Praca domowa 3

Autorzy: Daniel Tytkowski, Jan Skwarek

Wstęp

Zacznijmy od zaimportowania standardowych pakietów.

```
import pandas as pd
import numpy as np
import sklearn
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib
import warnings
warnings.filterwarnings('ignore')
# np.random.seed(23)
```

Wczytajmy dane, na których będziemy pracować.

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
df = pd.read_csv('drive/MyDrive/datasets/data.csv')
```

Dokonajmy podziału na zbiór do budowania, testowy i walidacyjny.

```
from sklearn.model_selection import train_test_split
```

```
y = df['Bankrupt?']
```

```
X = df.drop(['Bankrupt?'], axis=1)
```

Dla walidatorów: 'random_state' ustawiony na '420'.

```
X_build, X_val, y_build, y_vali =train_test_split(X, y, test_size=0.3, random_st
```

```
X_train, X_test, y_train, y_test = train_test_split(X_build, y_build, test_size=
```

EDA

Zobaczmy jak wyglądają nasze dane.

```
X_train.head()
```

	ROA(C) before interest and depreciation before interest	ROA(A) before interest and % after tax	ROA(B) before interest and depreciation after tax	Operating Gross Margin	Realized Sales Gross Margin	Operating Profit Rate	I
2544	0.452104	0.522841	0.507040	0.616202	0.617218	0.998884	
2611	0.495881	0.561055	0.545211	0.602077	0.602135	0.999020	
1689	0.556184	0.606029	0.598694	0.613536	0.613536	0.999087	
1062	0.552089	0.605266	0.587879	0.604542	0.604578	0.999031	
5541	0.518598	0.571522	0.563413	0.614357	0.614357	0.999123	

5 rows × 95 columns

```
X_train.shape
```

```
(3341, 95)
```

```
X_train.describe()
```

	ROA(C) before interest and depreciation before interest	ROA(A) before interest and % after tax	ROA(B) before interest and depreciation after tax	Operating Gross Margin	Realized Sales Gross Margin	Operat Pro R
count	3341.000000	3341.000000	3341.000000	3341.000000	3341.000000	3341.00(
mean	0.506394	0.559923	0.554648	0.608057	0.608043	0.99{
std	0.060868	0.065556	0.061606	0.017777	0.017762	0.00{
min	0.000000	0.006923	0.000000	0.000000	0.000000	0.61{
25%	0.476868	0.536361	0.527812	0.600564	0.600556	0.99{
50%	0.504168	0.561110	0.553402	0.606163	0.606149	0.99{
75%	0.537074	0.590057	0.585149	0.613738	0.613723	0.99{

max	0.971530	1.000000	1.000000	1.000000	1.000000	0.999

8 rows × 95 columns

Sprawdźmy braki danych.

```
X_train.isna().sum().sort_values(ascending=False)
```

```
 ROA(C) before interest and depreciation before interest    0
 Operating Funds to Liability                               0
 Total expense/Assets                                       0
 Total income/Total expense                                 0
 Retained Earnings to Total Assets                          0
                                                           ..
 Net Value Growth Rate                                      0
 Total Asset Growth Rate                                    0
 Continuous Net Profit Growth Rate                          0
 Regular Net Profit Growth Rate                             0
 Equity to Liability                                        0
Length: 95, dtype: int64
```

```
X_train.nunique().sort_values(ascending=False)
```

```
 Equity to Liability                  3341
 CFO to Assets                        3341
 Cash Flow to Total Assets            3341
 Current Liability to Equity          3341
 Current Liabilities/Equity           3341
                                      ...
 Interest-bearing debt interest rate   877
 Net Worth Turnover Rate (times)       579
 Total Asset Turnover                  323
 Liability-Assets Flag                   2
 Net Income Flag                         1
Length: 95, dtype: int64
```

'Net Income Flag' w naszym zbiorze przyjmuje tylko jedną wartość, natomiast 'Liability-Assets Flag' przyjmuje tylko dwie wartości. Przyjrzyjmy się im bardziej.

```
X_train[' Liability-Assets Flag'].value_counts()
```

```
0    3336
1       5
Name:  Liability-Assets Flag, dtype: int64
```

Tylko sześć wartości przyjmuje wartość '1'. Obejrzyjmy je sobie.

```
X_train.loc[df[' Liability-Assets Flag'] == 1]
```

	ROA(C)	ROA(A)	

	ROA(C) before interest and depreciation before interest	ROA(A) before interest and % after tax	ROA(B) before interest and depreciation after tax	Operating Gross Margin	Realized Sales Gross Margin	Operating Profit Rate	]
56	0.066933	0.057185	0.054821	0.601861	0.601861	0.998825	
2001	0.438795	0.090166	0.464586	0.540776	0.540776	0.997789	
6613	0.279676	0.283362	0.303014	0.637520	0.637520	0.998785	
2735	0.436894	0.453718	0.479522	0.585062	0.585062	0.998495	
6640	0.196802	0.211023	0.221425	0.598056	0.598056	0.998933	

5 rows × 95 columns

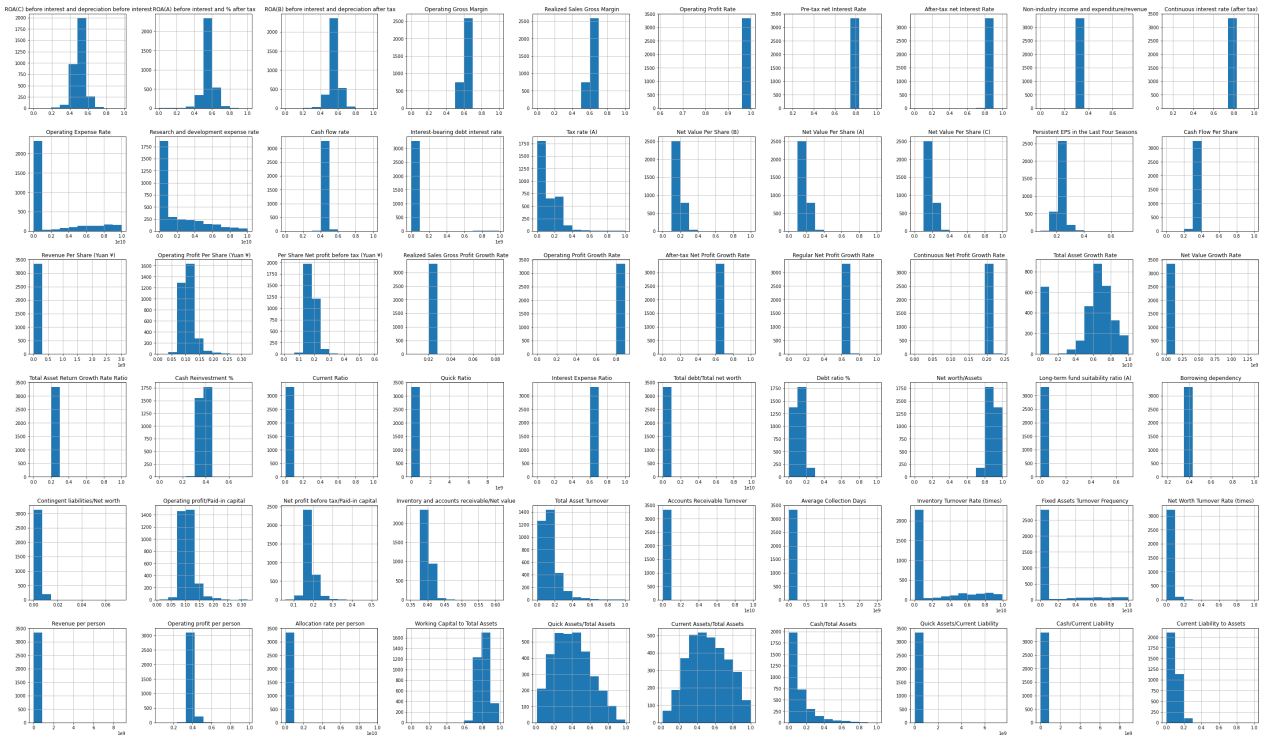
Nie róbmy na razie nic z tą zmienną (wedle polecenia EDA ma być - podobnie jak preprocessing - szczątkowa).

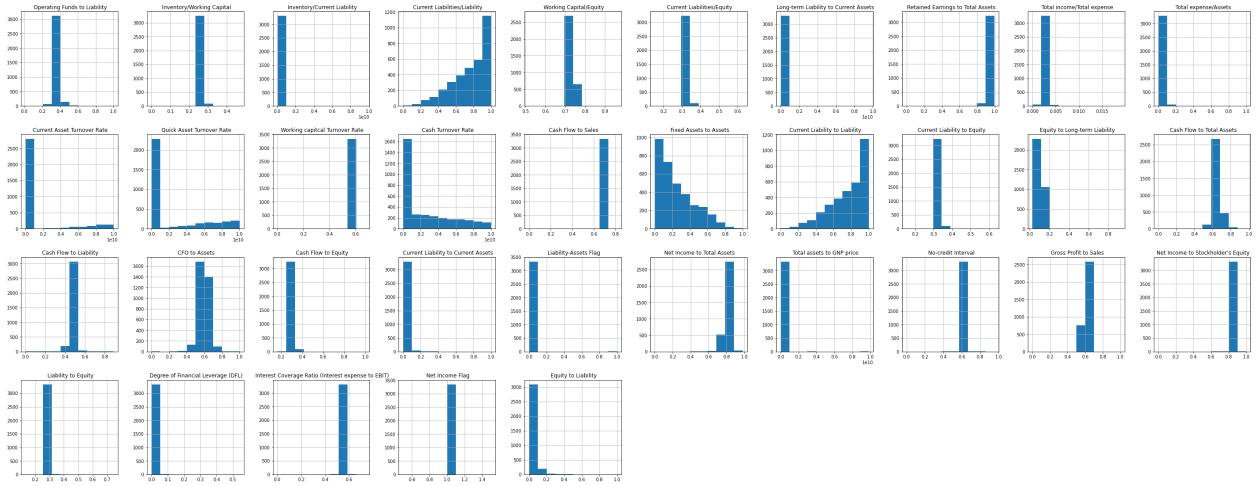
```
X_train[' Net Income Flag'].value_counts()
```

```
1    3341
Name:  Net Income Flag, dtype: int64
```

Jedna wartość zmiennej dla każdego rzędu. Bezużyteczna dla nas zmienna. Zobaczmy histogramy.

```
X_train.hist(figsize=(50, 50))
plt.show()
```





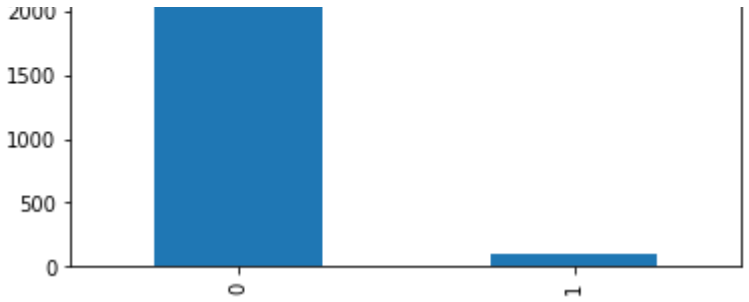
Zbadajmy jeszcze zmienną celu.

```
y_train.head()
```

```
2544    0
2611    0
1689    0
1062    0
5541    0
Name: Bankrupt?, dtype: int64
```

```
y_train.value_counts().plot(kind="bar")
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f599d0c8ad0>
```




```
y_train.value_counts()
```

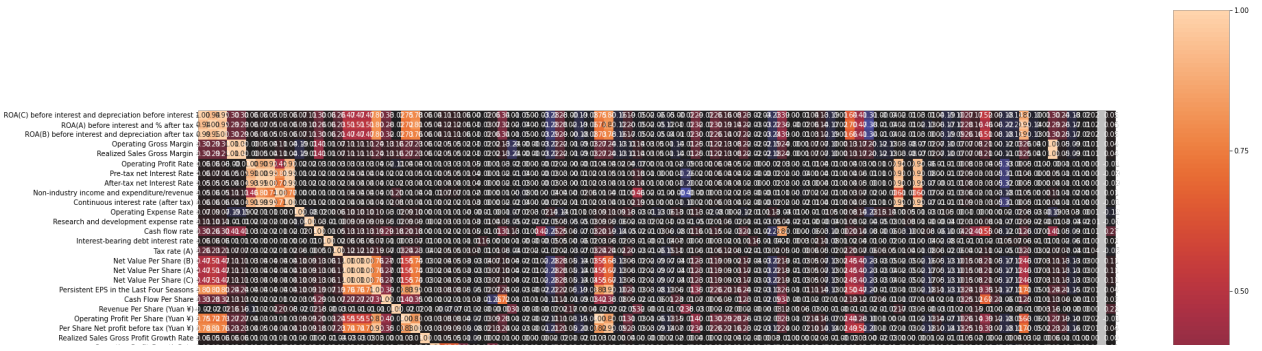
```
0    3242
1      99
Name: Bankrupt?, dtype: int64
```

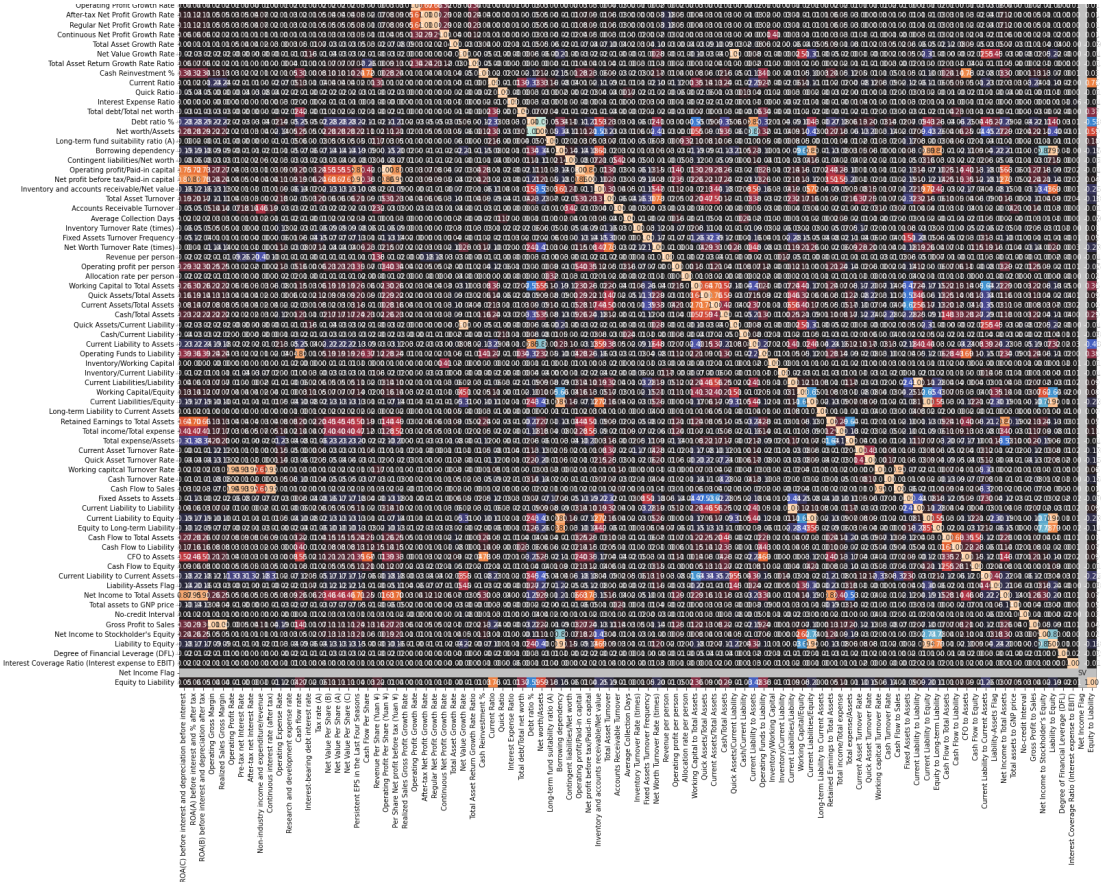
Jak można było się spodziewać, zmienna celu nie jest równomiernie rozłożona. Zobaczmy sobie na koniec jak zmienne ze sobą korelują.

```
! pip install dython
from dython.nominal import associations
```

```
Requirement already satisfied: dython in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: numpy>=1.19.5 in /usr/local/lib/python3.7/di
Requirement already satisfied: seaborn>=0.11.0 in /usr/local/lib/python3.7/
Requirement already satisfied: pandas>=1.3.2 in /usr/local/lib/python3.7/di
Requirement already satisfied: scipy>=1.7.1 in /usr/local/lib/python3.7/dis
Requirement already satisfied: matplotlib>=3.4.3 in /usr/local/lib/python3.
Requirement already satisfied: scikit-plot>=0.3.7 in /usr/local/lib/python3
Requirement already satisfied: scikit-learn>=0.24.2 in /usr/local/lib/pytho
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.7/di
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.7/
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/pytho
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dis
Requirement already satisfied: pyparsing>=2.2.1 in /usr/local/lib/python3.7
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.
Requirement already satisfied: pytz>=2017.3 in /usr/local/lib/python3.7/dis
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.7/dis
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/pytho
```

```
complete_correlation = associations(X_train, figsize=(30,30))
```





Preprocessing

Zastąpmy sobie skrajne wartości percentylami. Postąpimy tak dla każdej zmiennej, która nie jest dyskretna.

```
for col in X_train[X_train.columns.drop([' Liability-Assets Flag', ' Net Income
```

```
  p_025 = X_train[col].quantile(0.025) # 5th quantile
  p_975 = X_train[col].quantile(0.975) # 95th quantile
  X_train[col].clip(p_025, p_975, inplace=True)
  #od razu na testowym to samo, zeby miec te same kwantyle
  X_test[col].clip(p_025, p_975, inplace=True)
```

Zmieńmy kierunek zmiennych ujemnie skorelowanych.

```
import scipy.stats as ss
```

```
X_train['Bankrupt?']=y_train
```

```
for column in X_train.columns:
  if(ss.pearsonr(X_train['Bankrupt?'], X_train[column])[0]<0):
    X_train[column]=-X_train[column]
# korelacja Pearsona
```

```
sns.heatmap(X_train.corr())
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f7836424ad0>
```


```
X_train = X_train.drop(['Bankrupt?'],axis=1)
```

Znormalizujmy dane MinMaxScaler'em.

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
```

```
X_train.loc[:, ~X_train.columns.isin([' Net Income Flag', ' Liability-Assets Fla
```

```
X_train
```

	ROA(C) before interest and depreciation before interest	ROA(A) before interest and % after tax	ROA(B) before interest and depreciation after tax	Operating Gross Margin	Realized Sales Gross Margin	Operating Profit Rate	]
2544	0.718692	0.596176	0.669927	0.478572	0.456413	0.520554	
2611	0.540387	0.452746	0.516984	0.758983	0.756950	0.358576	
1689	0.294770	0.283944	0.302691	0.531507	0.529788	0.279377	
1062	0.311449	0.286809	0.346021	0.710054	0.708272	0.345766	
5541	0.447859	0.413461	0.444051	0.515197	0.513419	0.235853	
...	...	...	...	...	...	...	
4067	0.502462	0.418372	0.473653	0.706334	0.705257	0.326208	
1189	0.559448	0.471365	0.518056	0.805623	0.804909	0.369319	
2604	0.618023	0.531724	0.574472	0.630652	0.629297	0.335436	
2472	0.419266	0.322411	0.392355	0.737809	0.736847	0.327998	
1890	0.705786	0.581035	0.678936	0.773576	0.772745	0.444661	

3341 rows × 95 columns

Dokonajmy wszystkich powyższych operacji na zbiorze testowym.

```
#zastapienie outlierów już wcześniej zrobione

#zmiana kierunku
X_test['Bankrupt?']=y_test
for column in X_test.columns:
  if(ss.pearsonr(X_test['Bankrupt?'], X_test[column])[0]<0):
    X_test[column]=-X_test[column]
X_test = X_test.drop(['Bankrupt?'],axis=1)

#scaler
```

```
X_test.loc[:, ~X_test.columns.isin([' Net Income Flag', ' Liability-Assets Flag'
```

Modele

Zaimportujmy niezbędne pakiety.

```
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
```

Model 1 - regresja logistyczna

Pierwszy modelem, jaki nauczymy, będzie regresja logistyczna.

```
model1 = LogisticRegression()
model1.fit(X_train,y_train)
```

```
LogisticRegression()
```

Użyjmy teraz trzech różnych metryk do przebadania efektywności wyszkolonego przez nas modelu:

1. Pierwszą z nich będzie 'accuracy score' - dokładność jest najpowszechniejszą metryką do oceny jakości klasyfikacji. Jest prosta w zrozumieniu i interpretacji.
2. Kolejną metryką będzie 'f1 score' - średnia harmoniczna 'precision' i 'recall'.
3. 'ROC AUC score' - ROC jest krzywą prawdopodobieństwa, a AUC reprezentuje stopień lub miarę rozdzielności. Mówi, jak bardzo model jest w stanie rozróżnić klasy.

```
from sklearn.metrics import accuracy_score
from sklearn.metrics import f1_score
from sklearn.metrics import roc_auc_score
```

```
print(model1.score(X_train, y_train))
print(model1.score(X_test, y_test))
```

```
0.9760550733313379
0.9650837988826816
```

```
print(f1_score(y_train, model1.predict(X_train)))
print(f1_score(y_test, model1.predict(X_test)))
```

```
0.36507936507936506
0.3055555555555556
```

```
print(roc_auc_score(y_train, model1.predict(X_train)))
print(roc_auc_score(y_test, model1.predict(X_test)))
```

```
0.6155447130154101
0.6214274951027174
```

```
y_train.value_counts()
```

```
0    3242
1      99
Name: Bankrupt?, dtype: int64
```

```
y_test.value_counts()
```

```
0    1389
1      43
Name: Bankrupt?, dtype: int64
```

Nie jest źle. Szczególnie dobry wynik uzyskaliśmy przy metryce 'accuracy score'. Metryka f1 daje słaby wynik prawdopobonie przez to, że jest dużo mniej 1 w zmiennej docelowej(tak samo będzie w kolejnych modelach). Zastanówmy się nad hiperparametrami. Przeanalizujmy jakie przyjmuje regresja logistyczna.

Okazuje się, że regresja logistyczna nie ma żadnych krytycznie ważnych hiperparametrów do ustawiania. Na jej wyniki mogą jednak najbardziej wpłynąć dwa hiperparametry:

1. 'solver' - algorytm do wykorzystania w zadaniu optymalizacyjnym. Domyślnie jest to „lbfgs”.
2. 'penalty' - norma kary.

Zaimportujmy grid searcha i znajdźmy najlepszą konfigurację.

```
from sklearn.model_selection import GridSearchCV
```

```
solvers = ['newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga']
penalties = ['l1', 'l2', 'elasticnet', 'none']
param_grid = dict(solver = solvers, penalty = penalties)

lg_model = LogisticRegression()
grid = GridSearchCV(estimator = lg_model, param_grid = param_grid,
                    cv = 3, n_jobs = -1, verbose = 2)

grid_result = grid.fit(X_train, y_train)
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
```

```
Fitting 3 folds for each of 20 candidates, totalling 60 fits
```

```
Fitting 3 folds for each of 20 candidates, totalling 60 fits
Best: 0.973660 using {'penalty': 'l2', 'solver': 'newton-cg'}
```

Są to najlepsze parametry. Co ciekawe, przy wszystkich hiperparametrach ustawionych domyślnie, wynik był odrobinę lepszy.

Model 2 - K najbliższych sąsiadów

Z 'KNeigborsClassifier' postąpimy analogicznie jak w przypadku regresji logistycznej.

```
model2 = KNeighborsClassifier()
model2.fit(X_train, y_train)
```

```
KNeighborsClassifier()
```

```
print(model2.score(X_train, y_train))
print(model2.score(X_test, y_test))
```

```
0.9748578269979048
0.9671787709497207
```

```
print(f1_score(y_train, model2.predict(X_train)))
print(f1_score(y_test, model2.predict(X_test)))
```

```
0.3333333333333333
0.2295081967213115
```

```
print(roc_auc_score(y_train, model2.predict(X_train)))
print(roc_auc_score(y_test, model2.predict(X_test)))
```

```
0.605135251341297
0.5774356656118673
```

Zastanówmy się znów nad hiperparametrami. Oto kluczowe z nich oferowane przez 'KNeighborsClassifier':

1. 'n_neighbors' - wybierz najlepsze k na podstawie wartości, które obliczyliśmy wcześniej.
2. 'weights' - sprawdź, czy dodawanie wag do punktów danych jest korzystne dla modelu, czy nie. „Uniform” nie przypisuje wagi, podczas gdy „distance” waży punkty przez odwrotność ich odległości, co oznacza, że bliższe punkty będą miały większą wagę niż dalsze punkty.
3. 'metric' - metryka odległości, która zostanie użyta, obliczy podobieństwo.

```
n_neighbors = [5, 7, 9, 11, 13, 15]
weights = ['uniform', 'distance']
```

```
metric = ['minkowski', 'euclidean', 'manhattan']
param_grid = dict(n_neighbors = n_neighbors, weights = weights, metric = metric)

knn_model = KNeighborsClassifier()
grid = GridSearchCV(estimator = knn_model, param_grid = param_grid,
                    cv = 3, n_jobs = -1, verbose = 2)

grid_result = grid.fit(X_train, y_train)
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
```

```
Fitting 3 folds for each of 36 candidates, totalling 108 fits
Best: 0.972163 using {'metric': 'manhattan', 'n_neighbors': 7, 'weights': '
```

Najlepsze rezultaty osiągnęły powyższe parametry.

Model 3 - Random Forest

```
model3 = RandomForestClassifier()
model3.fit(X_train, y_train)
```

```
RandomForestClassifier()
```

```
print(model3.score(X_train, y_train))
print(model3.score(X_test, y_test))
```

```
1.0
0.9664804469273743
```

```
print(f1_score(y_train, model3.predict(X_train)))
print(f1_score(y_test, model3.predict(X_test)))
```

```
1.0
0.22580645161290322
```

```
print(roc_auc_score(y_train, model3.predict(X_train)))
print(roc_auc_score(y_test, model3.predict(X_test)))
```

```
1.0
0.5770756944095635
```

Zastanówmy się znów nad hiperparametrami. Oto kluczowe z nich oferowane przez 'RandomForestClassifier':

1. 'bootstrap' - czy podczas budowania drzew używane są próbki bootstrap.
2. 'max_depth' - maksymalna głębokość drzewa.
3. 'max_features' - liczba funkcji, które należy wziąć pod uwagę, szukając najlepszego podziału.
4. 'min_samples_leaf' - minimalna liczba próbek, które muszą znajdować się w węźle

4. 'min_samples_leaf' - minimalna liczba próbek, które muszą znajdować się w węźle liścia.
5. 'min_samples_split' - minimalna liczba próbek wymagana do podziału węzła wewnętrznego.
6. 'n_estimators' - ilość drzew w lesie.

```
param_grid = {
    'bootstrap': [True],
    'max_depth': [80, 90, 100, 110],
    'max_features': [2, 3],
    'min_samples_leaf': [3, 4, 5],
    'min_samples_split': [8, 10, 12],
    'n_estimators': [100, 200, 300, 1000]
}
rf = RandomForestClassifier()
grid = GridSearchCV(estimator = rf, param_grid = param_grid,
                           cv = 3, n_jobs = -1, verbose = 2)
grid_result = grid.fit(X_train, y_train)
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
```

```
    Fitting 3 folds for each of 288 candidates, totalling 864 fits
    Best: 0.972763 using {'bootstrap': True, 'max_depth': 90, 'max_features': 3
```

Te hiperparametry wydają się być najlepsze.

Najlepsze hiperparametry

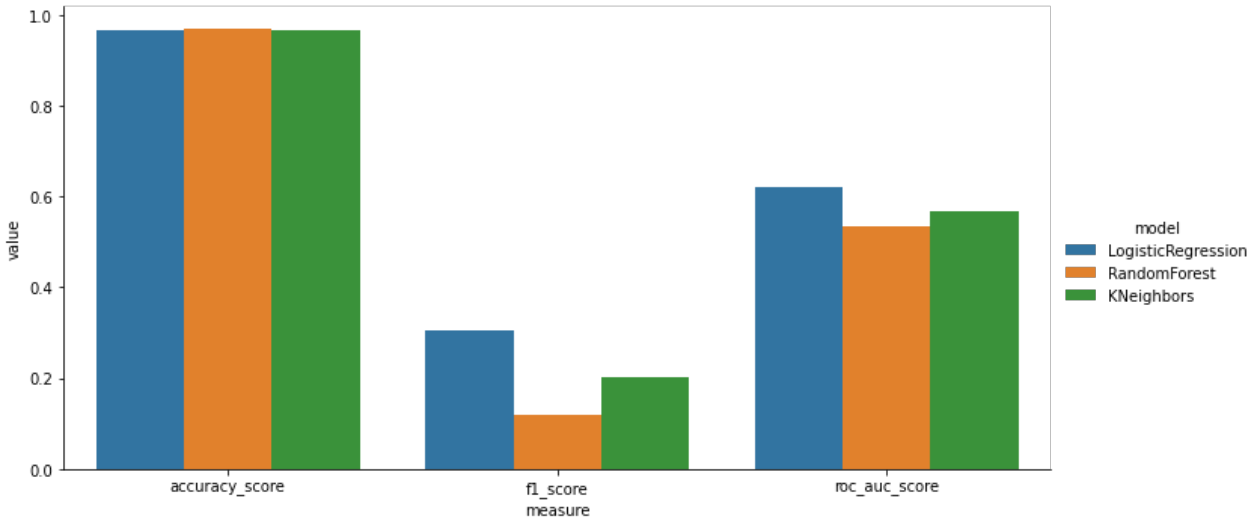
Posłużyliśmy się Grid Search'em, aby znaleźć najlepsze hiperparametry dla poszczególnych modeli. Oto one:

```
lr_best = LogisticRegression(penalty = 'l2', solver = 'newton-cg')
kn_best = KNeighborsClassifier(metric = 'manhattan', n_neighbors = 7, weights =
rf_best = RandomForestClassifier(bootstrap = True, max_depth = 90, max_features
```

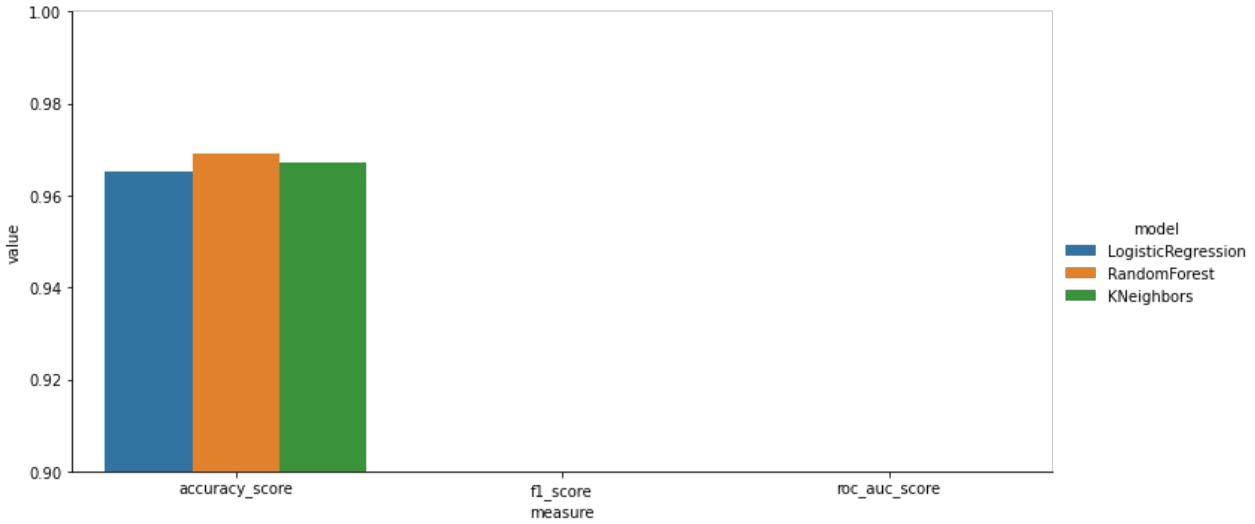
```
test_results = []
for model, model_name in [(lr_best, "LogisticRegression"), (rf_best, "RandomFore
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    measures_results = {"model": model_name}
    for measure in [accuracy_score, f1_score, roc_auc_score]:
        measures_results[measure.__name__] = measure(y_test, y_pred)
    test_results.append(measures_results)

test_results_df = pd.DataFrame(test_results)
```

```
test_results_df = pd.melt(test_results_df, id_vars="model", var_name="measure",
```

```
sns.catplot(x='measure', y='value', hue='model', data=test_results_df, kind='bar
plt.show()
```




```
sns.catplot(x='measure', y='value', hue='model', data=test_results_df, kind='bar
plt.ylim(0.90, 1)
plt.show()
```



Na podstawie powyższego wydaje się, że regresja logistyczna wypada tutaj zdecydowanie najlepiej. Co prawda jej 'accuracy_score' jest najmniejszy, ale wysuwa się na prowadzenie w dwóch pozostałych metrykach. 'accuracy score' przyjął takie dobre wartości prawdopodobnie z powodu tego, iż jedna z klas była znacznie liczniejsza od drugiej (faktycznie tak było)

Odniesienie się do wniosków z walidacji

W tym paragrafie postaramy się odnieść do uwag wystosowanych do nas przez zespół walidacyjny. Będziemy to robić kolejno w punktach. Większość z nich to drobne uwagi (dotyczące między innymi nazewnictwa, subiektywnego braku zrozumienia pewnych idei, pojawiły się też uwagi, które były najzwyczajniej błędne - odniesiemy się jednak do każdej).

1. Dwa razy sprawdzacie braki danych na X_train.

- Poprawione

2. random_seed=42 (nie 420)

- Nie ma to większego znaczenia. 'random_seed=42' przyjął się powszechnie z uwagi na pewne odniesienie kulturowe. Nie jest to jednak błąd merytoryczny.

3. 5 wartości przyjmuje Liability-Assets Flag (nie 6).

- Liability-Assets Flag przyjmuje dwie wartości.

4. Nieczytelne histogramy.

- w naszej opinii są czytelne. Krótkie EDA nie miało na zasadzie szczegółowe zobrazowanie każdej zmiennej, a pewien ogólny obrazek jak mniej więcej te wartości wyglądają.

5. Niezbalansowane klasy.

- Niestety z uwagi na ograniczony czas nie mieliśmy czasu zrobić w pełni optymalnego podziału.

6. Z macierzy korelacji mało co można odcytać, jaki rodzaj korelacji liczony.

- Można odczytać rodzaj korelacji (dla pewności dodatkowo go opisałem w komentarzu). Ponadto, macierz ta, podobnie jak histogramy, ma charakter poglądowy.

7. Oddzielny scaler dla danych testowych.

- Nieprawda, scaler jest ten sam (co prawda została stworzona dodatkowa instancja MinMaxScaler'a, lecz nie została ona użyta - już ją usunąłem bo była to mimo wszystko

niepotrzebna zmienna).

8. ROC nie jest krzywą prawdopobobieństwa

- Można ją tak kolokwialnie nazwać.

9. Gini zamiast AUC i wszystkie uwagi odnośnie wyboru metryk

- Wybór metryk był dowolny. Nie musiał być w pełni optymalny, mimo że staraliśmy się, żeby był.

10. Przy outlierach policzyli kwantyle na całym zbiorze danych a nie tylko X_train

- Poprawione