In [1]:
```
import pandas as pd
import numpy as np
from sklearn import model_selection
from sklearn import metrics
from datetime import datetime
import sklearn.tree as tree
from sklearn.linear_model import LogisticRegression
import copy
from sklearn.base import BaseEstimator, ClassifierMixin
import random
```

Lazy FCA

Оформим алгоритм `Lazy FCA` как класс, реализующий интерфейс пакета `sklearn` для ML моделей.

In [2]:
```
class LazyFCA(BaseEstimator, ClassifierMixin):
    def __init__(
        self, threshold=0.5,
        random=False, sample_share=0.5,
        bias='random', random_seed=None):

        self.threshold = threshold
        self.random = random
        self.sample_share = sample_share
        self.bias = bias
        self.random_seed = random_seed
        self.binary_mapping = dict()

    def fit(self, X, y):
        pd.options.mode.chained_assignment = None
        X = self.scaled_X(X)
        y = self.scaled_y(y)
        self.positive_sample = X[y == 1]
        self.negative_sample = X[y == 0]

        if self.random:
            sample_size = int(self.sample_share * self.positive_sample.shape[0])
            self.positive_sample = self.positive_sample.sample(
                    n=sample_size, random_state=self.random_seed)
            self.negative_sample = self.negative_sample.sample(
                    n=sample_size, random_state=self.random_seed)

        self.positive_obj = {}
        self.negative_obj = {}
        pos = self.positive_sample
        neg = self.negative_sample
        for i_col in X.columns:
            self.positive_obj[i_col] = pos[i_col][pos[i_col] == 1].index
            self.negative_obj[i_col] = neg[i_col][neg[i_col] == 1].index

    def predict(self, X):
        pd.options.mode.chained_assignment = None
        random.seed(self.random_seed)
        X = self.scaled_X(X)
        predictions = []
        for i_obj in range(X.shape[0]):
```

```
            i_extent = self.extent(X.iloc[i_obj])
            support_pos  = self.calculate_support(i_extent, 'positive')
            support_neg  = self.calculate_support(i_extent, 'negative')

            if support_neg == support_pos:
                if self.bias == 'random':
                    prediction = random.choice([True, False])
                elif self.bias == 'positive':
                    prediction = True
                else:
                    prediction = False
            else:
                prediction = support_pos > support_neg
            predictions.append(self.binary_mapping[prediction])
        return predictions

    def scaled_X(self, X_dataset):
        intervals = 5
        for i_col in X_dataset.columns:
            values = list(X_dataset[i_col].unique())

            if len(values) == 2 and 0 in values and 1 in values:
                continue
            elif len(values) == 1 and (0 in values or 1 in values):
                continue

            elif len(values) <= 2 or X_dataset[i_col].dtypes == np.dtype('O'):
                values = sorted(list(X_dataset[i_col].unique()))
                for i_val in values:
                    X_dataset['{}_{}'.format(i_col, i_val)]\
                        = (X_dataset[i_col] == i_val).astype(int)

            elif X_dataset[i_col].dtype == np.dtype('int64'):
                min_val = X_dataset[i_col].min()
                max_val = X_dataset[i_col].max()
                gap = max_val - min_val
                start = min_val + gap / intervals
                finish = max_val - gap / intervals
                k = 0
                for i in np.linspace(start, finish, intervals):
                    X_dataset['{}_{}'.format(i_col, k)]\
                        = (X_dataset[i_col] >= i).astype(int)
                    k += 1

            X_dataset.drop([i_col], axis=1, inplace = True)
        return X_dataset

    def scaled_y(self, y_series):
        values = sorted(y_series.unique())
        if len(values) != 2:
            raise Exception('Only a binary target feature is possible')
        self.binary_mapping[False] = values[0]
        self.binary_mapping[True] = values[1]
        return (y_series == values[1]).astype(int)

    def calculate_support(self, obj_ext, base):

        base_sample = (self.positive_sample if base == 'positive'
                else self.negative_sample)
        review_sample = (self.negative_sample if base == 'positive'
                else self.positive_sample)
```

```
        review_obj = (self.negative_obj if base == 'positive'
                else self.positive_obj)

        res = 0
        for _, i_obj in base_sample.iterrows():
            i_inters = self.intersection(
                obj_ext, self.extent(i_obj))
            support_card = 0
            if i_inters:
                support = review_obj[i_inters[0]]
                for i_col in i_inters:
                    support = self.intersection(support, review_obj[i_col])
                    if not support: break
                support_card = len(support) / review_sample.shape[0]
                if support_card < self.threshold:
                    res += len(i_inters) / len(obj_ext)

        res = res / base_sample.shape[0]
        return res

    def extent(self, series):
        return series[series == 1].index.tolist()

    def intersection(self, L, R):
        return [val for val in L if val in R]

    def belongs(self, sub, base):
        return len(self.intersection(sub, base)) == len(sub)
```

Tic-Tac-Toe Dataset

Функция шкалирования для датасета по крестикам-ноликам

In [3]:
```
def scale(dataset):
    for i in range(9):
        str_i = str(i + 1)
        dataset['v' + str_i] = (dataset['V' + str_i] == 'x').astype(int)
    dataset['v10'] = (dataset['V10'] == 'positive').astype(int)
    dataset.drop(['V' + str(i+1) for i in range(10)], axis=1, inplace = True)
    return dataset
```

Функция тренерующая переданную модель `model` на датасете по крестикам-ноликам и вычисляющая точность предсказаний полученной модели.

In [4]:
```
def tic_tac_toe(model, progress_bar=False):
    results = {'accuracy': [], 'precision': [], 'recall': [], 'f1': [], 'seconds

    for i in range(10):
        if progress_bar:
            print(f'Progress: {i + 1} / 10')

        train_data = scale(pd.read_csv(f'tic-tac-toe/train{i + 1}.csv'))
        X_train = train_data.iloc[:, :-1]
        y_train = train_data.iloc[:, -1]

        model.fit(X_train, y_train)

        test_data = scale(pd.read_csv(f'tic-tac-toe/test{i + 1}.csv'))
```

```
        X_test = test_data.iloc[:, :-1]
        y_test = test_data.iloc[:, -1]

        s = datetime.now()
        y_pred = model.predict(X_test)
        f = datetime.now()

        results['accuracy'].append(metrics.accuracy_score(y_test, y_pred))
        results['precision'].append(metrics.precision_score(y_test, y_pred))
        results['recall'].append(metrics.recall_score(y_test, y_pred))
        results['f1'].append(metrics.f1_score(y_test, y_pred))

        results['seconds'].append((f - s).seconds)

    return pd.DataFrame(results)
```

Lazy FCA

Начнем с `Lazy FCA` натренерованной на $\frac{1}{5}$ всего датасета.

In [5]:

```
model = LazyFCA(
    threshold=0.000001, bias='negative',
    random=True, sample_share=0.2, random_seed=1)
tic_tac_toe(model)
```

Out[5]:

	accuracy	precision	recall	f1	seconds
0	1.000000	1.000000	1.0	1.000000	7
1	0.988506	0.980769	1.0	0.990291	7
2	0.990000	0.984848	1.0	0.992366	8
3	0.966292	0.951613	1.0	0.975207	8
4	0.988764	0.984127	1.0	0.992000	7
5	0.988235	0.982456	1.0	0.991150	6
6	0.973684	0.958904	1.0	0.979021	10
7	1.000000	1.000000	1.0	1.000000	9
8	1.000000	1.000000	1.0	1.000000	9
9	0.989011	0.983333	1.0	0.991597	6

Теперь посмотрим на `Lazy FCA` на полном датасете.

In [6]:

```
model = LazyFCA(threshold=0.000001, bias='negative')
tic_tac_toe(model)
```

Out[6]:

	accuracy	precision	recall	f1	seconds
0	1.0	1.0	1.0	1.0	35
1	1.0	1.0	1.0	1.0	37
2	1.0	1.0	1.0	1.0	40
3	1.0	1.0	1.0	1.0	34
4	1.0	1.0	1.0	1.0	47

	accuracy	precision	recall	f1	seconds
5	1.0	1.0	1.0	1.0	33
6	1.0	1.0	1.0	1.0	40
7	1.0	1.0	1.0	1.0	43
8	1.0	1.0	1.0	1.0	36
9	1.0	1.0	1.0	1.0	34

Decision Tree

Сравним результаты `Lazy FCA` с классической моделью `Decision Tree`

In [7]:
```
model = tree.DecisionTreeClassifier(criterion='entropy')

tic_tac_toe(model)
```

Out[7]:

	accuracy	precision	recall	f1	seconds
0	0.989247	1.000000	0.983607	0.991736	0
1	0.954023	0.979592	0.941176	0.960000	0
2	0.990000	0.984848	1.000000	0.992366	0
3	0.988764	1.000000	0.983051	0.991453	0
4	0.988764	1.000000	0.983871	0.991870	0
5	0.988235	0.982456	1.000000	0.991150	0
6	1.000000	1.000000	1.000000	1.000000	0
7	0.971963	0.972973	0.986301	0.979592	0
8	0.990291	1.000000	0.985714	0.992806	0
9	0.989011	1.000000	0.983051	0.991453	0

Даже модель, натренированная на 20% от всех данных оказалась лучше дерева решений, а полная модель достигла абсолютной точности.

Titanic Dataset

Рассмотрим теперь работу `Lazy FCA` на знаменитом датасете - данных о смертности пассажиров Титаника, и сравним полученную точность с точностью логистической регрессии.

In [8]:
```
titanic_data = pd.read_csv('titanic/train.csv')\
    .drop(columns=['Name', 'Ticket', 'PassengerId', 'Cabin'])\
    .dropna()\
    .rename(columns={"Survived": "target"})
titanic_data
```

Out[8]:

	target	Pclass	Sex	Age	SibSp	Parch	Fare	Embarked
0	0	3	male	22.0	1	0	7.2500	S
1	1	1	female	38.0	1	0	71.2833	C

	target	Pclass	Sex	Age	SibSp	Parch	Fare	Embarked
2	1	3	female	26.0	0	0	7.9250	S
3	1	1	female	35.0	1	0	53.1000	S
4	0	3	male	35.0	0	0	8.0500	S
...	...	...	...	...	...	...	...	...
885	0	3	female	39.0	0	5	29.1250	Q
886	0	2	male	27.0	0	0	13.0000	S
887	1	1	female	19.0	0	0	30.0000	S
889	1	1	male	26.0	0	0	30.0000	C
890	0	3	male	32.0	0	0	7.7500	Q

712 rows × 8 columns

Шкалирование численных и категориальных данных. Численные разбиваются на `intervals` равных интервалов и для каждого интервала создается своя фича. В категориальных данных для каждой категории создается своя фича.

In [9]:

```
def scaling(data, numeric, categorical, intervals=5):
    for attr in numeric:
        min_val = data[attr].min()
        max_val = data[attr].max()
        gap = max_val - min_val
        k = 0
        for i in np.linspace(min_val + gap / intervals, max_val - gap / interval
            data[attr + '_' + str(k)] = (data[attr] >= i).astype(int)
            k += 1
        data = data.drop(attr, axis=1)

    for attr in categorical:
        for i in data[attr].unique():
            data[attr + '_' + str(i)] = (data[attr] == i).astype(int)
        data = data.drop(attr, axis=1)
    return data
```

In [10]:

```
titanic_data = scaling(titanic_data, numeric=['Age', 'Fare'], categorical=['Pcla
titanic_data
```

Out[10]:

	target	Age_0	Age_1	Age_2	Age_3	Age_4	Fare_0	Fare_1	Fare_2	Fare_3	...	Parch_0	Par
0	0	1	0	0	0	0	0	0	0	0	...	1	
1	1	1	1	0	0	0	0	0	0	0	...	1	
2	1	1	0	0	0	0	0	0	0	0	...	1	
3	1	1	1	0	0	0	0	0	0	0	...	1	
4	0	1	1	0	0	0	0	0	0	0	...	1	
...	...	...	...	...	...	...	...	...	...	...	...	...	
885	0	1	1	0	0	0	0	0	0	0	...	0	
886	0	1	0	0	0	0	0	0	0	0	...	1	
887	1	1	0	0	0	0	0	0	0	0	...	1	

	target	Age_0	Age_1	Age_2	Age_3	Age_4	Fare_0	Fare_1	Fare_2	Fare_3	...	Parch_0	Par
889	1	1	0	0	0	0	0	0	0	0	...	1	
890	0	1	1	0	0	0	0	0	0	0	...	1	

712 rows × 32 columns

Функция тренерующая переданную модель `model` на датасете по пассажирам титаника и вычисляющая точность предсказаний полученной модели.

In [11]:
```
def titanc(model, progress_bar=False):
    columns = list(titanic_data.columns)
    columns.remove('target')

    X = titanic_data.loc[:, columns]
    y = titanic_data.target

    results = {'accuracy': [], 'precision': [], 'recall': [], 'f1': [], 'seconds
    for k in range(10):
        if progress_bar:
            print(f'Progress: {k + 1} / 10')
        X_train, X_test, y_train, y_test = model_selection\
            .train_test_split(X, y, test_size=0.33, random_state=k)

        model.fit(X_train, y_train)

        s = datetime.now()
        y_pred = model.predict(X_test)
        f = datetime.now()

        results['accuracy'].append(metrics.accuracy_score(y_test, y_pred))
        results['precision'].append(metrics.precision_score(y_test, y_pred))
        results['recall'].append(metrics.recall_score(y_test, y_pred))
        results['f1'].append(metrics.f1_score(y_test, y_pred))

        results['seconds'].append((f - s).seconds)

    return pd.DataFrame(results)
```

Lazy classification

Найдем сначала перебором лучшие параметры для нашей модели, используя 10% от всего датасета при тренеровке моделей.

In [12]:
```
for i in ['random', 'positive', 'negative']:
    for j in np.linspace(0.1, 0.9, 5):
        model = LazyFCA(threshold=j, bias=i, random=True, sample_share=0.1)
        res = titanc(model)
        print()
        print('Parameters:', model)
        print()

        print(res)
        print()
        print('F1:', res['f1'].mean())
```

```
Parameters: LazyFCA(random=True, sample_share=0.1, threshold=0.1)
```

	accuracy	precision	recall	f1	seconds
0	0.702128	0.609524	0.688172	0.646465	3
1	0.527660	0.458101	0.854167	0.596364	2
2	0.761702	0.701149	0.670330	0.685393	2
3	0.791489	0.772152	0.663043	0.713450	3
4	0.791489	0.773333	0.644444	0.703030	3
5	0.574468	0.459459	0.772727	0.576271	3
6	0.748936	0.672897	0.750000	0.709360	2
7	0.634043	0.537879	0.739583	0.622807	2
8	0.748936	0.679612	0.729167	0.703518	2
9	0.591489	0.465753	0.790698	0.586207	3

F1: 0.6542864431070503

Parameters: LazyFCA(random=True, sample_share=0.1, threshold=0.30000000000000004)

	accuracy	precision	recall	f1	seconds
0	0.740426	0.710526	0.580645	0.639053	3
1	0.719149	0.750000	0.468750	0.576923	3
2	0.714894	0.671429	0.516484	0.583851	2
3	0.710638	0.785714	0.358696	0.492537	3
4	0.765957	0.721519	0.633333	0.674556	3
5	0.778723	0.781250	0.568182	0.657895	4
6	0.697872	0.790698	0.354167	0.489209	3
7	0.787234	0.787500	0.656250	0.715909	3
8	0.774468	0.747126	0.677083	0.710383	3
9	0.787234	0.743243	0.639535	0.687500	3

F1: 0.6227815763994476

Parameters: LazyFCA(random=True, sample_share=0.1)

	accuracy	precision	recall	f1	seconds
0	0.765957	0.816667	0.526882	0.640523	2
1	0.765957	0.741176	0.656250	0.696133	2
2	0.719149	0.755102	0.406593	0.528571	2
3	0.761702	0.764706	0.565217	0.650000	3
4	0.702128	0.777778	0.311111	0.444444	2
5	0.702128	0.781250	0.284091	0.416667	3
6	0.680851	0.714286	0.364583	0.482759	3
7	0.770213	0.800000	0.583333	0.674699	2
8	0.736170	0.869565	0.416667	0.563380	3
9	0.808511	0.815385	0.616279	0.701987	3

F1: 0.5799162464712022

Parameters: LazyFCA(random=True, sample_share=0.1, threshold=0.7000000000000001)

	accuracy	precision	recall	f1	seconds
0	0.731915	0.800000	0.430108	0.559441	2
1	0.740426	0.761194	0.531250	0.625767	3
2	0.736170	0.837209	0.395604	0.537313	3
3	0.714894	0.662338	0.554348	0.603550	2
4	0.774468	0.803279	0.544444	0.649007	3
5	0.778723	0.810345	0.534091	0.643836	3
6	0.753191	0.851852	0.479167	0.613333	2
7	0.778723	0.814286	0.593750	0.686747	3
8	0.689362	0.848485	0.291667	0.434109	3
9	0.787234	0.846154	0.511628	0.637681	3

F1: 0.5990783406092139

Parameters: LazyFCA(random=True, sample_share=0.1, threshold=0.9)

	accuracy	precision	recall	f1	seconds
0	0.685106	0.594059	0.645161	0.618557	2
1	0.676596	0.574627	0.802083	0.669565	3
2	0.744681	0.670330	0.670330	0.670330	3
3	0.770213	0.771429	0.586957	0.666667	3
4	0.787234	0.756410	0.655556	0.702381	3
5	0.804255	0.783784	0.659091	0.716049	3
6	0.740426	0.649573	0.791667	0.713615	3
7	0.765957	0.715789	0.708333	0.712042	3
8	0.731915	0.761905	0.500000	0.603774	3
9	0.791489	0.760563	0.627907	0.687898	3

F1: 0.6760877172884138

Parameters: LazyFCA(bias='positive', random=True, sample_share=0.1, threshold=0.
1)

	accuracy	precision	recall	f1	seconds
0	0.770213	0.709677	0.709677	0.709677	3
1	0.736170	0.634921	0.833333	0.720721	3
2	0.757447	0.663462	0.758242	0.707692	3
3	0.574468	0.478022	0.945652	0.635036	3
4	0.587234	0.477707	0.833333	0.607287	4
5	0.621277	0.496503	0.806818	0.614719	3
6	0.625532	0.530769	0.718750	0.610619	3
7	0.710638	0.609375	0.812500	0.696429	3
8	0.570213	0.484076	0.791667	0.600791	3
9	0.740426	0.631579	0.697674	0.662983	3

F1: 0.6565954987933036

Parameters: LazyFCA(bias='positive', random=True, sample_share=0.1,
threshold=0.30000000000000004)

	accuracy	precision	recall	f1	seconds
0	0.723404	0.818182	0.387097	0.525547	2
1	0.761702	0.794118	0.562500	0.658537	3
2	0.761702	0.796610	0.516484	0.626667	3
3	0.744681	0.758065	0.510870	0.610390	3
4	0.787234	0.900000	0.500000	0.642857	2
5	0.740426	0.754717	0.454545	0.567376	4
6	0.761702	0.750000	0.625000	0.681818	3
7	0.817021	0.884058	0.635417	0.739394	3
8	0.774468	0.772152	0.635417	0.697143	3
9	0.723404	0.652174	0.523256	0.580645	3

F1: 0.6330373476704871

Parameters: LazyFCA(bias='positive', random=True, sample_share=0.1)

	accuracy	precision	recall	f1	seconds
0	0.744681	0.732394	0.559140	0.634146	3
1	0.731915	0.779661	0.479167	0.593548	3
2	0.719149	0.745098	0.417582	0.535211	3
3	0.795745	0.923077	0.521739	0.666667	3
4	0.812766	0.896552	0.577778	0.702703	3
5	0.770213	0.869565	0.454545	0.597015	5
6	0.608511	0.600000	0.125000	0.206897	5
7	0.748936	0.803279	0.510417	0.624204	5
8	0.740426	0.818182	0.468750	0.596026	4
9	0.731915	0.629213	0.651163	0.640000	5

F1: 0.579641715435474

```
Parameters: LazyFCA(bias='positive', random=True, sample_share=0.1,
        threshold=0.7000000000000001)

   accuracy  precision    recall        f1  seconds
0  0.757447   0.810345  0.505376  0.622517        3
1  0.740426   0.727273  0.583333  0.647399        3
2  0.719149   0.698413  0.483516  0.571429        3
3  0.748936   0.726027  0.576087  0.642424        2
4  0.731915   0.846154  0.366667  0.511628        2
5  0.757447   0.682353  0.659091  0.670520        3
6  0.736170   0.688889  0.645833  0.666667        3
7  0.731915   0.663366  0.697917  0.680203        2
8  0.770213   0.769231  0.625000  0.689655        2
9  0.706383   1.000000  0.197674  0.330097        2

F1: 0.6032538324409837

Parameters: LazyFCA(bias='positive', random=True, sample_share=0.1, threshold=0.
9)

   accuracy  precision    recall        f1  seconds
0  0.731915   0.674419  0.623656  0.648045        3
1  0.723404   0.829787  0.406250  0.545455        3
2  0.740426   0.720588  0.538462  0.616352        3
3  0.731915   0.655914  0.663043  0.659459        4
4  0.770213   0.772727  0.566667  0.653846        3
5  0.757447   0.792453  0.477273  0.595745        3
6  0.740426   0.872340  0.427083  0.573427        3
7  0.753191   0.827586  0.500000  0.623377        3
8  0.782979   0.784810  0.645833  0.708571        3
9  0.774468   0.708861  0.651163  0.678788        3

F1: 0.6303064237769018

Parameters: LazyFCA(bias='negative', random=True, sample_share=0.1, threshold=0.
1)

   accuracy  precision    recall        f1  seconds
0  0.740426   0.645455  0.763441  0.699507        2
1  0.740426   0.710843  0.614583  0.659218        3
2  0.736170   0.655914  0.670330  0.663043        4
3  0.608511   0.500000  0.728261  0.592920        5
4  0.753191   0.677778  0.677778  0.677778        4
5  0.770213   0.697674  0.681818  0.689655        4
6  0.765957   0.730337  0.677083  0.702703        4
7  0.689362   0.596639  0.739583  0.660465        4
8  0.710638   0.620690  0.750000  0.679245        3
9  0.693617   0.568627  0.674419  0.617021        5

F1: 0.664155642728866

Parameters: LazyFCA(bias='negative', random=True, sample_share=0.1,
        threshold=0.30000000000000004)

   accuracy  precision    recall        f1  seconds
0  0.727660   0.745763  0.473118  0.578947        3
1  0.727660   0.750000  0.500000  0.600000        4
2  0.736170   0.745763  0.483516  0.586667        4
3  0.757447   0.746479  0.576087  0.650307        2
4  0.787234   0.777778  0.622222  0.691358        3
5  0.782979   0.784615  0.579545  0.666667        3
6  0.748936   0.717647  0.635417  0.674033        3
7  0.672340   0.581197  0.708333  0.638498        3
8  0.778723   0.823529  0.583333  0.682927        3
9  0.753191   0.818182  0.418605  0.553846        4
```

```
F1: 0.632324925977988

Parameters: LazyFCA(bias='negative', random=True, sample_share=0.1)

   accuracy  precision    recall        f1  seconds
0  0.727660   0.745763  0.473118  0.578947        3
1  0.757447   0.767123  0.583333  0.662722        3
2  0.719149   0.650602  0.593407  0.620690        3
3  0.719149   0.770833  0.402174  0.528571        3
4  0.689362   0.774194  0.266667  0.396694        3
5  0.787234   0.839286  0.534091  0.652778        3
6  0.706383   0.714286  0.468750  0.566038        3
7  0.778723   0.768293  0.656250  0.707865        3
8  0.778723   0.866667  0.541667  0.666667        3
9  0.782979   0.888889  0.465116  0.610687        3

F1: 0.5991658932265643

Parameters: LazyFCA(bias='negative', random=True, sample_share=0.1,
        threshold=0.7000000000000001)

   accuracy  precision    recall        f1  seconds
0  0.770213   0.729412  0.666667  0.696629        3
1  0.774468   0.830769  0.562500  0.670807        3
2  0.723404   0.732143  0.450549  0.557823        3
3  0.702128   0.610000  0.663043  0.635417        3
4  0.774468   0.836364  0.511111  0.634483        3
5  0.761702   0.728571  0.579545  0.645570        3
6  0.744681   0.781250  0.520833  0.625000        3
7  0.791489   0.840580  0.604167  0.703030        3
8  0.748936   0.776119  0.541667  0.638037        3
9  0.795745   0.779412  0.616279  0.688312        4

F1: 0.6495107642849458

Parameters: LazyFCA(bias='negative', random=True, sample_share=0.1, threshold=0.
9)

   accuracy  precision    recall        f1  seconds
0  0.685106   0.806452  0.268817  0.403226        3
1  0.761702   0.750000  0.625000  0.681818        3
2  0.757447   0.702381  0.648352  0.674286        3
3  0.663830   0.561905  0.641304  0.598985        3
4  0.753191   0.695122  0.633333  0.662791        3
5  0.753191   0.666667  0.681818  0.674157        3
6  0.736170   0.803571  0.468750  0.592105        3
7  0.791489   0.747368  0.739583  0.743455        3
8  0.727660   0.750000  0.500000  0.600000        3
9  0.740426   0.671233  0.569767  0.616352        4

F1: 0.6247175436972273
```

Лучшими по F_1 метрике оказались параметры $\texttt{bias} = \texttt{random}$ и $\texttt{threshold} = 0.9$.

Запустим нашу модель на всех данных, используя эти параметры.

In [15]:
```python
model = LazyFCA(threshold=0.9, bias='random')
res = titanc(model)

print(res)
print()
print('F1:', res['f1'].mean())
```

```
   accuracy  precision    recall        f1  seconds
```

```
0  0.765957   0.796875  0.548387  0.649682       52
1  0.774468   0.811594  0.583333  0.678788       51
2  0.761702   0.753623  0.571429  0.650000       61
3  0.761702   0.750000  0.586957  0.658537       58
4  0.787234   0.794118  0.600000  0.683544       70
5  0.787234   0.779412  0.602273  0.679487       58
6  0.748936   0.793651  0.520833  0.628931       56
7  0.795745   0.852941  0.604167  0.707317       52
8  0.774468   0.786667  0.614583  0.690058       62
9  0.812766   0.850000  0.593023  0.698630       57

F1: 0.672497398340006
```

Logistic regression

Сравним теперь результаты `Lazy FCA` с классической моделью логистической регрессии.

In [16]:
```
model = LogisticRegression(solver='lbfgs', random_state=0)
res = titanc(model)

print(res)
print()
print('F1:', res['f1'].mean())
```

```
   accuracy  precision    recall        f1  seconds
0  0.753191   0.684211  0.698925  0.691489        0
1  0.770213   0.710000  0.739583  0.724490        0
2  0.765957   0.691489  0.714286  0.702703        0
3  0.778723   0.738095  0.673913  0.704545        0
4  0.791489   0.735632  0.711111  0.723164        0
5  0.787234   0.711111  0.727273  0.719101        0
6  0.748936   0.703297  0.666667  0.684492        0
7  0.808511   0.800000  0.708333  0.751381        0
8  0.787234   0.734694  0.750000  0.742268        0
9  0.795745   0.720930  0.720930  0.720930        0

F1: 0.716456374814656
```

К сожалению, наш алгоритм, даже использующий весь датасет, не смог обойти по F_1 метрике логистическую регрессию.