          'Statistics=%.2f, p=%.4f' % (stat, p))
    # интерпретация
    alpha = 0.05
    if p > alpha:
        print('Распределение нормально (H0 не отклоняется)\n')
    else:
        print('Распределение не нормально (H0 отклоняется)\n')
```

```
balance | defaultYes : 1
 Statistics=0.05, p=0.2469
Распределение нормально (H0 не отклоняется)

balance | defaultYes : 0
 Statistics=0.04, p=0.0010
Распределение не нормально (H0 отклоняется)

income | defaultYes : 1
 Statistics=0.11, p=0.0010
Распределение не нормально (H0 отклоняется)

income | defaultYes : 0
 Statistics=0.07, p=0.0010
Распределение не нормально (H0 отклоняется)
```

Итак, второй тест дал нам те же выводы о нормальности. Из этого следует, что допущения линейного дискриминантного анализа для наших данных не выполняются, и модель будет смещённой. **Далее построим модели LDA и QDA исключительно в учебных целях.**

Построим модель LDA на обучающей выборке, используя все непрерывные объясняющие переменные.

In [38]:
```
y_train = DF_train['defaultYes']
X_train = DF_train[['balance', 'income']]
X_train.head(5)
```

Out[38]:

	balance	income
627	622.929507	48874.555768
9748	280.461679	26212.907618
3937	883.016251	37961.155920
3108	766.124055	51614.056204

	balance	income
3160	1253.699614	37645.799959

In [39]:
```
# обучаем модель
fit_lda = LinearDiscriminantAnalysis().fit(X_train, y_train)

# прогноз на тестовую
X_test = DF_test[['balance', 'income']]
y_hat_test = fit_lda.predict(X_test)
```

In [40]:
```
# априорные вероятности классов
fit_lda.priors_
```

Out[40]:
```
array([0.96625, 0.03375])
```

In [41]:
```
# средние по классам
fit_lda.means_
```

Out[41]:
```
array([[  802.28360127, 33546.32043533],
       [ 1739.16671211, 32633.88861594]])
```

In [42]:
```
# отчёт по точности на тестовой
print('Модель LDA от balance, income: \n',
      classification_report(y_test, y_hat_test))
```

```
Модель LDA от balance, income:
               precision    recall  f1-score   support

           0       0.98      1.00      0.99      1937
           1       0.88      0.24      0.37        63

    accuracy                           0.97      2000
   macro avg       0.93      0.62      0.68      2000
weighted avg       0.97      0.97      0.97      2000
```

Квадратичный дискриминантный анализ (QDA)

Построим модель QDA на обучающей выборке, используя все непрерывные объясняющие переменные. Матрица X для обучающей и тестовой выборки останется такой же, как в модели LDA.

In [43]:
```
# обучаем модель
fit_qda = QuadraticDiscriminantAnalysis().fit(X_train, y_train)

# прогноз на тестовую
y_hat_test = fit_qda.predict(X_test)

# отчёт по точности на тестовой
print('Модель QDA от balance, income: \n',
      classification_report(y_test, y_hat_test))
```

```
Модель QDA от balance, income:
              precision    recall  f1-score   support

           0       0.98      1.00      0.99      1937
           1       0.84      0.25      0.39        63

    accuracy                           0.97      2000
   macro avg       0.91      0.63      0.69      2000
weighted avg       0.97      0.97      0.97      2000
```

Модели LDA и QDA оказались очень близки по точности на тестовой выборке.

Источники

1. *James G., Witten D., Hastie T. and Tibshirani R.* An Introduction to Statistical Learning with Applications in R. URL: http://www-bcf.usc.edu/~gareth/ISL/ISLR%20First%20Printing.pdf
2. *Jordi Warmenhoven* ISLR-python / github.com. URL: https://github.com/JWarmenhoven/ISLR-python
3. Logistic Regression in Python / realpython.com. URL: https://realpython.com/logistic-regression-python/
4. Руководство по библиотеке Seaborn / из курса "Python для анализа данных" от Физтеха. URL: https://mipt-stats.gitlab.io/courses/python/09_seaborn.html
5. *Tony S. Yu* Matplotlib Style Gallery / tonysyu.github.io. URL: http://tonysyu.github.io/raw_content/matplotlib-style-gallery/gallery.html
6. Intro to data structures / pandas.pydata.org. URL: https://pandas.pydata.org/pandas-docs/stable/user_guide/dsintro.html
7. *Baijayanta Roy* All about Categorical Variable Encoding / towardsdatascience.com. URL: https://towardsdatascience.com/all-about-categorical-variable-encoding-305f3361fd02