AIED - Uczenie maszynowe

Fraud detection - transakcje bankowe

Dla zbioru danych `creditcard.csv` nie udało się zapewnić stopy błędów FPR na poziomie poniżej 0.5% w żadnym przypadku analizy. Zamiast tego analiza skupiała się na jej minimalizowaniu i jednoczesnym utrzymywaniu stopy błędów FNR na jak najniższym poziomie.

Dane wejściowe - wczytanie i analiza

In [ ]:
```
import pandas as pd

dataframe = pd.read_csv("./data/creditcard.csv")
```

Rozmiar danych wejściowych i podstawowe statystyki

In [ ]:
```
dataframe.shape
```

Out[ ]:
```
(284807, 31)
```

In [ ]:
```
dataframe.describe()
```

Out[ ]:

	Time	V1	V2	V3	V4	V5
count	284807.000000	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05
mean	94813.859575	1.168375e-15	3.416908e-16	-1.379537e-15	2.074095e-15	9.604066e-16
std	47488.145955	1.958696e+00	1.651309e+00	1.516255e+00	1.415869e+00	1.380247e+00
min	0.000000	-5.640751e+01	-7.271573e+01	-4.832559e+01	-5.683171e+00	-1.137433e+02
25%	54201.500000	-9.203734e-01	-5.985499e-01	-8.903648e-01	-8.486401e-01	-6.915971e-01
50%	84692.000000	1.810880e-02	6.548556e-02	1.798463e-01	-1.984653e-02	-5.433583e-02
75%	139320.500000	1.315642e+00	8.037239e-01	1.027196e+00	7.433413e-01	6.119264e-01
max	172792.000000	2.454930e+00	2.205773e+01	9.382558e+00	1.687534e+01	3.480167e+01

8 rows × 31 columns

In [ ]:
```
dataframe['Class'].value_counts()
```

Out[ ]:
```
0    284315
1       492
Name: Class, dtype: int64
```

Standaryzacja danych

Usunięcie średniej i normalizacja odchylenia standardowego. Kolumny klasy i czasu nie są standaryzowane.

In [ ]:
```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
featureCols = dataframe.columns.difference(['Class'], sort=False)
dataframe[featureCols] = scaler.fit_transform(dataframe[featureCols])
print("std:", dataframe[featureCols].stack().std())
print("mean:", dataframe[featureCols].stack().mean())
dataframe
```

```
std: 1.000000058519166
mean: -4.657000846487342e-19
```

Out[ ]:

	Time	V1	V2	V3	V4	V5	V6	V7	
0	-1.996583	-0.694242	-0.044075	1.672773	0.973366	-0.245117	0.347068	0.193679	0.(
1	-1.996583	0.608496	0.161176	0.109797	0.316523	0.043483	-0.061820	-0.063700	0.(
2	-1.996562	-0.693500	-0.811578	1.169468	0.268231	-0.364572	1.351454	0.639776	0.:
3	-1.996562	-0.493325	-0.112169	1.182516	-0.609727	-0.007469	0.936150	0.192071	0.:
4	-1.996541	-0.591330	0.531541	1.021412	0.284655	-0.295015	0.071999	0.479302	-0.:
...	...	...	...	...	...	...	...	...	
284802	1.641931	-6.065842	6.099286	-6.486245	-1.459641	-3.886611	-1.956690	-3.975628	6.
284803	1.641952	-0.374121	-0.033356	1.342145	-0.521651	0.629040	0.794446	0.019667	0.:
284804	1.641974	0.980024	-0.182434	-2.143205	-0.393984	1.905833	2.275262	-0.239939	0.!
284805	1.641974	-0.122755	0.321250	0.463320	0.487192	-0.273836	0.468155	-0.554672	0.!
284806	1.642058	-0.272331	-0.114899	0.463866	-0.357570	-0.009089	-0.487602	1.274769	-0.:

284807 rows × 31 columns

Usunięcie duplikatów

In [ ]:
```
dataframe.drop_duplicates(keep='first', inplace=True)
dataframe.shape
```

Out[ ]:
```
(283726, 31)
```

Podział na klasy terningowe i testowe

In [ ]:
```
from sklearn.model_selection import train_test_split

X = dataframe.drop(['Class'], axis=1)
y = dataframe.Class

X_train, X_test, y_train, y_test = train_test_split(
    X, y, random_state=0, test_size=0.20)

print("train rows: {}, test rows: {}".format(
    X_train.shape[0], X_test.shape[0]))  # rows
```

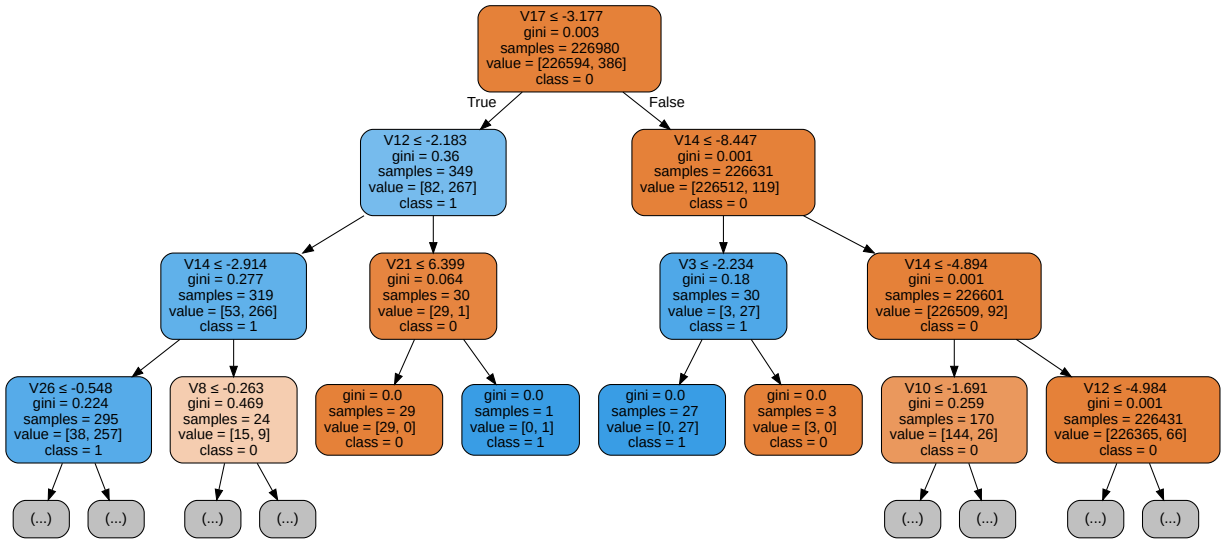
```
train rows: 226980, test rows: 56746
```

Drzewo decyzyjne

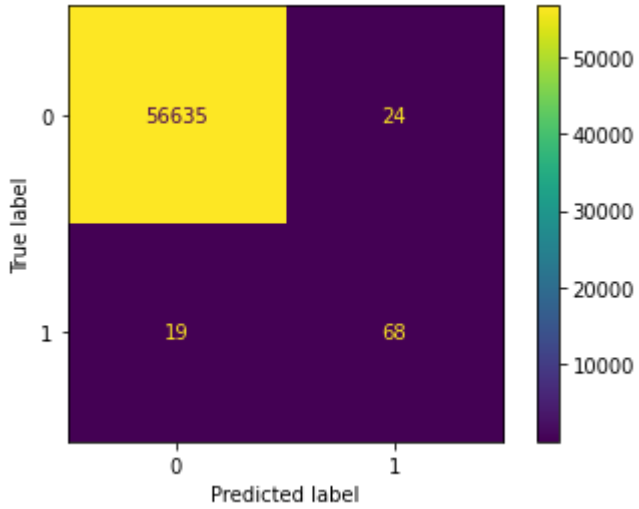
1. Wariant podstawowy

In [ ]:
```
from helpers import computeDecitionTree

y_pred, graph = computeDecitionTree(
    X_train, X_test, y_train)
graph
```

Out[ ]:



In [ ]:
```
from helpers import analyse
accuracy, sensitivity, specificity = analyse(y_test, y_pred, True)
```




```
accuracy = 0.9992, sensitivity = 0.7816, specificity = 0.9996
```

Wnioski

- wysoka dokładność nie oznacza poprawnego działania w przypadku niezbalansowanych danych
- stosunkowo niska czułość (77%) wskazuje na duży odsetek fałszywie negatywnych klasyfikacji

2. Parametry klasyfikatora

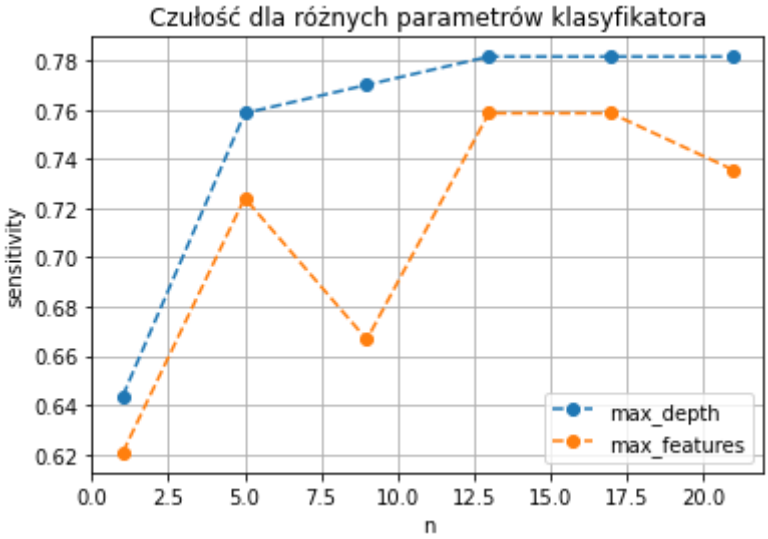
Głębokość i liczba cech

In [ ]:
```
r = range(1, 24, 4)
depth_sensitivities = []
for depth in r:
    y_pred, graph = computeDecitionTree(
        X_train, X_test, y_train, max_depth=depth)
```

```
    accuracy, sensitivity, specificity = analyse(y_test, y_pred)
    depth_sensitivities.append(sensitivity)

features_sensitivities = []
for features in r:
    y_pred, graph = computeDecitionTree(
        X_train, X_test, y_train, max_features=features)
    accuracy, sensitivity, specificity = analyse(y_test, y_pred)
    features_sensitivities.append(sensitivity)
```

In [ ]:

```
import matplotlib.pyplot as plt
import numpy as np

plot_style = {"marker": 'o', "linestyle": '--'}

fig, ax = plt.subplots()

ax.plot(r, depth_sensitivities, label="max_depth", **plot_style)
ax.plot(r, features_sensitivities, label="max_features", **plot_style)

ax.set(xlabel='n', ylabel='sensitivity',
       title='Czułość dla różnych parametrów klasyfikatora')
ax.grid()
ax.legend()

plt.show()
print("max depth sensitivity: {}, max features sensitivity: {}".format(
    max(depth_sensitivities), max(features_sensitivities)))
```

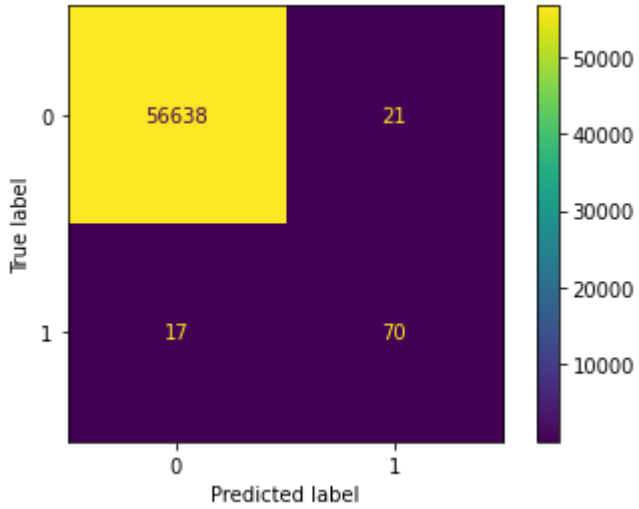



```
max depth sensitivity: 0.7816091954022989, max features sensitivity: 0.758620
6896551724
```

Kryterium podziału

In [ ]:

```
y_pred, graph = computeDecitionTree(
    X_train, X_test, y_train, criterion='entropy')
accuracy, sensitivity, specificity = analyse(y_test, y_pred, True)
```




```
accuracy = 0.9993, sensitivity = 0.8046, specificity = 0.9996
```

Wnioski

- Parametry klasyfikatora wpływają na jego czułość i zależą bezpośrednio od jego typu
- Dla przykładu kart kredytowych wartość parametrów głębokości i liczby cech dających największą czułość to 13. Potem czułość klasyfikatora stabilizuje się na określonynm poziomie.
- Ręczne ustawienie parametrów daje porównywalną czułość, jak ustawienia i optymalizacje domyślne (80% vs 78%).
- Zmiana kryterium na entropię delikatnie pomaga zwiekszyć czułość

3. Selekcja cech - ANOVA

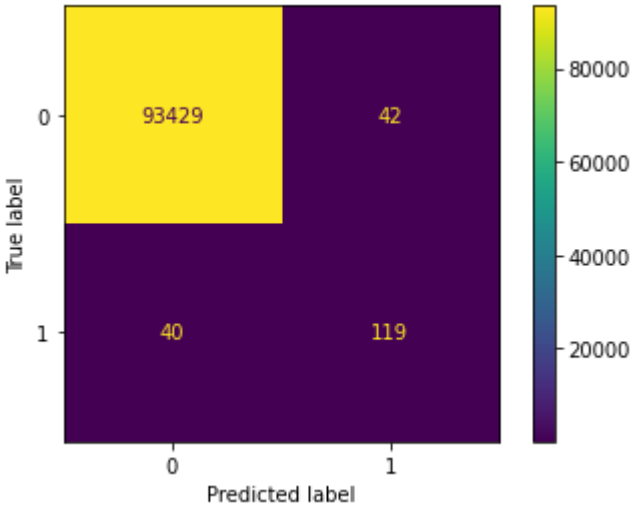
In [ ]:

```
from sklearn.feature_selection import SelectPercentile, f_classif
select = SelectPercentile(f_classif, percentile=10).fit(X, y)

X_anova = pd.DataFrame(select.transform(X))
X_anova.columns = select.get_feature_names_out()

X_train_anova, X_test_anova, y_train_anova, y_test_anova = train_test_split(
    X_anova, y, random_state=331, test_size=0.33)

print("train rows: {}, test rows: {}".format(
    X_train_anova.shape[0], X_test_anova.shape[0]))  # rows
```

```
train rows: 190096, test rows: 93630
```

In [ ]:

```
y_pred_anova, graph = computeDecitionTree(
    X_train_anova, X_test_anova, y_train_anova)

accuracy, sensitivity, specificity = analyse(y_test_anova, y_pred_anova, True
```




```
accuracy = 0.9991, sensitivity = 0.7484, specificity = 0.9996
```

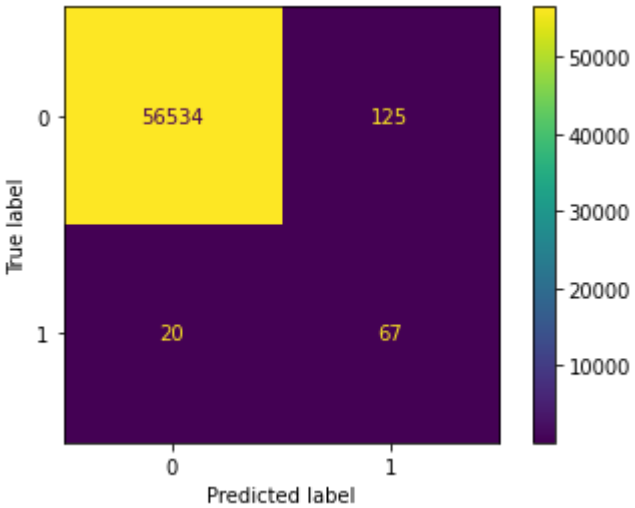
Wnioski

- Selekcja cech, w przypadku drzewa decyzyjnego nie wpływa znacząco na czułość klasyfikatora, ponieważ drzewo samo w sobie przeprowadza optymlizację zgodnie z kryterium gini albo entropii. Nawet może prowadzić do zmniejszenia czułości klasyfikatora.

4. Problem niezbalansowanych klas

Oversampling - SMOTE

In [ ]:
```
from imblearn.over_sampling import SMOTE

X_train_smote, y_train_smote = SMOTE(random_state=0).fit_resample(X_train, y_

y_pred_smote, graph = computeDecitionTree(
    X_train_smote, X_test, y_train_smote)
analyse(y_test, y_pred_smote, True)
```




```
accuracy = 0.9974, sensitivity = 0.7701, specificity = 0.9978
```
Out[ ]:
```
(0.9974447538152469, 0.7701149425287356, 0.9977938191637692)
```

Oversampling - SVMSMOTE

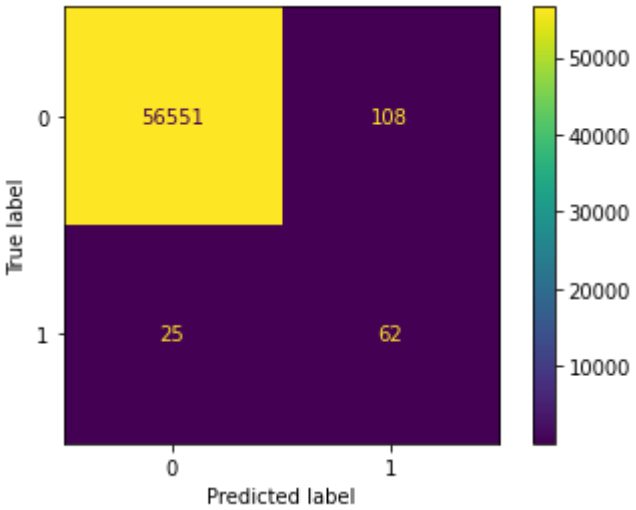
In [ ]:
```
""" from imblearn.over_sampling import  SVMSMOTE

X_train_smote, y_train_smote = SVMSMOTE().fit_resample(X_train, y_train)
```

```
y_pred_smote, graph = computeDecitionTree(
    X_train_smote, X_test, y_train_smote)
analyse(y_test, y_pred_smote, True) """
```

Out[ ]:
```
' from imblearn.over_sampling import  SVMSMOTE\n\nX_train_smote, y_train_smot
e = SVMSMOTE().fit_resample(X_train, y_train)\n\ny_pred_smote, graph = comput
eDecitionTree(\n    X_train_smote, X_test, y_train_smote)\nanalyse(y_test, y_
pred_smote, True) '
```

Oversampling - ADASYN

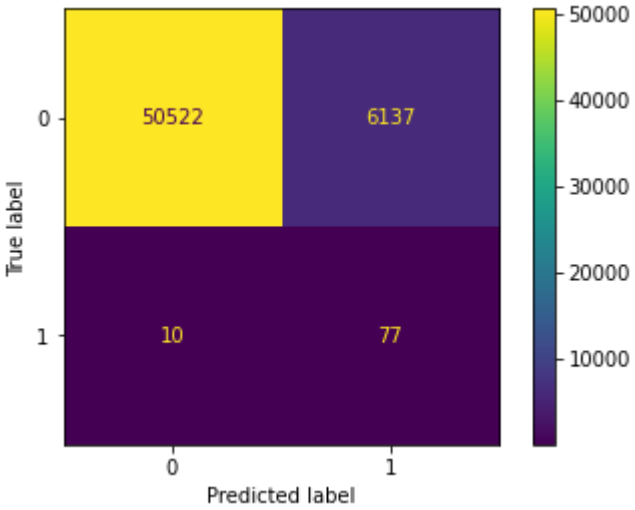
In [ ]:
```
from imblearn.over_sampling import ADASYN

X_train_adasyn, y_train_adasyn = ADASYN(random_state=0).fit_resample(X_train,

y_pred_adasyn, graph = computeDecitionTree(
    X_train_adasyn, X_test, y_train_adasyn)
analyse(y_test, y_pred_adasyn, True)
```



Out[ ]:
```
accuracy = 0.9977, sensitivity = 0.7126, specificity = 0.9981
(0.9976562224650196, 0.7126436781609196, 0.9980938597574966)
```

Undersampling - random

In [ ]:
```
from imblearn.under_sampling import RandomUnderSampler
cc = RandomUnderSampler(random_state=0)
X_train_under, y_train_under = cc.fit_resample(X_train, y_train)

y_pred_under, graph = computeDecitionTree(
    X_train_under, X_test, y_train_under)
analyse(y_test, y_pred_under, True)
```



Out[ ]:
```
accuracy = 0.8917, sensitivity = 0.8851, specificity = 0.8917
(0.8916751841539492, 0.8850574712643678, 0.8916853456644134)
```

Oversampling i undersampling - SMOTEENN

In [ ]:
```
""" from imblearn.combine import SMOTEENN
smote_enn = SMOTEENN(random_state=0)
X_train_smoteenn, y_train_smoteenn = smote_enn.fit_resample(X_train, y_train)

y_pred_smoteenn, graph = computeDecitionTree(
    X_train_smoteenn, X_test, y_train_smoteenn)
analyse(y_test, y_pred_smoteenn, True) """
```

Out[ ]:
```
' from imblearn.combine import SMOTEENN\nsmote_enn = SMOTEENN(random_state=0)
\nX_train_smoteenn, y_train_smoteenn = smote_enn.fit_resample(X_train, y_trai
n)\n\ny_pred_smoteenn, graph = computeDecitionTree(\n    X_train_smoteenn, X_
test, y_train_smoteenn)\nanalyse(y_test, y_pred_smoteenn, True) '
```

Oversampling i undersampling - SMOTETOMEK

In [ ]:
```
""" from imblearn.combine import SMOTETomek
smote_enn = SMOTETomek(random_state=0)
X_train_smotetomek, y_train_smotetomek = smote_enn.fit_resample(X_train, y_tr

y_pred_smotetomek, graph = computeDecitionTree(
    X_train_smotetomek, X_test, y_train_smotetomek)
analyse(y_test, y_pred_smotetomek, True) """
```

Out[ ]:
```
' from imblearn.combine import SMOTETomek\nsmote_enn = SMOTETomek(random_stat
e=0)\nX_train_smotetomek, y_train_smotetomek = smote_enn.fit_resample(X_trai
n, y_train)\n\ny_pred_smotetomek, graph = computeDecitionTree(\n    X_train_s
motetomek, X_test, y_train_smotetomek)\nanalyse(y_test, y_pred_smotetomek, Tr
ue) '
```

Wnioski

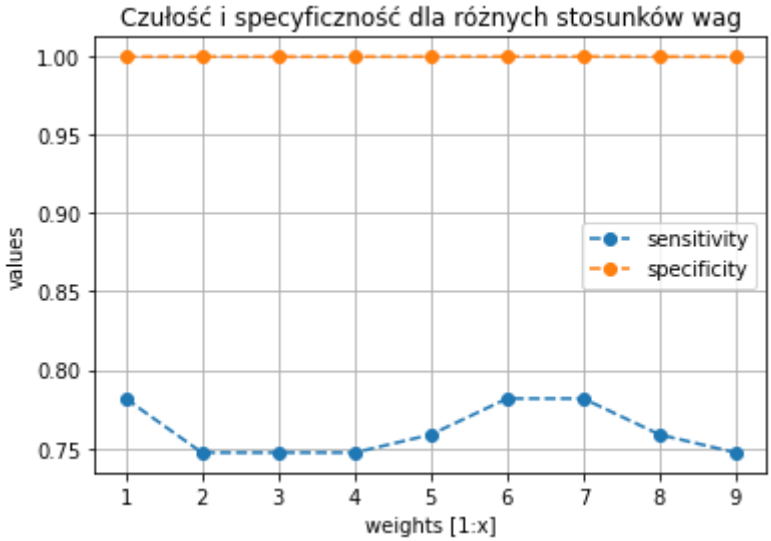
- Oversampling z użyciem algorytmu SMOTE - syntezy nowych próbek - obniża czułość do ok. 71% względem wariantu podstawowego
- Oversampling z użyciem algorytmu ADASYN - syntezy nowych próbek - obniża czułość do ok. 71% względem wariantu podstawowego
- Oversampling nie wpływa znacznie na specyficzność
- Undersampling z użyciem klastrowania jest zbyt złożony obliczeniowo, żeby prowadzić badania z jego użyciem na tym zbiorze danych

- Unsersampling z użyciem losowego odrzucania próbek znacząco poprawia czułość (88.5%) kosztem znaczącego spadku specyficzności (89%)
- Kombinacje oversamplingu i undersamplingu są zbyt złożone obliczeniowo, żeby prowadzić badania z ich użyciem na tym zbiorze danych
- Operacje resamplingu i ich skuteczność bardzo zależą od podziału danych na testowe i treningowe, szczególnie w przypadku danych niezbalansowanych

5. Niesymetryczne koszty błędów

In [ ]:
```
r = range(1, 10)
sensitivities = []
specificities = []
for weight in r:
    y_pred, graph = computeDecitionTree(
        X_train, X_test, y_train, class_weight={0: 1, 1: weight})
    accuracy, sensitivity, specificity = analyse(y_test, y_pred, False)
    sensitivities.append(sensitivity)
    specificities.append(specificity)
```

In [ ]:
```
fig, ax = plt.subplots()

ax.plot(r, sensitivities, label="sensitivity", **plot_style)
ax.plot(r, specificities, label="specificity", **plot_style)

ax.set(xlabel='weights [1:x]', ylabel='values',
       title='Czułość i specyficzność dla różnych stosunków wag')
ax.grid()
ax.legend()

plt.show()
print("max sensitivity: {}, max specificity: {}".format(
    max(sensitivities), max(specificities)))
```




```
max sensitivity: 0.7816091954022989, max specificity: 0.9996823099595827
```

Wnioski

- Manipulacja kosztem błędów pomaga nieznacznie podnieść czułość i nie zmienia specyficzności.
- Zbyt duży stosunek wag klas (np. 0.9999999:0.0000001) może prowadzić do drastycznego zmniejszenia specyficzności.

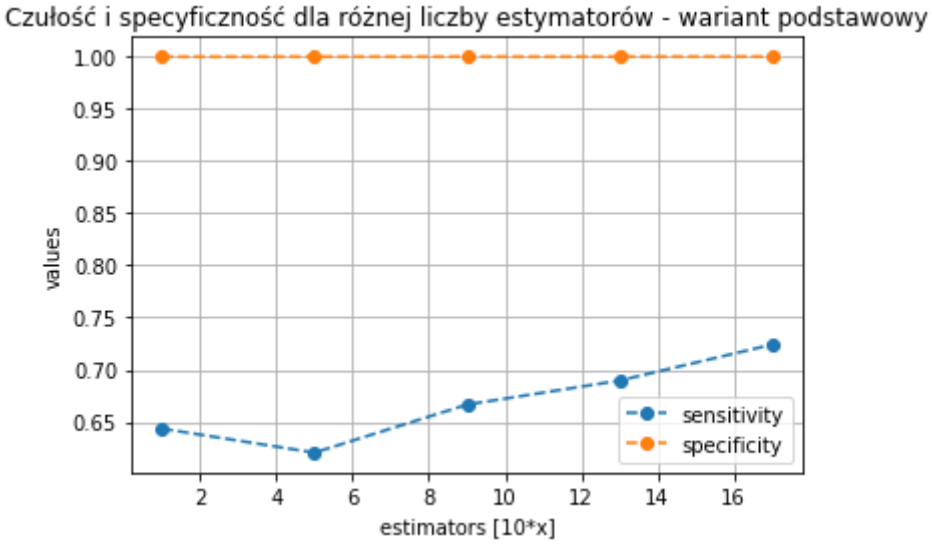
- W przypadku danych niezbalansowanych, bez wcześniejszej ich obróbki, duże znaczenie dla wyników ma losowy rozdział danych na testowe i treningowe.

6. Składanie klasyfikatorów

Wariant podstawowy

In [ ]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier


r = range(1, 20, 4)
sensitivities = []
specificities = []
for factor in r:
    clf = AdaBoostClassifier(DecisionTreeClassifier(
        max_depth=1), n_estimators=10 * factor, random_state=0)
    clf.fit(X_train, y_train)
    y_pred_adaboost = clf.predict(X_test)
    accuracy, sensitivity, specificity = analyse(y_test, y_pred_adaboost, Fal
    sensitivities.append(sensitivity)
    specificities.append(specificity)
```

In [ ]:
```
fig, ax = plt.subplots()

ax.plot(r, sensitivities, label="sensitivity", **plot_style)
ax.plot(r, specificities, label="specificity", **plot_style)

ax.set(xlabel='estimators [10*x]', ylabel='values',
       title='Czułość i specyficzność dla różnej liczby estymatorów - wariant
ax.grid()
ax.legend()

plt.show()
print("max sensitivity: {}, max specificity: {}".format(
    max(sensitivities), max(specificities)))
```




```
max sensitivity: 0.7241379310344828, max specificity: 0.9997882066397219
```

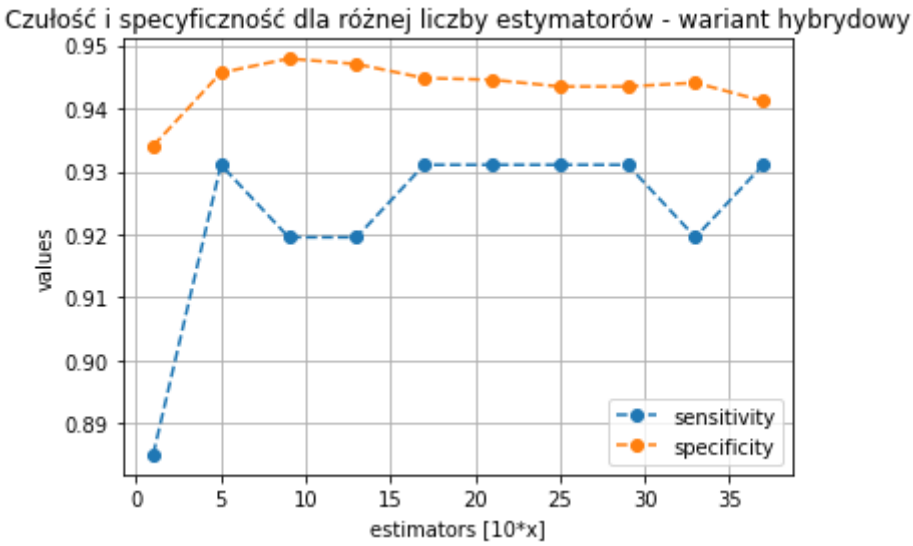
Warianty hybrydowe

In [ ]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier
```

```
r = range(1, 40, 4)
sensitivities = []
specificities = []
for factor in r:
    clf = AdaBoostClassifier(DecisionTreeClassifier(
        max_depth=1), n_estimators=10 * factor, random_state=0)
    clf.fit(X_train_under, y_train_under)
    y_pred_adaboost = clf.predict(X_test)
    accuracy, sensitivity, specificity = analyse(y_test, y_pred_adaboost, Fal
    sensitivities.append(sensitivity)
    specificities.append(specificity)
```

In [ ]:

```
fig, ax = plt.subplots()

ax.plot(r, sensitivities, label="sensitivity", **plot_style)
ax.plot(r, specificities, label="specificity", **plot_style)

ax.set(xlabel='estimators [10*x]', ylabel='values',
       title='Czułość i specyficzność dla różnej liczby estymatorów - wariant
ax.grid()
ax.legend()

plt.show()
print("max sensitivity: {}, max specificity: {}".format(
    max(sensitivities), max(specificities)))
```

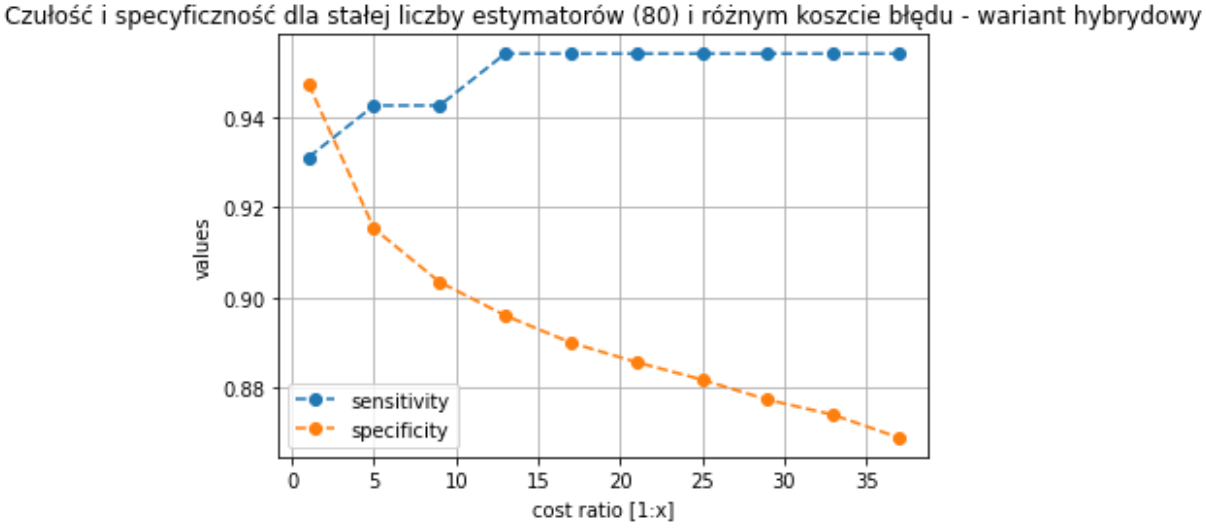



```
max sensitivity: 0.9310344827586207, max specificity: 0.947881183924884
```

In [ ]:

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier


r = range(1, 40, 4)
sensitivities = []
specificities = []
for factor in r:
    clf = AdaBoostClassifier(DecisionTreeClassifier(
        max_depth=1, class_weight={0:1,1:factor}), n_estimators=80, random_st
    clf.fit(X_train_under, y_train_under)
    y_pred_adaboost = clf.predict(X_test)
    accuracy, sensitivity, specificity = analyse(y_test, y_pred_adaboost, Fal
    sensitivities.append(sensitivity)
    specificities.append(specificity)
```

In [ ]:
```
fig, ax = plt.subplots()

ax.plot(r, sensitivities, label="sensitivity", **plot_style)
ax.plot(r, specificities, label="specificity", **plot_style)

ax.set(xlabel='cost ratio [1:x]', ylabel='values',
       title='Czułość i specyficzność dla stałej liczby estymatorów (80) i ró
ax.grid()
ax.legend()

plt.show()
print("max sensitivity: {}, max specificity: {}".format(
    max(sensitivities), max(specificities)))
```




```
max sensitivity: 0.9540229885057471, max specificity: 0.9473693499708784
```

Wnioski

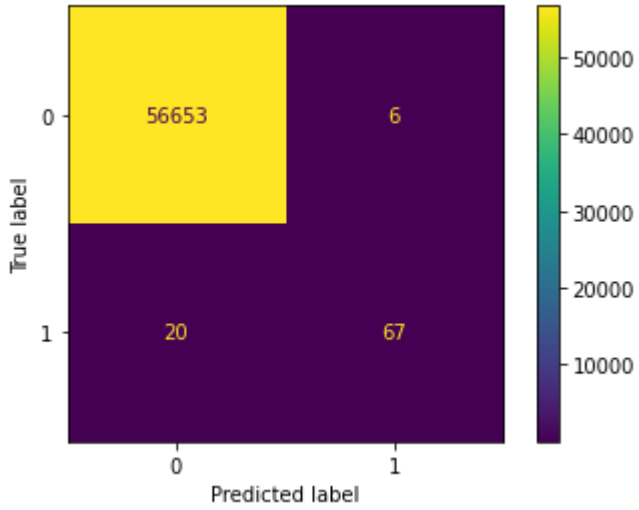
- Używając tylko składania klasyfikatorów możmy osiągnąć znaczną poprawę czułości przy zachowaniu wysokiej specyficzności
- Czułość wzrasta wraz ze wzrostem liczby klasyfikatorów, a od pewnego momentu stabilizuje się
- Warianty hybrydowe, stosujące pozostałe techniki, wraz ze składaniem klasyfikatorów pozwalają osiągnąć lepsze rezultaty:
  - wraz z undersamplingiem czułość na poziomie 94% i specyficzność 93% dla 330 klasyfikatorów,
  - wraz z undersamplingiem, stałą liczbą klasyfikatorów (80) dla różnych kosztów błędu czułość rośnie skokowo z ciągłym spadkiem specyficzności;
- AdaBoost znacząco poprawia wyniki klasyfikatora - drzewa decyzyjnego, jednak do pełnego wykorzystania potencjału konieczna jest modyfikacja zbioru uczącego oraz parametrów klasyfikatora

Multi-layer Perceptron

1. Wariant podstawowy

In [ ]:
```
from helpers import computeMLP, analyse
y_pred = computeMLP(X_train, X_test, y_train)
```

In [ ]:
```
accuracy, sensitivity, specificity = analyse(y_test,y_pred,True)
```




```
accuracy = 0.9995, sensitivity = 0.7701, specificity = 0.9999
```

Wnioski

- wysoka dokładność nie oznacza poprawnego działania w przypadku niezbalansowanych danych
- stosunkowo niska czułość (77%) wskazuje na duży odsetek fałszywie negatywnych klasyfikacji

2. Parametry klasyfikatora

In [ ]:
```
import matplotlib.pyplot as plt
import numpy as np
plot_style = {"marker": 'o', "linestyle": '--'}
```

Liczba warstw ukrytych

In [ ]:
```
r = range(1, 10)
num_layers_sensitivities = []
num_layers_specificities = []
for value in r:
    y_pred = computeMLP(
        X_train, X_test, y_train, hidden_layer_sizes=(10,)*value)
    accuracy, sensitivity, specificity = analyse(y_test, y_pred)
    num_layers_sensitivities.append(sensitivity)
    num_layers_specificities.append(specificity)
```

In [ ]:
```
fig, ax = plt.subplots()

ax.plot(r, num_layers_sensitivities, label="sensitivity", **plot_style)
ax.plot(r, num_layers_specificities, label="specificity", **plot_style)

ax.set(xlabel='liczba warstw ukrytych', ylabel='values',
       title='Czułość i specyficzność dla różnej liczby warstw ukrytych (po 1
ax.grid()
ax.legend()

plt.show()
```

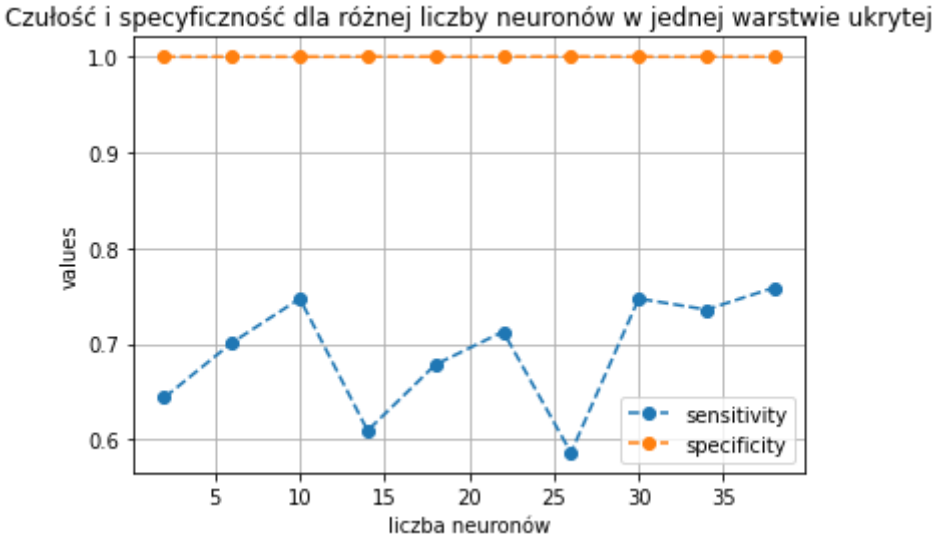
```
print("max sensitivity: {}, max specificity: {}".format(
    max(num_layers_sensitivities), max(num_layers_specificities)))
```


```
max sensitivity: 0.8275862068965517, max specificity: 0.9998058560864117
```

Liczba neuronów w warstwie ukrytej

In [ ]:
```
r = range(2, 40+1, 4)
num_neurons_sensitivities = []
num_neurons_specificities = []
for value in r:
    y_pred = computeMLP(
        X_train, X_test, y_train, hidden_layer_sizes=(value,))
    accuracy, sensitivity, specificity = analyse(y_test, y_pred)
    num_neurons_sensitivities.append(sensitivity)
    num_neurons_specificities.append(specificity)
```

In [ ]:
```
fig, ax = plt.subplots()

ax.plot(r, num_neurons_sensitivities, label="sensitivity", **plot_style)
ax.plot(r, num_neurons_specificities, label="specificity", **plot_style)

ax.set(xlabel='liczba neuronów', ylabel='values',
       title='Czułość i specyficzność dla różnej liczby neuronów w jednej war
ax.grid()
ax.legend()

plt.show()
print("max sensitivity: {}, max specificity: {}".format(
    max(num_neurons_sensitivities), max(num_neurons_specificities)))
```




```
max sensitivity: 0.7586206896551724, max specificity: 0.9999470516599305
```

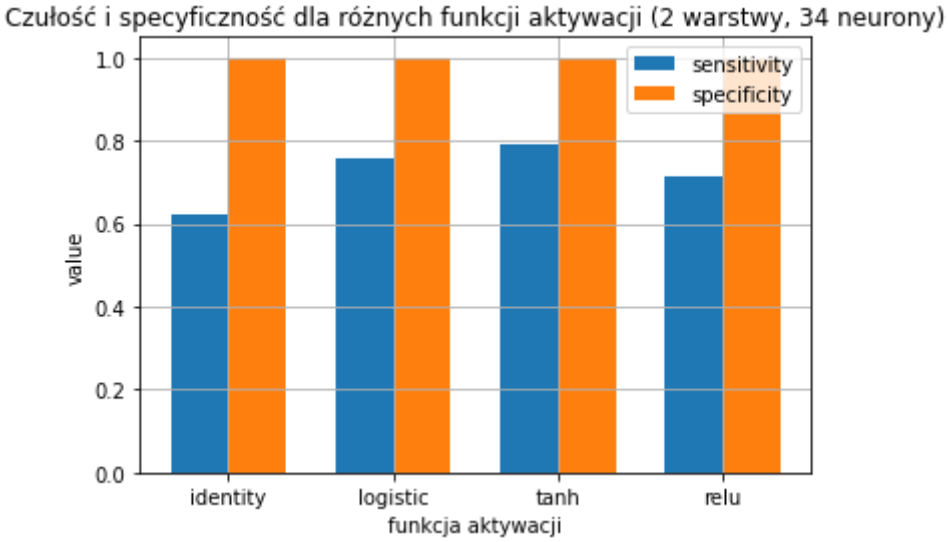
Funkcja aktywacji

In [ ]:

```
from helpers import computeMLP, analyse
activations = ['identity', 'logistic', 'tanh', 'relu']
sensitivities = []
specificities = []
for a in activations:
    y_pred = computeMLP(X_train, X_test, y_train, hidden_layer_sizes=(34,34),
    accuracy, sensitivity, specificity = analyse(y_test,y_pred, False)
    sensitivities.append(sensitivity)
    specificities.append(specificity)
```

In [ ]:

```
fig, ax = plt.subplots()

x = np.arange(len(activations))
width = 0.35

ax.bar(x - width/2, sensitivities, label="sensitivity",width = width)
ax.bar(x + width/2 ,specificities, label="specificity",width = width)

ax.set(xlabel='funkcja aktywacji', ylabel='value', xticks=x, xticklabels=acti
       title='Czułość i specyficzność dla różnych funkcji aktywacji (2 warstw
ax.grid()
ax.legend()

plt.show()
print("max sensitivity: {}, max specificity: {}".format(
    max(sensitivities), max(specificities)))
```




```
max sensitivity: 0.7931034482758621, max specificity: 0.9999647011066203
```

Wnioski

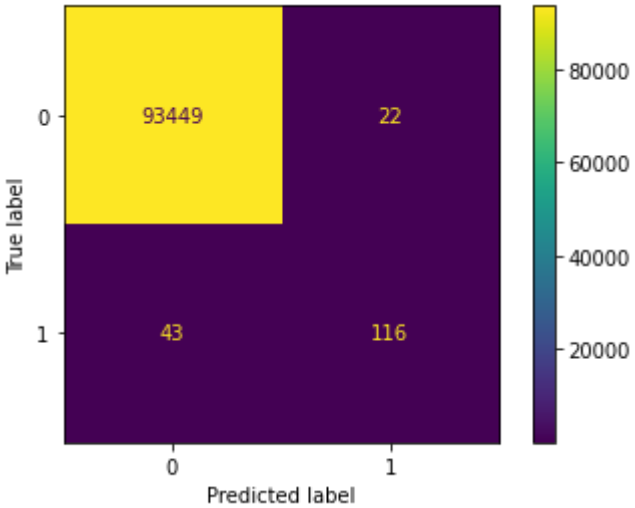
- zdecydowanie lepsze rezyltaty daje zastosowanie większej już jedna liczby warstw ukrytych,
- wraz ze wzrostem liczby warstw ukrytych czułość rośnie w ogólnym trendzie,
- wraz ze wzorstem liczby neuronów w warstwie ukrytej czułość wzrasta, jednak do pewnego stopnia,
- dla danych standaryzowanych wszystkie analizowane funkcje aktywacji dają podobne rezultaty z przewagą tangensa hiperbolicznego, a wyjątek stanowi identyczność, która wykonuje mapowanie f(x) = x

3. Selekcja cech

ANOVA

In [ ]:

```
y_pred_anova = computeMLP(
    X_train_anova, X_test_anova, y_train_anova, hidden_layer_sizes=(34,34))

accuracy, sensitivity, specificity = analyse(y_test_anova, y_pred_anova, True
```

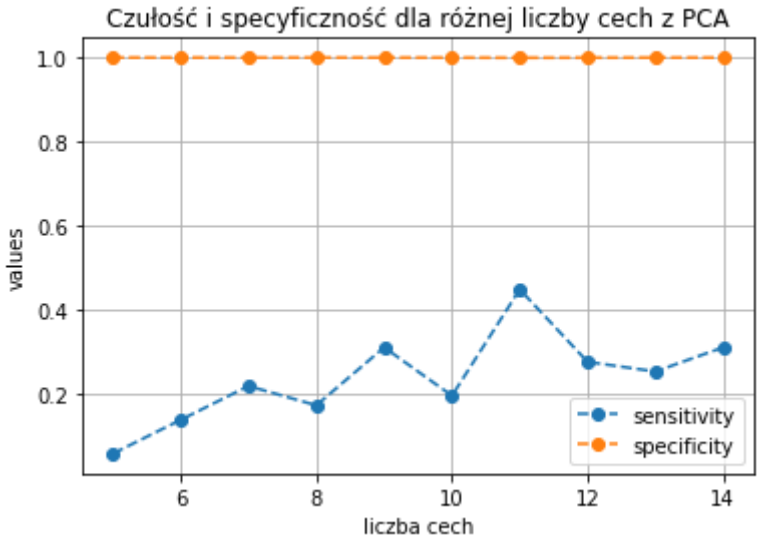



```
accuracy = 0.9993, sensitivity = 0.7296, specificity = 0.9998
```

PCA

In [ ]:
```
from sklearn.decomposition import PCA
r = range(5,15)
components_sensitivities = []
components_specificities = []
for value in r:
    pca = PCA(n_components=value, random_state=0)
    pca.fit(X)

    X_pca = pd.DataFrame(pca.transform(X))

    X_train_pca, X_test_pca, y_train_pca, y_test_pca = train_test_split(
        X_pca, y, random_state=0, test_size=0.2)

    y_pred_pca = computeMLP(
        X_train_pca, X_test_pca, y_train_pca, hidden_layer_sizes=(34,34))

    accuracy, sensitivity, specificity = analyse(y_test_pca, y_pred_pca, Fals
    components_sensitivities.append(sensitivity)
    components_specificities.append(specificity)
```

In [ ]:
```
fig, ax = plt.subplots()

ax.plot(r, components_sensitivities, label="sensitivity", **plot_style)
ax.plot(r, components_specificities, label="specificity", **plot_style)

ax.set(xlabel='liczba cech', ylabel='values',
       title='Czułość i specyficzność dla różnej liczby cech z PCA')
ax.grid()
ax.legend()

plt.show()
print("max sensitivity: {}, max specificity: {}".format(
    max(components_sensitivities), max(components_specificities)))
```




```
max sensitivity: 0.4482758620689655, max specificity: 0.9999294022132407
```

Wnioski

- selekcja cech ANOVA, biorąc do analizy te, które mają największe znaczenie dla rozróżnienia klas, w wypadku MLP nie zmienia czułości, ale skraca drastycznie czas trenowania klasyfikatora
- selekcja cech nie zmniejsza specyficzności

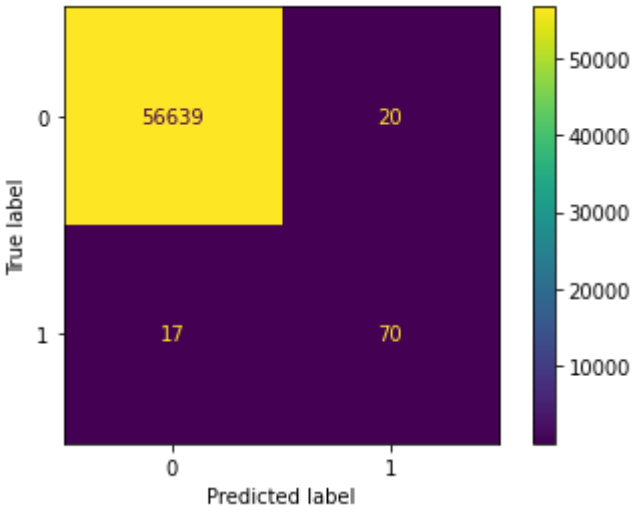
- metoda PCA kompresująca informacje czyni klasyfikator bezużytecznym, ponieważ wyniki wariantu domyślnego bazują już na zbiorze danych zanonimizowanych metodą PCA. Cechami, które mogły zmienić rzutowanie danych są cechy `amount` i `time`, które nie zostały poddane anonimizacji, a w obecnej analizie poddane są transformacji. W analizie macierzy korelacji widać, że korelacja tych cech z innymi, w przeciwieństwie do pozostałych wartości, czasem nie jest bliska zeru.

4. Problem niezbalansowanych klas

Analiza bazuje na danych uzyskanych przy analizie drzewa decyzyjnego.

Oversampling - SMOTE

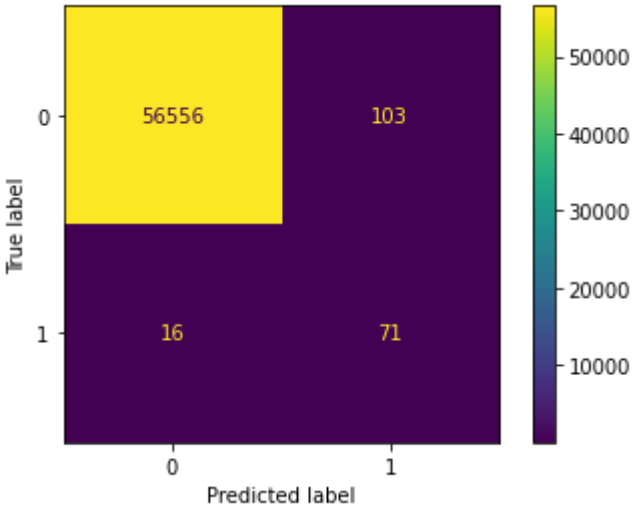
In [ ]:
```
y_pred_smote = computeMLP(
    X_train_smote, X_test, y_train_smote, hidden_layer_sizes=(34,34))
analyse(y_test, y_pred_smote, True)
```



Out[ ]:
```
accuracy = 0.9993, sensitivity = 0.8046, specificity = 0.9996
(0.9993479716632009, 0.8045977011494253, 0.9996470110662031)
```

Oversampling - ADASYN

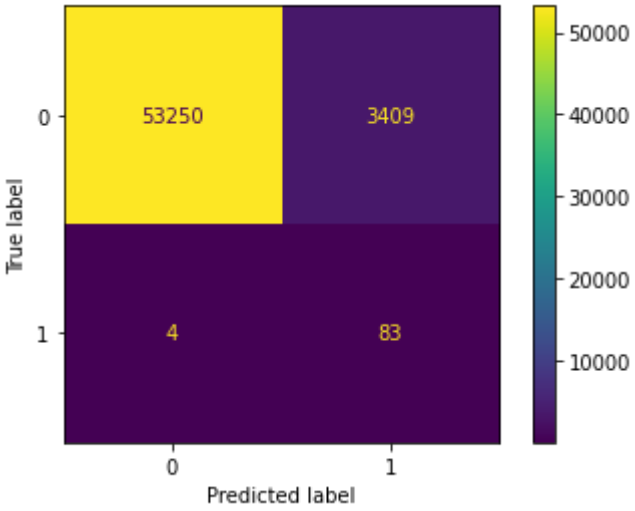
In [ ]:
```
y_pred_adasyn = computeMLP(
    X_train_adasyn, X_test, y_train_adasyn, hidden_layer_sizes=(34,34))
analyse(y_test, y_pred_adasyn, True)
```



```
accuracy = 0.9979, sensitivity = 0.8161, specificity = 0.9982
(0.9979029358897543, 0.8160919540229885, 0.9981821069909458)
```
Out[ ]:

Undersampling - random

In [ ]:
```
y_pred_under = computeMLP(
    X_train_under, X_test, y_train_under, hidden_layer_sizes=(34,34))
analyse(y_test, y_pred_under, True)
```




```
accuracy = 0.9399, sensitivity = 0.9540, specificity = 0.9398
(0.9398547915271561, 0.9540229885057471, 0.939833036234314)
```
Out[ ]:

Wnioski

- Oversampling z użyciem algorytmu SMOTE - syntezy nowych próbek - poprawie czułość do ok. 81% względem wariantu standardowego
- Oversampling z użyciem algorytmu ADASYN - syntezy nowych próbek - poprawie czułość do ok. 81% względem wariantu standardowego
- Oversampling nie wpływa znacznie na specyficzność
- Undersampling znacząco zwiększa czułość koszten specyficzności. Wyniki jednak są lepsze niż w przypadku drzewa decyzyjnego

5. Niesymetryczne koszty błędów

Klasyfikator MLP z biblioteki scikit-learn nie ma zaimplementowanej funkcjonalności wagi klas. Źródło: https://github.com/scikit-learn/scikit-learn/issues/9113

Kosztem błędów, w przypadku klasyfikatora MLP można sterować imlicite poprzez re-sampling zbioru danych treningowych, czego analiza została wykonana w części `4. Problem niezbalanzowanych klas` .

6. Składanie klasyfikatorów

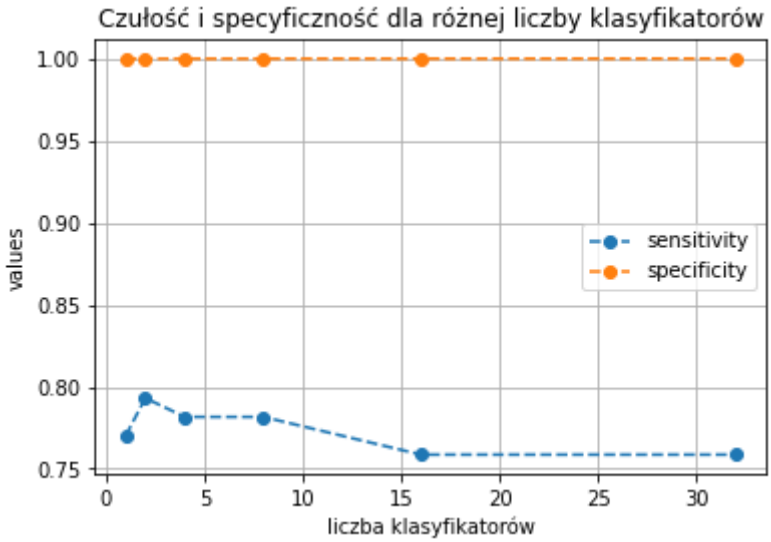
Wariant podstawowy

In [ ]:
```
from sklearn.ensemble import BaggingClassifier
from sklearn.neural_network import MLPClassifier
from helpers import analyse

r = [1,2,4,8,16,32]
```

```
sensitivities = []
specificities = []
for value in r:
    clf = BaggingClassifier(base_estimator=MLPClassifier(hidden_layer_sizes=(
                            n_estimators=value, random_state=0, n_jobs=8,
                            max_samples=0.8, max_features=0.8).fit(X_train, y
    y_pred = clf.predict(X_test);

    accuracy, sensitivity, specificity = analyse(y_test, y_pred, False)
    sensitivities.append(sensitivity)
    specificities.append(specificity)
```

In [ ]:

```
fig, ax = plt.subplots()

ax.plot(r, sensitivities, label="sensitivity", **plot_style)
ax.plot(r, specificities, label="specificity", **plot_style)

ax.set(xlabel='liczba klasyfikatorów', ylabel='values',
       title='Czułość i specyficzność dla różnej liczby klasyfikatorów')
ax.grid()
ax.legend()

plt.show()
print("max sensitivity: {}, max specificity: {}".format(
    max(sensitivities), max(specificities)))
```




```
max sensitivity: 0.7931034482758621, max specificity: 0.9999117527665508
```

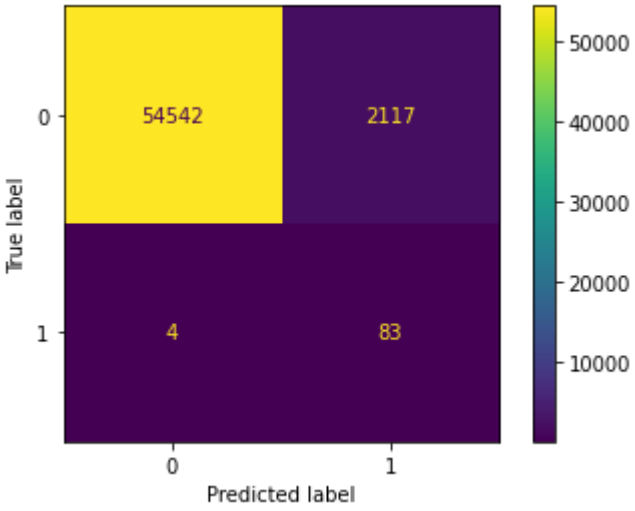
Wariant hybrydowy

Parametry klasyfikatora + undersampling + Bagging ensamble

In [ ]:

```
from sklearn.ensemble import BaggingClassifier
from sklearn.neural_network import MLPClassifier
from helpers import analyse
from sklearn.decomposition import PCA
from imblearn.under_sampling import RandomUnderSampler


cc = RandomUnderSampler(random_state=0)
X_train_under, y_train_under = cc.fit_resample(X_train, y_train)

clf = BaggingClassifier(base_estimator=MLPClassifier(hidden_layer_sizes=(34,3
                        n_estimators=100, max_samples=0.8, max_features=0.8,
y_pred_under = clf.predict(X_test);
```

```
accuracy, sensitivity, specificity = analyse(y_test, y_pred_under, True)
```




```
accuracy = 0.9626, sensitivity = 0.9540, specificity = 0.9626
```

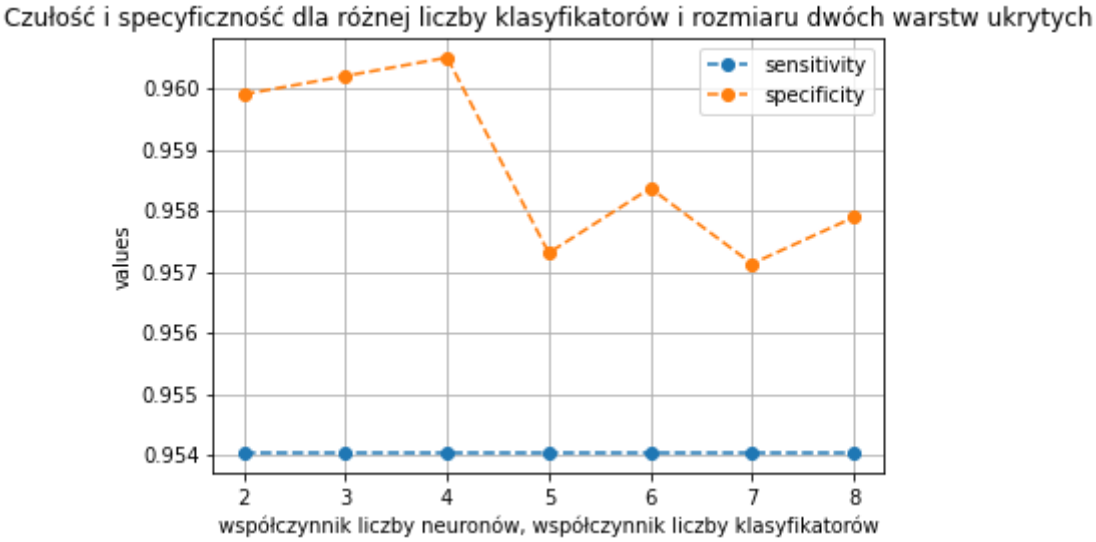
In [ ]:

```
from sklearn.ensemble import BaggingClassifier
from sklearn.neural_network import MLPClassifier
from helpers import analyse

r = range(2,9)
sensitivities = []
specificities = []
for value in r:
    cc = RandomUnderSampler(random_state=0)
    X_train_under, y_train_under = cc.fit_resample(X_train, y_train)

    clf = BaggingClassifier(base_estimator=MLPClassifier(hidden_layer_sizes=(
                            n_estimators=value*10, max_samples=0.9, max_featu
    y_pred_under = clf.predict(X_test);

    accuracy, sensitivity, specificity = analyse(y_test, y_pred_under, False)
    sensitivities.append(sensitivity)
    specificities.append(specificity)
```

```
/home/damian_koper/.virtualenvs/aiedLab-2Ceq4bPp/lib/python3.8/site-packages/
sklearn/neural_network/_multilayer_perceptron.py:692: ConvergenceWarning: Sto
chastic Optimizer: Maximum iterations (500) reached and the optimization has
n't converged yet.
  warnings.warn(
```

In [ ]:

```
fig, ax = plt.subplots()

ax.plot(r, sensitivities, label="sensitivity", **plot_style)
ax.plot(r, specificities, label="specificity", **plot_style)

ax.set(xlabel='współczynnik liczby neuronów, współczynnik liczby klasyfikator
       title='Czułość i specyficzność dla różnej liczby klasyfikatorów i rozm
ax.grid()
ax.legend()

plt.show()
print("max sensitivity: {}, max specificity: {}".format(
    max(sensitivities), max(specificities)))
```




```
max sensitivity: 0.9540229885057471, max specificity: 0.960500538308124
```

Wnioski

- Implementacja biblioteki scikit-learn nie pozwala użyć algorytmu AdaBoost w przypadku MLP
- Używając tylko składania klasyfikatorów możmy osiągnąć znaczną poprawę czułości przy zachowaniu wysokiej specyficzności
- MLP nie jest tak podatne na wzrost czułości w składaniu klasyfikatorów jak drzewo decyzyjne
- Warianty hybrydowe, stosujące pozostałe techniki, wraz ze składaniem klasyfikatorów pozwalają osiągnąć lepsze rezultaty
- Składanie klasyfikatorów z undersamplingiem okazało się dawać taką samą czułość ale lepszą specyficzność niż sam undersampling

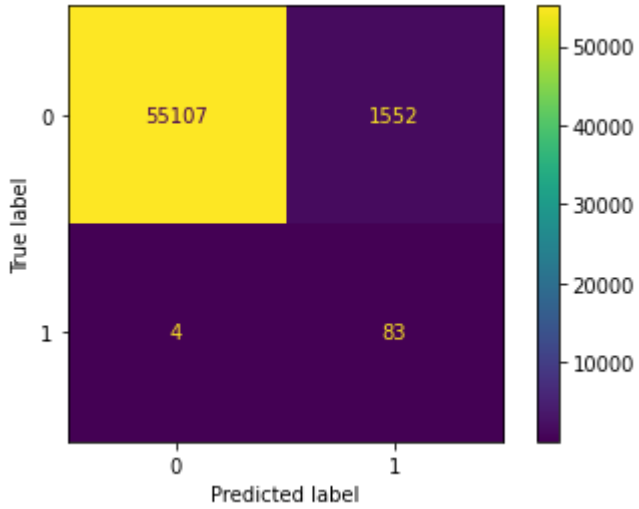
Bonus - regresja logistyczna

Niepełna analiza jako chęć replikacji wyninków innych osób na tym samym zbiorze danych.
https://www.kaggle.com/mariapushkareva/credit-card-fraud-detection-f1-score-0-86/notebook

LogReg + SMOTE

In [ ]:

```
from imblearn.over_sampling import SMOTE
from sklearn.linear_model import LogisticRegression
from helpers import analyse

X_train_smote, y_train_smote = SMOTE(random_state=0).fit_resample(X_train, y_
logreg = LogisticRegression()
logreg.fit(X_train_smote, y_train_smote)
y_pred_smote = logreg.predict(X_test)
a,se,sp = analyse(y_test, y_pred_smote, True)
```

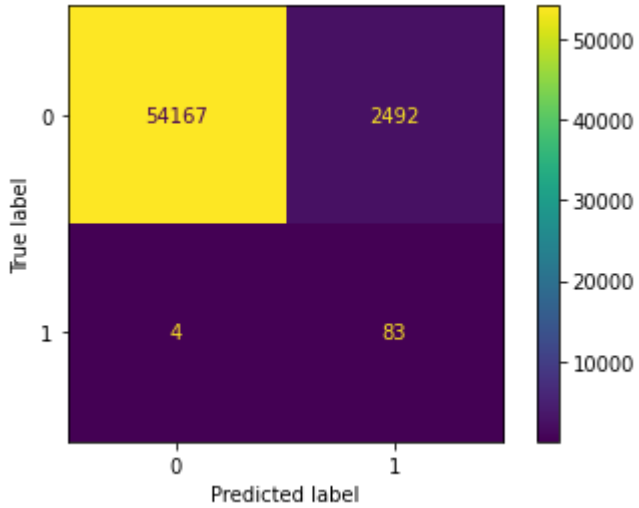



```
accuracy = 0.9726, sensitivity = 0.9540, specificity = 0.9726
```

LogReg + Randum Undersampling

In [ ]:

```
from sklearn.linear_model import LogisticRegression
from imblearn.under_sampling import RandomUnderSampler

cc = RandomUnderSampler(random_state=0)
X_train_under, y_train_under = cc.fit_resample(X_train, y_train)
logreg = LogisticRegression()
logreg.fit(X_train_under, y_train_under)
y_pred_under = logreg.predict(X_test)
a,se,sp = analyse(y_test, y_pred_under, True)
```




```
accuracy = 0.9560, sensitivity = 0.9540, specificity = 0.9560
```

Wnioski

- Regresja logistyczna w tym wypadku daje lepsze wyniki niż ten sam wariant dla drzewa decyzyjnego i MLP
- Regresja logistyczna jest mniej złożona obliczeniowo od pozostałych klasyfikatorów