wb_km2

April 7, 2022

1 WB - milestone 2 - inżynieria cech i wstępne modelowanie

1.1 physioNet dataset

1.1.1 Autorzy:

Paulina Jaszczuk
Jędrzej Sokołowski
Filip Szympliński

[ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import copy

from sklearn import preprocessing
from sklearn.model_selection import train_test_split

import xgboost as xgb
from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor,␣
↪BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

from tqdm.notebook import tqdm

from sklearn.metrics import mean_squared_error as MSE
from sklearn.metrics import roc_auc_score, accuracy_score, f1_score, roc_curve,␣
↪r2_score

import warnings
warnings.filterwarnings('ignore')


# ustawia domyślną wielkość wykresów
plt.rcParams['figure.figsize'] = (12,8)
# to samo tylko dla tekstu
plt.rcParams['font.size'] = 16
```

1.2 Import danych i poglądowe informacje

Zbiór danych medycznych physioNet opisujący pacjentów z oddziałów kariochirurgicznych.

[ ]:
```
data = pd.read_csv("patients_data_ready.csv", sep=",", index_col=[0])
```

[ ]:
```
data.rename(columns = {"MechVent_max" : "MechVent"}, inplace = True)
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 3855 entries, 0 to 3999
Data columns (total 82 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   Age              3855 non-null   float64
 1   Gender           3855 non-null   float64
 2   Height           3855 non-null   float64
 3   ICUType          3855 non-null   float64
 4   Weight           3855 non-null   float64
 5   MAP_mean         3855 non-null   float64
 6   MAP_max          3855 non-null   float64
 7   MAP_min          3855 non-null   float64
 8   HCT_mean         3855 non-null   float64
 9   HCT_max          3855 non-null   float64
 10  HCT_min          3855 non-null   float64
 11  SysABP_mean      3855 non-null   float64
 12  SysABP_max       3855 non-null   float64
 13  SysABP_min       3855 non-null   float64
 14  NIDiasABP_mean   3855 non-null   float64
 15  NIDiasABP_max    3855 non-null   float64
 16  NIDiasABP_min    3855 non-null   float64
 17  Lactate_mean     3855 non-null   float64
 18  Lactate_max      3855 non-null   float64
 19  Lactate_min      3855 non-null   float64
 20  HR_mean          3855 non-null   float64
 21  HR_max           3855 non-null   float64
 22  HR_min           3855 non-null   float64
 23  FiO2_mean        3855 non-null   float64
 24  FiO2_max         3855 non-null   float64
 25  FiO2_min         3855 non-null   float64
 26  Urine_mean       3855 non-null   float64
 27  Urine_max        3855 non-null   float64
 28  Urine_min        3855 non-null   float64
 29  BUN_mean         3855 non-null   float64
 30  BUN_max          3855 non-null   float64
 31  BUN_min          3855 non-null   float64
 32  Mg_mean          3855 non-null   float64
 33  Mg_max           3855 non-null   float64
```

```
34  Mg_min           3855 non-null   float64
35  Na_mean          3855 non-null   float64
36  Na_max           3855 non-null   float64
37  Na_min           3855 non-null   float64
38  MechVent         3855 non-null   float64
39  K_mean           3855 non-null   float64
40  K_max            3855 non-null   float64
41  K_min            3855 non-null   float64
42  PaCO2_mean       3855 non-null   float64
43  PaCO2_max        3855 non-null   float64
44  PaCO2_min        3855 non-null   float64
45  pH_mean          3855 non-null   float64
46  pH_max           3855 non-null   float64
47  pH_min           3855 non-null   float64
48  GCS_mean         3855 non-null   float64
49  GCS_max          3855 non-null   float64
50  GCS_min          3855 non-null   float64
51  Platelets_mean   3855 non-null   float64
52  Platelets_max    3855 non-null   float64
53  Platelets_min    3855 non-null   float64
54  Temp_mean        3855 non-null   float64
55  Temp_max         3855 non-null   float64
56  Temp_min         3855 non-null   float64
57  NISysABP_mean    3855 non-null   float64
58  NISysABP_max     3855 non-null   float64
59  NISysABP_min     3855 non-null   float64
60  PaO2_mean        3855 non-null   float64
61  PaO2_max         3855 non-null   float64
62  PaO2_min         3855 non-null   float64
63  Glucose_mean     3855 non-null   float64
64  Glucose_max      3855 non-null   float64
65  Glucose_min      3855 non-null   float64
66  Creatinine_mean  3855 non-null   float64
67  Creatinine_max   3855 non-null   float64
68  Creatinine_min   3855 non-null   float64
69  DiasABP_mean     3855 non-null   float64
70  DiasABP_max      3855 non-null   float64
71  DiasABP_min      3855 non-null   float64
72  WBC_mean         3855 non-null   float64
73  WBC_max          3855 non-null   float64
74  WBC_min          3855 non-null   float64
75  HCO3_mean        3855 non-null   float64
76  HCO3_max         3855 non-null   float64
77  HCO3_min         3855 non-null   float64
78  NIMAP_mean       3855 non-null   float64
79  NIMAP_max        3855 non-null   float64
80  NIMAP_min        3855 non-null   float64
81  Survived         3855 non-null   int64
```

```
dtypes: float64(81), int64(1)
memory usage: 2.4 MB
```

Dane zawierają 5 cech statycznych (`Age`, `Gender`, `Height`, `Weight`, `ICUType` - rodzaj oddziału, na którym przebywał pacjent) oraz 75 cech dynamicznych, mierzonych co najmniej jednokrotnie. Wśród nich jest jedna zmienna binarna - `MechVent` - kolumna odpowiadająca informacji, czy pacjent został poddany wentylacji z użyciem respiratora. Zmienne dynamiczne, oprócz `MechVent` wyrażone są przez minimum, średnią i maximum z wszystkich pomiarów dla danego pacjenta. Zmienna celu - `Survived` - jest binarna (1, jeśli pacjent przeżył i 0, jeśli nie przeżył).

1.3 Inżynieria danych

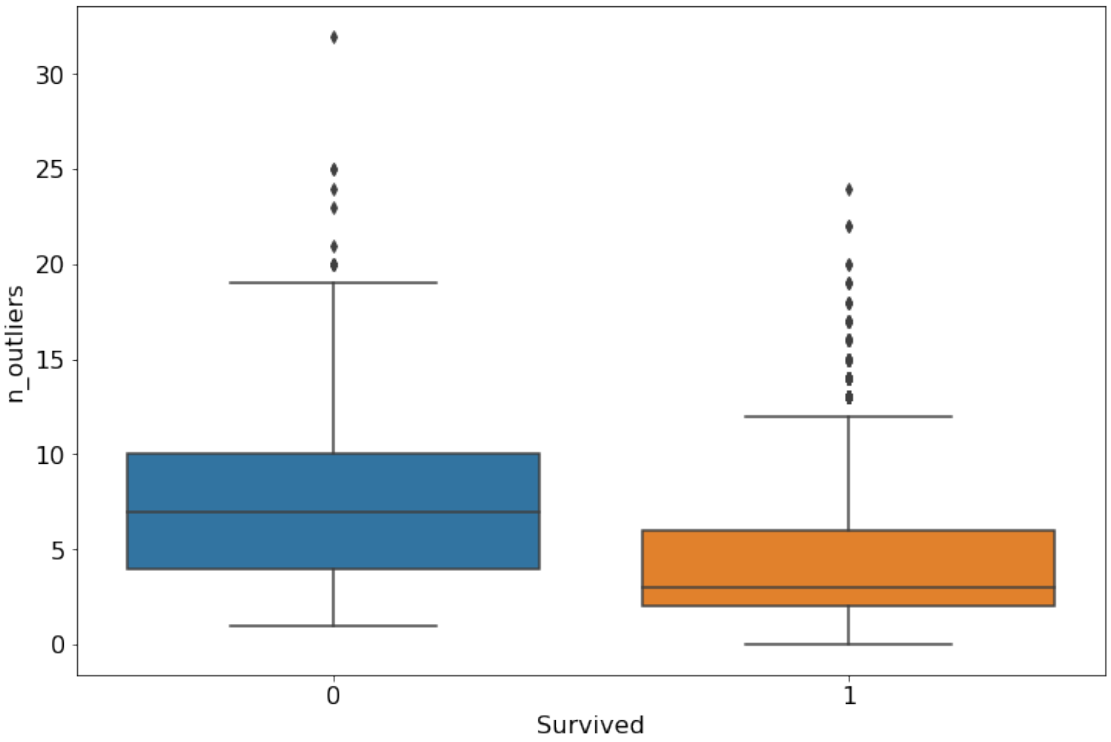
Wszystkie braki danych zostały przez nas usunięte lub zaimputowane na poprzednim etapie pracy. Dane nie zawierają wartości kategorycznych, które należałoby zakodować.

1.3.1 Outliery

Na poprzednim etapie prac zauważyliśmy, że wśród danych występują liczne outliery, jednak z racji na medyczny charakter danych, mogą one stanowić ważną informację w procesie modelowania i samej predykcji.

[ ]:
```
#sprawdzamy liczbę outlierów dla każdego pacjenta
data_outliers = data.drop(["Gender", "ICUType", "MechVent"], axis=1)

Q1 = data_outliers.quantile(0.25)
Q3 = data_outliers.quantile(0.75)
IQR = Q3 - Q1

n_outliers = []

for i in range(data.shape[0]):
  n_outliers.append(((data_outliers.iloc[i] < (Q1 - 1.5 * IQR)) |␣
 ↪(data_outliers.iloc[i] > (Q3 + 1.5 * IQR))).sum())

data["n_outliers"] = n_outliers
```

[ ]:
```
#boxplot liczby outlierów z podziałem na zmienną celu
sns.boxplot( x=data['Survived'], y=data['n_outliers'])
plt.show()
```

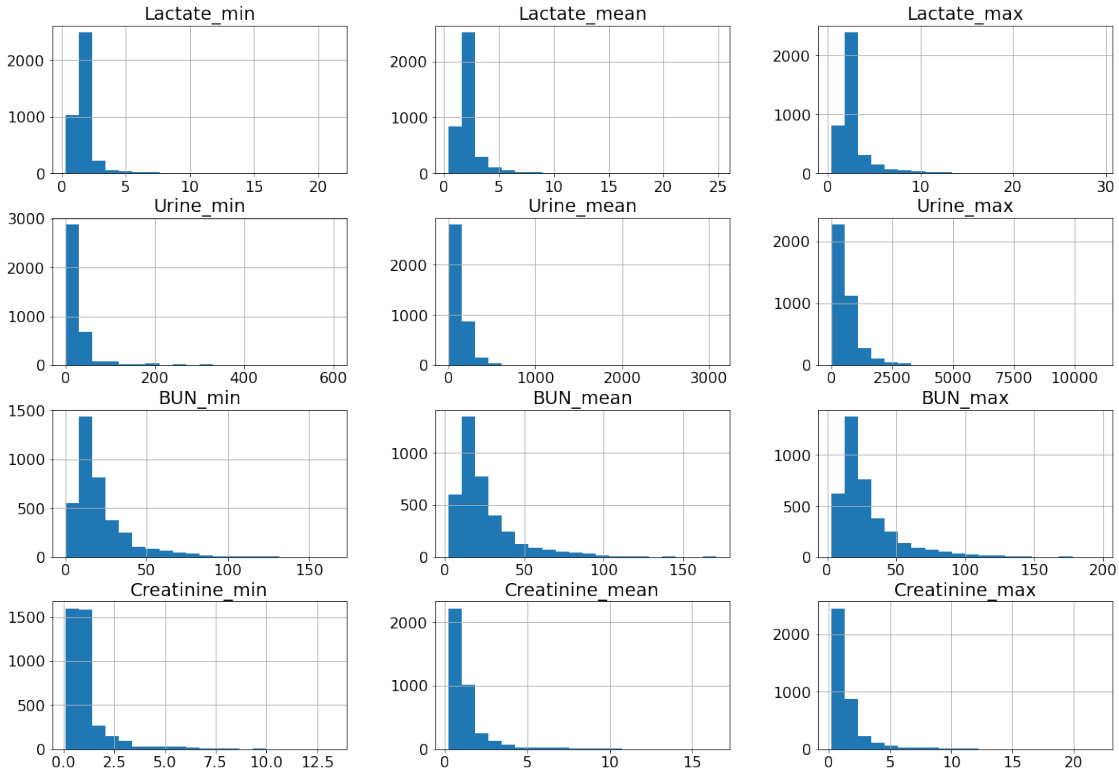


Jak widać, pacjenci, którzy nie przeżyli zdecydowanie częściej mieli skrajnie wysokie lub skrajnie niskie wartości odczytów medycznych - wśród ich danych jest wyraźnie więcej outlierów. Potwierdza to nasze przypuszczenie, że nie możemy pozbyć się tych informacji z naszego zbioru danych.

1.3.2 Przekształcenia danych

Na poprzednim etapie prac zidentyfikowaliśmy kilka zmiennych, których rozkłady mocno skupiają się w pobliżu zera.

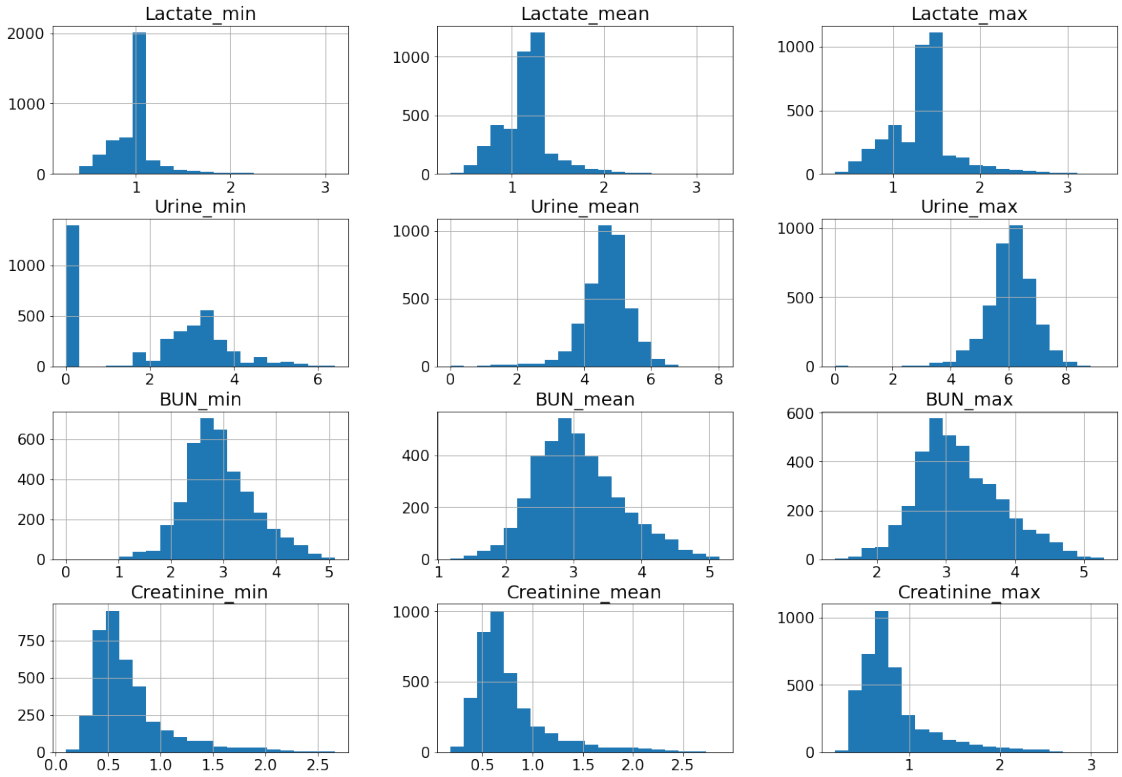
[ ]:
```
cols_to_log = ['Lactate_min', 'Lactate_mean', 'Lactate_max',
               'Urine_min', 'Urine_mean', 'Urine_max',
               'BUN_min', 'BUN_mean', 'BUN_max',
               'Creatinine_min', 'Creatinine_mean', 'Creatinine_max']
data[cols_to_log].hist(bins = 20, figsize=(20, 14))
plt.show()
```



Postanowiliśmy stworzyć drugą ramkę danych, w której dane kolumny zlogarytmujemy. Dalsze prace - modelowanie - będziemy wykonywać na obu ramkach danych i porównamy wyniki.

[ ]:
```
#tworzymy drugą ramkę ze zlogarytmowanymi wyżej wymienionymi kolumnami
data_log = copy.deepcopy(data)
data_log[cols_to_log] = data_log[cols_to_log].apply(lambda x: np.log1p(x),␣
↪axis=1)
```

[ ]:
```
data_log[cols_to_log].hist(bins = 20, figsize=(20, 14))
plt.show()
```



Rozkłady zmiennych zlogarytmowanych prezentują się znacznie bardziej informatywnie.

1.4 Wstępne modelowanie

Nasze dane są mocno niezbalansowane - pacjentów, którzy zmarli jest zdecydawanie mniej niż tych, którzy przeżyli (stosunek mniej więcej 1:6). Jednocześnie to właśnie właściwa predykcja przypadków śmierci interesuje nas najbardziej. Między innymi z tego właśnie powodu zdecydowaliśmy się zastosować kilka metryk, a część z nich wyliczaliśmy oddzielnie dla obu klas - metryka ważona, z racji niezbalansowania klas, mogłaby być mocno zawyżona.

Do ocenienia działań modelu zastosowaliśmy metryki F1 score - osobno dla każdej klasy a także ważony, a także precision i recall oddzielnie dla każdej klasy. W przypadku metryk liczonych osobno dla obu klas, pierwsza wartość dotyczy pacjentów, którzy zmarli, zaś druga tych, którzy przeżyli.

[ ]:
```
def show_model_metrics(model, X, y):
    y_pred = model.predict(X)
    print(f"F1 score: {f1_score(y, y_pred, average=None)}")
    print(f"F1 score micro: {f1_score(y, y_pred, average='micro')}")
    print(f"F1 score weighted: {f1_score(y, y_pred, average='weighted')}")
    print(f"Precision score: {precision_score(y, y_pred, average=None)}")
    print(f"Recall score: {recall_score(y, y_pred, average=None)}")
```

[ ]:
```
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score, accuracy_score, f1_score, roc_curve,␣
↪confusion_matrix, precision_score, recall_score

y = data["Survived"]
X = data.drop("Survived", axis= 1)

#podział na train/val/test zbioru oryginalnego
X_train_val, X_test, y_train_val, y_test = train_test_split(X, y,␣
↪random_state=420, test_size=0.2)
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val,␣
↪random_state=420, test_size=0.125)
```

[ ]:
```
y_log = data_log["Survived"]
X_log = data_log.drop("Survived", axis= 1)

#podział na train/val/test zbioru z kolumnami zlogarytmowanymi
X_train_val_log, X_test_log, y_train_val_log, y_test_log =␣
↪train_test_split(X_log, y_log, random_state=420, test_size=0.2)
X_train_log, X_val_log, y_train_log, y_val_log =␣
↪train_test_split(X_train_val_log, y_train_val_log, random_state=420,␣
↪test_size=0.125)
```

1.4.1 XGBoost - dane po tranformacjach logarytmicznych

[ ]:
```
xgb_clf = xgb.XGBClassifier(random_state=1,
                    booster='gbtree',
                    use_label_encoder=False,
                    )
xgb_clf.fit(X_train_log, y_train_log)
```

[ ]:
```
XGBClassifier(random_state=1, use_label_encoder=False)
```

[ ]:
```
y_pred = xgb_clf.predict(X_val_log)
print(accuracy_score(y_val_log, y_pred, normalize=True))
show_model_metrics(xgb_clf, X_val_log, y_val_log)
```

```
0.8626943005181347
F1 score: [0.2739726 0.9241774]
F1 score micro: 0.8626943005181347
F1 score weighted: 0.8281626987886429
Precision score: [0.625      0.87297297]
Recall score: [0.1754386  0.98176292]
```

[ ]:
```
xgb_clf = xgb.XGBClassifier(random_state=42,
                    booster='gbtree',
                    use_label_encoder=False,
```

```
                    learning_rate=0.01, n_estimators=20
                    )
xgb_clf.fit(X_train_log, y_train_log)
```

[ ]:
```
XGBClassifier(learning_rate=0.01, n_estimators=20, random_state=42,
              use_label_encoder=False)
```

[ ]:
```
y_pred = xgb_clf.predict(X_val_log)
print(accuracy_score(y_val_log, y_pred, normalize=True))
show_model_metrics(xgb_clf, X_val_log, y_val_log)
```

```
0.8575129533678757
F1 score: [0.20289855 0.92176387]
F1 score micro: 0.8575129533678757
F1 score weighted: 0.8156101822171706
Precision score: [0.58333333 0.86631016]
Recall score: [0.12280702 0.98480243]
```

[ ]:
```
xgb_clf = xgb.XGBClassifier(random_state=42,
                    booster='gbtree',
                    use_label_encoder=False,
                    learning_rate=0.5, n_estimators=50
                    )
xgb_clf.fit(X_train, y_train)
```

[ ]:
```
XGBClassifier(learning_rate=0.5, n_estimators=50, random_state=42,
              use_label_encoder=False)
```

[ ]:
```
y_pred = xgb_clf.predict(X_val_log)
print(accuracy_score(y_val_log, y_pred, normalize=True))
show_model_metrics(xgb_clf, X_val_log, y_val_log)
```

```
0.8575129533678757
F1 score: [0.17910448 0.92198582]
F1 score micro: 0.8575129533678757
F1 score weighted: 0.8122857216508134
Precision score: [0.6       0.8643617]
Recall score: [0.10526316 0.98784195]
```

1.4.2 XGBoost - dane bez tranformacjacji logarytmicznych

[ ]:
```
xgb_clf = xgb.XGBClassifier(random_state=1,
                    booster='gbtree',
                    use_label_encoder=False,
                    )
xgb_clf.fit(X_train, y_train)
```

[ ]:
```
XGBClassifier(random_state=1, use_label_encoder=False)
```

[ ]:
```
y_pred = xgb_clf.predict(X_val)
print(accuracy_score(y_val, y_pred, normalize=True))
show_model_metrics(xgb_clf, X_val, y_val)
```

```
0.8626943005181347
F1 score: [0.2739726 0.9241774]
F1 score micro: 0.8626943005181347
F1 score weighted: 0.8281626987886429
Precision score: [0.625      0.87297297]
Recall score: [0.1754386  0.98176292]
```

[ ]:
```
xgb_clf = xgb.XGBClassifier(random_state=42,
                    booster='gbtree',
                    use_label_encoder=False,
                    learning_rate=0.01, n_estimators=20
                    )
xgb_clf.fit(X_train, y_train)
```

[ ]:
```
XGBClassifier(learning_rate=0.01, n_estimators=20, random_state=42,
              use_label_encoder=False)
```

[ ]:
```
y_pred = xgb_clf.predict(X_val)
print(accuracy_score(y_val, y_pred, normalize=True))
show_model_metrics(xgb_clf, X_val, y_val)
```

```
0.8575129533678757
F1 score: [0.20289855 0.92176387]
F1 score micro: 0.8575129533678757
F1 score weighted: 0.8156101822171706
Precision score: [0.58333333 0.86631016]
Recall score: [0.12280702 0.98480243]
```

[ ]:
```
xgb_clf = xgb.XGBClassifier(random_state=42,
                    booster='gbtree',
                    use_label_encoder=False,
                    learning_rate=0.5, n_estimators=50
                    )
xgb_clf.fit(X_train, y_train)
```

[ ]:
```
XGBClassifier(learning_rate=0.5, n_estimators=50, random_state=42,
              use_label_encoder=False)
```

[ ]:
```
y_pred = xgb_clf.predict(X_val)
print(accuracy_score(y_val, y_pred, normalize=True))
show_model_metrics(xgb_clf, X_val, y_val)
```

```
0.8678756476683938
F1 score: [0.38554217 0.92597968]
F1 score micro: 0.8678756476683938
F1 score weighted: 0.846174141356631
Precision score: [0.61538462 0.88611111]
Recall score: [0.28070175 0.96960486]
```

1.4.3 Random Forest - dane po tranformacjach logarytmicznych

[ ]:
```
rForest = RandomForestClassifier()
rForest.fit(X_train_log, y_train_log)

y_pred = rForest.predict(X_val_log)
print(accuracy_score(y_val_log, y_pred, normalize=True))
show_model_metrics(rForest, X_val_log, y_val_log)
```

```
0.8575129533678757
F1 score: [0.12698413 0.92242595]
F1 score micro: 0.8575129533678757
F1 score weighted: 0.8049643353910475
Precision score: [0.66666667 0.86052632]
Recall score: [0.07017544 0.99392097]
```

[ ]:
```
rForest = RandomForestClassifier(criterion='entropy')
rForest.fit(X_train_log, y_train_log)

y_pred = rForest.predict(X_val_log)
print(accuracy_score(y_val_log, y_pred, normalize=True))
show_model_metrics(rForest, X_val_log, y_val_log)
```

```
0.8601036269430051
F1 score: [0.12903226 0.92394366]
F1 score micro: 0.8601036269430051
F1 score weighted: 0.8065603717575384
Precision score: [0.8        0.86089239]
Recall score: [0.07017544 0.99696049]
```

[ ]:
```
rForest = RandomForestClassifier(n_estimators=300)
rForest.fit(X_train_log, y_train_log)

y_pred = rForest.predict(X_val_log)
print(accuracy_score(y_val_log, y_pred, normalize=True))
show_model_metrics(rForest, X_val_log, y_val_log)
```

```
0.8575129533678757
F1 score: [0.12698413 0.92242595]
F1 score micro: 0.8575129533678757
F1 score weighted: 0.8049643353910475
```

```
Precision score: [0.66666667 0.86052632]
Recall score: [0.07017544 0.99392097]
```

[ ]:
```
rForest = RandomForestClassifier(n_estimators=30)
rForest.fit(X_train_log, y_train_log)

y_pred = rForest.predict(X_val_log)
print(accuracy_score(y_val_log, y_pred, normalize=True))
show_model_metrics(rForest, X_val_log, y_val_log)
```

```
0.8626943005181347
F1 score: [0.23188406 0.92460882]
F1 score micro: 0.8626943005181347
F1 score weighted: 0.8223152665001824
Precision score: [0.66666667 0.86898396]
Recall score: [0.14035088 0.98784195]
```

1.4.4 Random Forest - dane bez tranformacji logarytmicznych

[ ]:
```
rForest = RandomForestClassifier()
rForest.fit(X_train, y_train)

y_pred = rForest.predict(X_val)
print(accuracy_score(y_val, y_pred, normalize=True))
show_model_metrics(rForest, X_val, y_val)
```

```
0.8575129533678757
F1 score: [0.17910448 0.92198582]
F1 score micro: 0.8575129533678757
F1 score weighted: 0.8122857216508134
Precision score: [0.6       0.8643617]
Recall score: [0.10526316 0.98784195]
```

[ ]:
```
rForest = RandomForestClassifier(criterion='entropy')
rForest.fit(X_train, y_train)

y_pred = rForest.predict(X_val)
print(accuracy_score(y_val, y_pred, normalize=True))
show_model_metrics(rForest, X_val, y_val)
```

```
0.8549222797927462
F1 score: [0.09677419 0.92112676]
F1 score micro: 0.854922279792746
F1 score weighted: 0.7993959410818916
Precision score: [0.6        0.85826772]
Recall score: [0.05263158 0.99392097]
```

[ ]:
```
rForest = RandomForestClassifier(n_estimators=300)
rForest.fit(X_train, y_train)

y_pred = rForest.predict(X_val)
print(accuracy_score(y_val, y_pred, normalize=True))
show_model_metrics(rForest, X_val, y_val)
```

```
0.8626943005181347
F1 score: [0.18461538 0.92503536]
F1 score micro: 0.8626943005181347
F1 score weighted: 0.8156987320892313
Precision score: [0.75       0.86507937]
Recall score: [0.10526316 0.99392097]
```

[ ]:
```
rForest = RandomForestClassifier(n_estimators=30)
rForest.fit(X_train, y_train)

y_pred = rForest.predict(X_val)
print(accuracy_score(y_val, y_pred, normalize=True))
show_model_metrics(rForest, X_val, y_val)
```

```
0.8626943005181347
F1 score: [0.23188406 0.92460882]
F1 score micro: 0.8626943005181347
F1 score weighted: 0.8223152665001824
Precision score: [0.66666667 0.86898396]
Recall score: [0.14035088 0.98784195]
```

Widzimy, że skuteczność modeli `XGBoost` i `Random Forest` dla różnych parametrów oscylowała niezmiennie wokół 85%, natomiast z racji niezbalansowania jest to wynik mocno zawyżony.

1.4.5 DecisionTreeClassifier

Wstępne parametry i wpływ parametrów na predykcyjność

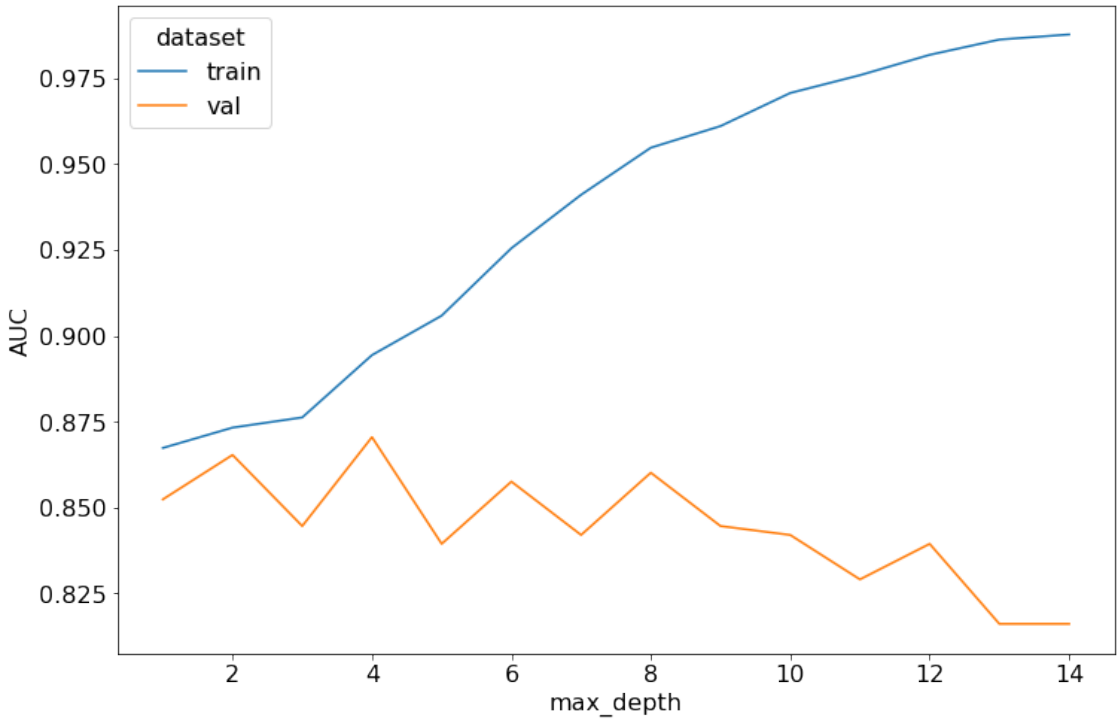
Bez log

[ ]:
```
dt_clf = DecisionTreeClassifier(criterion='gini',
                                max_depth=4)
dt_clf.fit(X_train, y_train)

print("------------------------------------------------------")
print("Train set scores")
show_model_metrics(dt_clf, X_train, y_train)

print("------------------------------------------------------")
print("Validation set scores")
show_model_metrics(dt_clf, X_val, y_val)
```

```
--------------------------------------------------
Train set scores
F1 score: [0.47897623 0.94122499]
F1 score micro: 0.8943661971830986
F1 score weighted: 0.8798887990025819
Precision score: [0.69312169 0.90952571]
Recall score: [0.36592179 0.97521368]
--------------------------------------------------
Validation set scores
F1 score: [0.375      0.92774566]
F1 score micro: 0.8704663212435233
F1 score weighted: 0.8461226002575699
Precision score: [0.65217391 0.88429752]
Recall score: [0.26315789 0.97568389]
```

[ ]:
```
cols = ["max_depth","AUC", "dataset"]
history = pd.DataFrame(columns=cols)

n_depth = np.arange(1,15,1)
for depth in tqdm(n_depth):
    dt = DecisionTreeClassifier(criterion='gini', max_depth=depth).fit(X_train,␣
 ↪y_train)
    train_score = dt.score(X_train,y_train)
    val_score = dt.score(X_val,y_val)
    history = history.append(dict(zip(cols, [depth, train_score, "train"])),␣
 ↪ignore_index=True)
    history = history.append(dict(zip(cols, [depth, val_score, "val"])),␣
 ↪ignore_index=True)

sns.lineplot(data=history, x = "max_depth", y = "AUC", hue = "dataset")
plt.show()
```

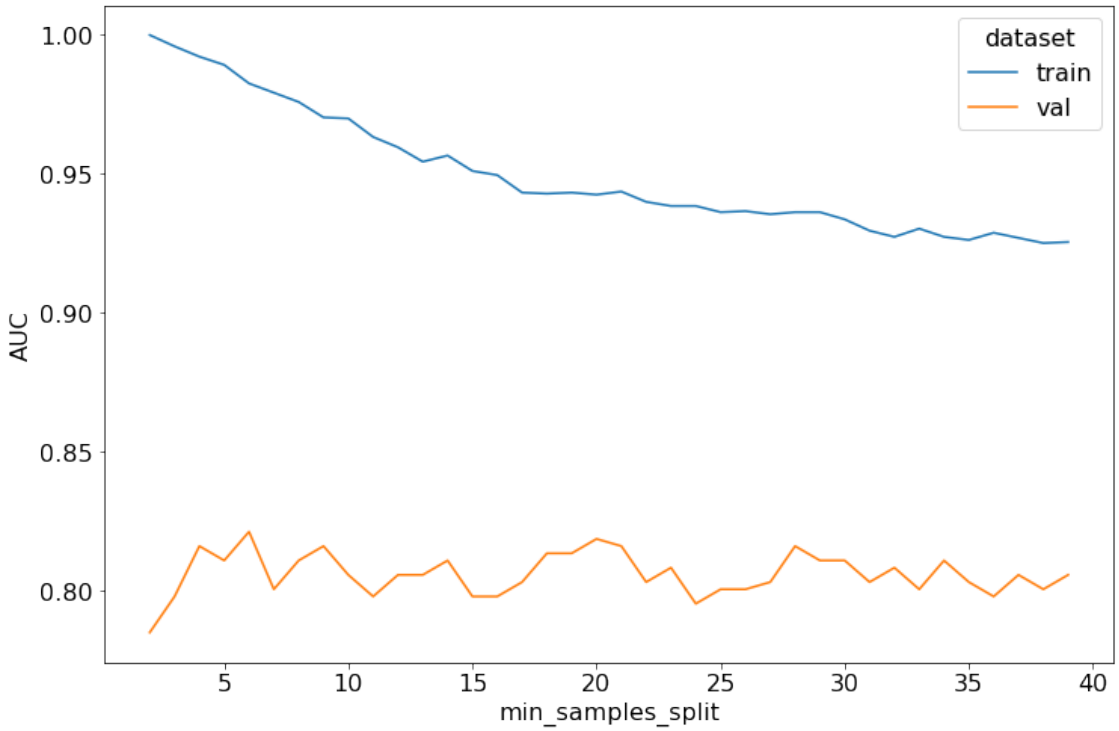
```
  0%|          | 0/14 [00:00<?, ?it/s]
```



Wraz ze wzrostem głębokości drzewa, wzrasta AUC na zbiorze treningowym, ale spada na zbiorze walidacyjnym - model się przeucza.

[ ]:
```
cols = ["min_samples_split","AUC", "dataset"]
history = pd.DataFrame(columns=cols)

n_depth = np.arange(2,40,1)
for depth in tqdm(n_depth):
    dt = DecisionTreeClassifier(criterion='gini', min_samples_split=depth).
 ↪fit(X_train, y_train)
    train_score = dt.score(X_train, y_train)
    val_score = dt.score(X_val, y_val)
    history = history.append(dict(zip(cols, [depth, train_score, "train"])),␣
 ↪ignore_index=True)
    history = history.append(dict(zip(cols, [depth, val_score, "val"])),␣
 ↪ignore_index=True)

sns.lineplot(data=history, x = "min_samples_split", y = "AUC", hue = "dataset")
plt.show()
```

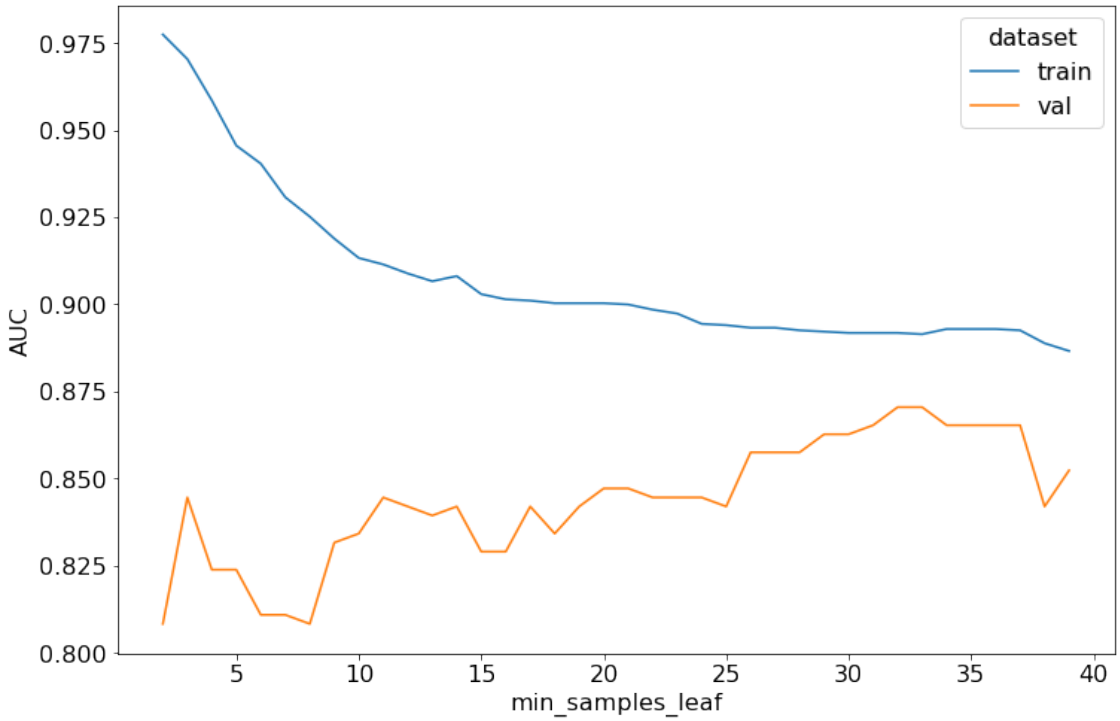
```
  0%|          | 0/38 [00:00<?, ?it/s]
```



AUC zbioru walidacyjnego pozostaje na podobnym poziomie.

[ ]:
```
cols = ["min_samples_leaf","AUC", "dataset"]
history = pd.DataFrame(columns=cols)

n_depth = np.arange(2,40,1)
for depth in tqdm(n_depth):
    dt = DecisionTreeClassifier(criterion='gini', min_samples_leaf=depth).
 ↪fit(X_train, y_train)
    train_score = dt.score(X_train, y_train)
    val_score = dt.score(X_val, y_val)
    history = history.append(dict(zip(cols, [depth, train_score, "train"])),␣
 ↪ignore_index=True)
    history = history.append(dict(zip(cols, [depth, val_score, "val"])),␣
 ↪ignore_index=True)

sns.lineplot(data=history, x = "min_samples_leaf", y = "AUC", hue = "dataset")
plt.show()
```

```
  0%|          | 0/38 [00:00<?, ?it/s]
```



AUC spada na zbiorze treningowym, ale rośne na walidacyjnym.

Z log

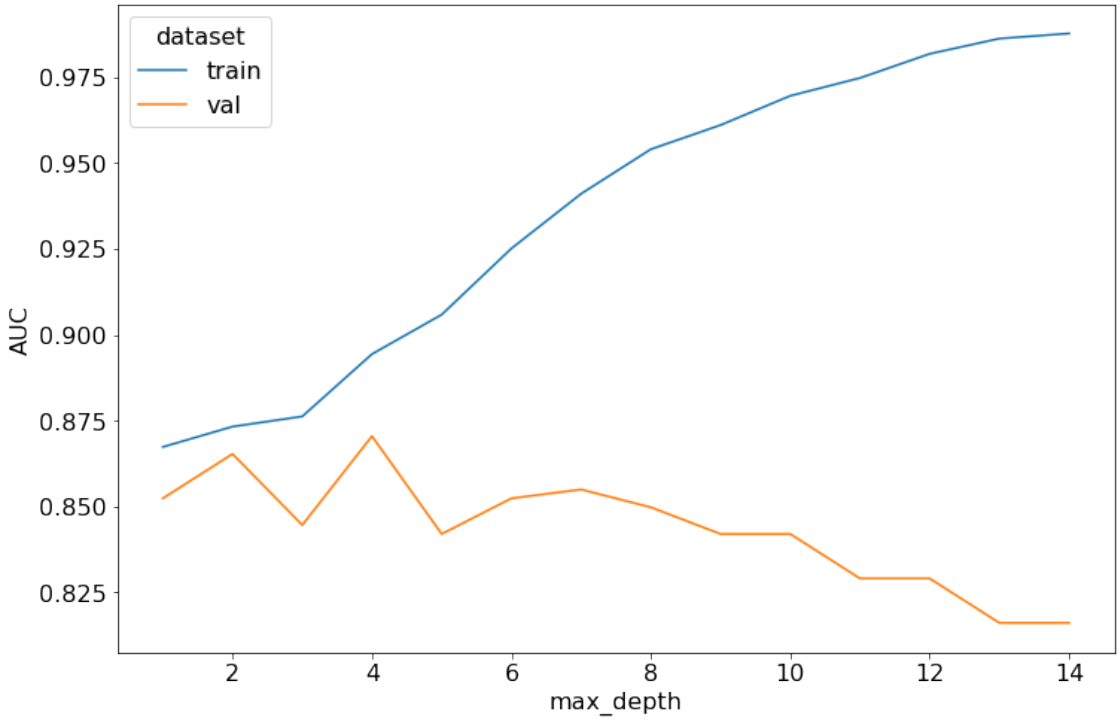
[ ]:
```
dt_clf_log = DecisionTreeClassifier(criterion='gini',
                                    max_depth=4)
dt_clf_log.fit(X_train_log, y_train_log)

print("-----------------------------------------------------")
print("Train set scores")
show_model_metrics(dt_clf_log, X_train_log, y_train_log)

print("-----------------------------------------------------")
print("Validation set scores")
show_model_metrics(dt_clf_log, X_val_log, y_val_log)
```

```
-----------------------------------------------
Train set scores
F1 score: [0.47897623 0.94122499]
F1 score micro: 0.8943661971830986
F1 score weighted: 0.8798887990025819
Precision score: [0.69312169 0.90952571]
Recall score: [0.36592179 0.97521368]
-----------------------------------------------
Validation set scores
```

```
F1 score: [0.36585366 0.92463768]
F1 score micro: 0.8652849740932642
F1 score weighted: 0.8421229420674474
Precision score: [0.6        0.88365651]
Recall score: [0.26315789 0.96960486]
```

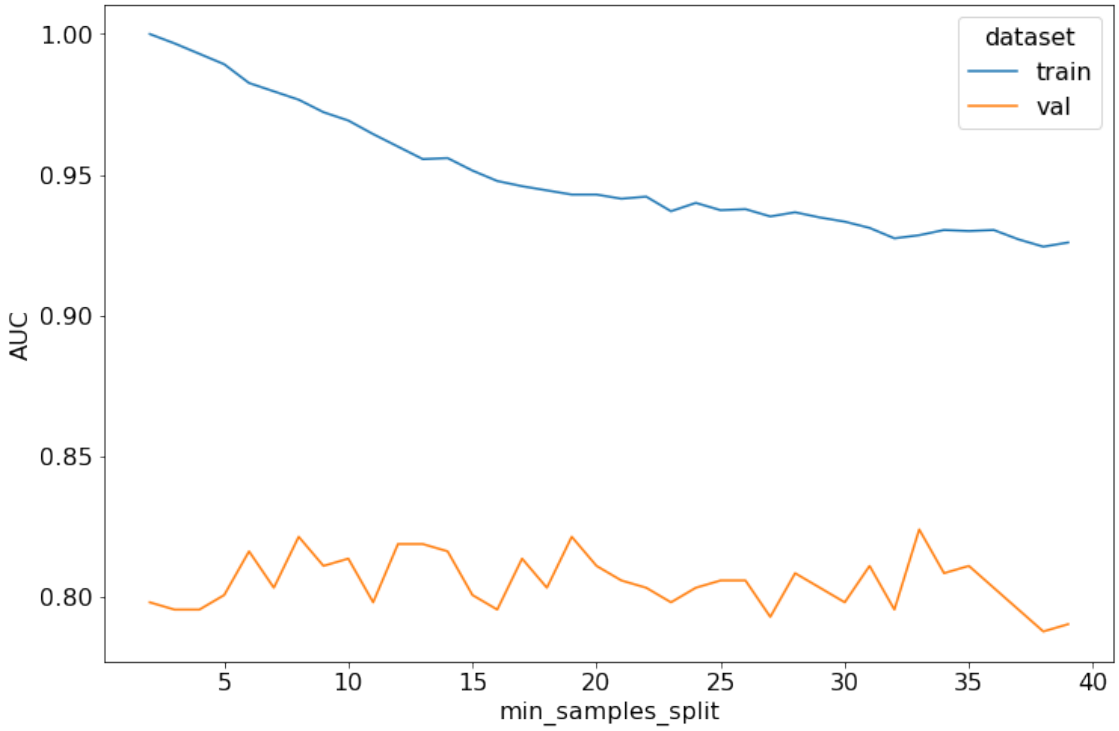
[ ]:
```
cols = ["max_depth","AUC", "dataset"]
history = pd.DataFrame(columns=cols)

n_depth = np.arange(1,15,1)
for depth in tqdm(n_depth):
    dt = DecisionTreeClassifier(criterion='gini', max_depth=depth).
 ↪fit(X_train_log, y_train_log)
    train_score = dt.score(X_train_log, y_train_log)
    val_score = dt.score(X_val_log, y_val_log)
    history = history.append(dict(zip(cols, [depth, train_score, "train"])),␣
 ↪ignore_index=True)
    history = history.append(dict(zip(cols, [depth, val_score, "val"])),␣
 ↪ignore_index=True)

sns.lineplot(data=history, x = "max_depth", y = "AUC", hue = "dataset")
plt.show()
```

```
  0%|          | 0/14 [00:00<?, ?it/s]
```



[ ]:
```
cols = ["min_samples_split","AUC", "dataset"]
history = pd.DataFrame(columns=cols)

n_depth = np.arange(2,40,1)
for depth in tqdm(n_depth):
    dt = DecisionTreeClassifier(criterion='gini', min_samples_split=depth).
 ↪fit(X_train_log, y_train_log)
    train_score = dt.score(X_train_log, y_train_log)
    val_score = dt.score(X_val_log, y_val_log)
    history = history.append(dict(zip(cols, [depth, train_score, "train"])),␣
 ↪ignore_index=True)
    history = history.append(dict(zip(cols, [depth, val_score, "val"])),␣
 ↪ignore_index=True)

sns.lineplot(data=history, x = "min_samples_split", y = "AUC", hue = "dataset")
plt.show()
```

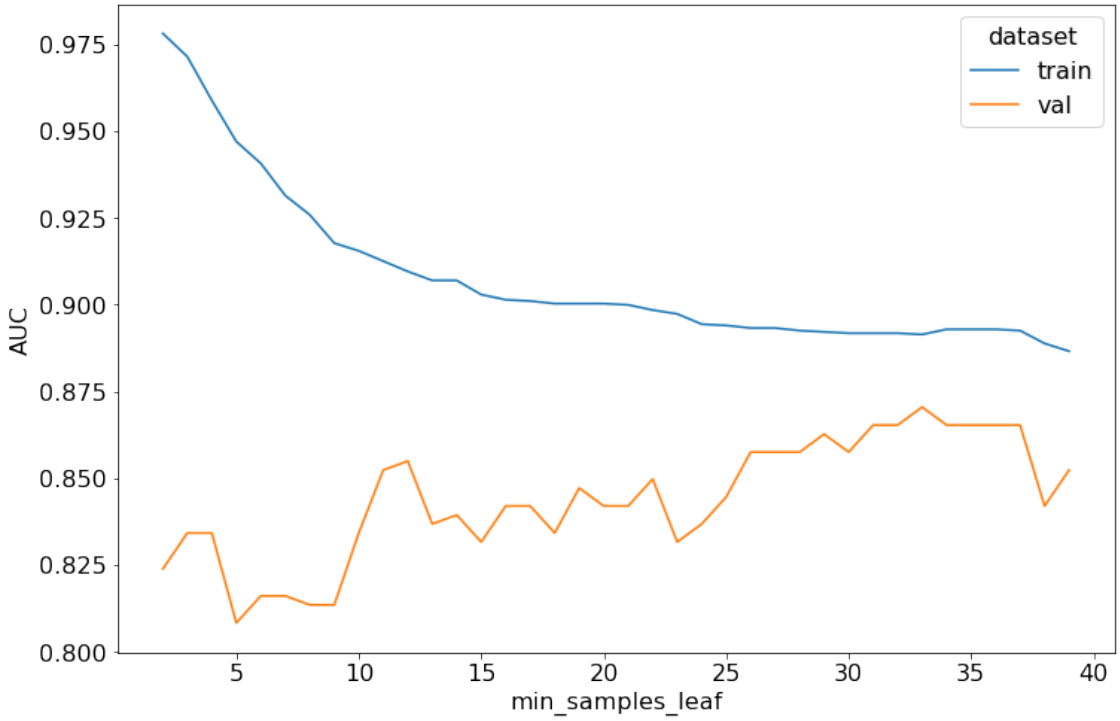
```
  0%|          | 0/38 [00:00<?, ?it/s]
```



[ ]:
```
cols = ["min_samples_leaf","AUC", "dataset"]
history = pd.DataFrame(columns=cols)
```

```
n_depth = np.arange(2,40,1)
for depth in tqdm(n_depth):
    dt = DecisionTreeClassifier(criterion='gini', min_samples_leaf=depth).
 ↪fit(X_train_log, y_train_log)
    train_score = dt.score(X_train_log, y_train_log)
    val_score = dt.score(X_val_log, y_val_log)
    history = history.append(dict(zip(cols, [depth, train_score, "train"])),␣
 ↪ignore_index=True)
    history = history.append(dict(zip(cols, [depth, val_score, "val"])),␣
 ↪ignore_index=True)

sns.lineplot(data=history, x = "min_samples_leaf", y = "AUC", hue = "dataset")
plt.show()
```

```
  0%|          | 0/38 [00:00<?, ?it/s]
```



Wnioski są analogiczne jak dla ramki bez zlogarytmowania.

Predyktory po wstępnym doborze parametrów Wybierzmy parametry, które dają nadzieję na najlepszą predykcyjność przy zachowanym jednocześnie nieprzetrenowanym modelu.

[ ]:
```
# min_samples_split jest nadpisywany przez min_samples_leaf
dt_clf2 = DecisionTreeClassifier(criterion='gini',
                                 max_depth=4,
```

```
                                  min_samples_leaf=30)
dt_clf2.fit(X_train, y_train)

print("-------------------------------------------------")
print("Train set scores")
show_model_metrics(dt_clf2, X_train, y_train)

print("-------------------------------------------------")
print("Validation set scores")
show_model_metrics(dt_clf2, X_val, y_val)
```

```
-------------------------------------------------
Train set scores
F1 score: [0.44776119 0.93909465]
F1 score micro: 0.8902891030392883
F1 score weighted: 0.8738991804833832
Precision score: [0.6741573  0.90555556]
Recall score: [0.33519553 0.97521368]
-------------------------------------------------
Validation set scores
F1 score: [0.37974684 0.92929293]
F1 score micro: 0.8730569948186528
F1 score weighted: 0.8481423403047329
Precision score: [0.68181818 0.88461538]
Recall score: [0.26315789 0.9787234 ]
```

[ ]:
```
dt_clf_log2 = DecisionTreeClassifier(criterion='gini',
                                  max_depth=4,
                                  min_samples_leaf=32)
dt_clf_log2.fit(X_train_log, y_train_log)

print("-------------------------------------------------")
print("Train set scores")
show_model_metrics(dt_clf_log2, X_train_log, y_train_log)

print("-------------------------------------------------")
print("Validation set scores")
show_model_metrics(dt_clf_log2, X_val_log, y_val_log)
```

```
-------------------------------------------------
Train set scores
F1 score: [0.44776119 0.93909465]
F1 score micro: 0.8902891030392883
F1 score weighted: 0.8738991804833832
Precision score: [0.6741573  0.90555556]
Recall score: [0.33519553 0.97521368]
-------------------------------------------------
Validation set scores
```

```
F1 score: [0.37974684 0.92929293]
F1 score micro: 0.8730569948186528
F1 score weighted: 0.8481423403047329
Precision score: [0.68181818 0.88461538]
Recall score: [0.26315789 0.9787234 ]
```

Z powyższych wyników widać, że wszystkie wartości po doborze parametrów są lepsze, jednakże różnica nie jest znacząca.

1.4.6 Wnioski

Jak widać, mimo dość wysokich wyników metryk ważonych, nasz model dość przeciętnie radzi sobie z jego głównym zadaniem - wykrywaniem pacjentów, którzy zmarli. Będzie to naszym priorytetem na dalszym etapie prac. Jeśli chodzi o ramkę z danymi poddanymi transformacji logarytmicznej, nie zmienia to niemal w ogóle wyników modelu.