Klasyfikacja niezbalansowana, klasyfikatory zespołowe i wyjaśnialna AI

Wykorzystanie Google Colab

Jeśli korzystasz z Google Colab skopiuj plik `feature_names.json` do katalogu głównego projektu.

Ładowanie i eksploracja danych

Na tym laboratorium wykorzystamy zbiór danych Polish companies bankruptcy. Dotyczy on klasyfikacji, na podstawie danych z raportów finansowych, czy firma zbankrutuje w ciągu najbliższych kilku lat. Jest to zadanie szczególnie istotne dla banków, funduszy inwestycyjnych, firm ubezpieczeniowych itp., które z tego powodu zatrudniają licznie data scientistów. Zbiór zawiera 64 cechy, obliczone przez ekonomistów, którzy stworzyli ten zbiór, są one opisane na podlinkowanej wcześniej stronie. Dotyczą one zysków, posiadanych zasobów oraz długów firm.

Ściągnij i rozpakuj dane (`Data Folder` -> `data.zip`) do katalogu `data` obok tego notebooka. Znajduje się tam 5 plików w formacie `.arff`, wykorzystywanym głównie przez oprogramowanie Weka. Jest to program do "klikania" ML w interfejsie graficznym, jakiś czas temu popularny wśród mniej technicznych data scientistów. W Pythonie ładuje się je za pomocą bibliotek SciPy i Pandas.

Jeśli korzystasz z Linuksa możesz skorzystać z poniższych poleceń do pobrania i rozpakowania tych plików.

```
!mkdir -p data
!wget
https://archive.ics.uci.edu/static/public/365/polish+companies+bankrup
tcy+data.zip -O data/data.zip
```

```
A subdirectory or file data already exists.
Error occurred while processing: data.
'wget' is not recognized as an internal or external command,
operable program or batch file.
```

```
!unzip data/data.zip -d data
```

```
'unzip' is not recognized as an internal or external command,
operable program or batch file.
```

W dalszej części laboratorium wykorzystamy plik `3year.arff`, w którym na podstawie finansowych firmy po 3 latach monitorowania chcemy przewidywać, czy firma zbankrutuje w ciągu najbliższych 3 lat. Jest to dość realistyczny horyzont czasowy.

Dodatkowo w pliku `feature_names.json` znajdują się nazwy cech. Są bardzo długie, więc póki co nie będziemy z nich korzystać.

```
import json
import os

from scipy.io import arff
import pandas as pd

data = arff.loadarff(os.path.join("data", "3year.arff"))

with open("feature_names.json") as file:
    feature_names = json.load(file)

X = pd.DataFrame(data[0])
```

Przyjrzyjmy się teraz naszym danym.

```
X.head()
```

```
      Attr1    Attr2    Attr3   Attr4    Attr5    Attr6     Attr7
Attr8  \
0  0.174190  0.41299  0.14371  1.3480 -28.9820  0.60383  0.219460
1.1225
1  0.146240  0.46038  0.28230  1.6294   2.5952  0.00000  0.171850
1.1721
2  0.000595  0.22612  0.48839  3.1599  84.8740  0.19114  0.004572
2.9881
3  0.024526  0.43236  0.27546  1.7833 -10.1050  0.56944  0.024526
1.3057
4  0.188290  0.41504  0.34231  1.9279 -58.2740  0.00000  0.233580
1.4094

    Attr9   Attr10  ...    Attr56    Attr57   Attr58    Attr59  Attr60
\
0  1.1961  0.46359  ...  0.163960  0.375740  0.83604  0.000007  9.7145

1  1.6018  0.53962  ...  0.027516  0.271000  0.90108  0.000000  5.9882

2  1.0077  0.67566  ...  0.007639  0.000881  0.99236  0.000000  6.7742

3  1.0509  0.56453  ...  0.048398  0.043445  0.95160  0.142980  4.2286

4  1.3393  0.58496  ...  0.176480  0.321880  0.82635  0.073039  2.5912


   Attr61   Attr62  Attr63  Attr64  class
0  6.2813   84.291  4.3303  4.0341   b'0'
1  4.1103  102.190  3.5716  5.9500   b'0'
2  3.7922   64.846  5.6287  4.4581   b'0'
```

```
3  5.0528   98.783  3.6950  3.4844   b'0'
4  7.0756  100.540  3.6303  4.6375   b'0'

[5 rows x 65 columns]
```

```
X.dtypes
```

```
Attr1     float64
Attr2     float64
Attr3     float64
Attr4     float64
Attr5     float64
           ...
Attr61    float64
Attr62    float64
Attr63    float64
Attr64    float64
class      object
Length: 65, dtype: object
```

```
X.describe()
```

```
              Attr1         Attr2         Attr3         Attr4
Attr5  \
count  10503.000000  10503.000000  10503.000000  10485.000000
1.047800e+04
mean       0.052844      0.619911      0.095490      9.980499 -
1.347662e+03
std        0.647797      6.427041      6.420056    523.691951
1.185806e+05
min      -17.692000      0.000000   -479.730000      0.002080 -
1.190300e+07
25%        0.000686      0.253955      0.017461      1.040100 -
5.207075e+01
50%        0.043034      0.464140      0.198560      1.605600
1.579300e+00
75%        0.123805      0.689330      0.419545      2.959500
5.608400e+01
max       52.652000    480.730000     17.708000  53433.000000
6.854400e+05

              Attr6         Attr7         Attr8         Attr9
Attr10  \
count  10503.000000  10503.000000  10489.000000  10500.000000
10503.000000
mean      -0.121159      0.065624     19.140113      1.819254
0.366093
std        6.970625      0.651152    717.756745      7.581659
6.428603
min     -508.120000    -17.692000     -2.081800     -1.215700   -
```

```
479.730000
25%        0.000000      0.002118      0.431270      1.011275
0.297340
50%        0.000000      0.050945      1.111000      1.199000
0.515500
75%        0.072584      0.142275      2.857100      2.059100
0.725635
max       45.533000     52.652000  53432.000000    740.440000
11.837000

       ...         Attr55        Attr56        Attr57        Attr58  \
count  ...   1.050300e+04  10460.000000  10503.000000  10474.000000
mean   ...   6.638549e+03     -0.530082     -0.014817      3.848794
std    ...   5.989196e+04     55.978608     18.684047    190.201224
min    ... -7.513800e+05  -5691.700000  -1667.300000   -198.690000
25%    ...   1.462100e+01      0.005137      0.006796      0.875560
50%    ...   8.822900e+02      0.051765      0.106880      0.953060
75%    ...   4.348900e+03      0.130010      0.271310      0.995927
max    ...   3.380500e+06    293.150000    552.640000  18118.000000

              Attr59        Attr60        Attr61        Attr62
Attr63  \
count  10503.000000  9.911000e+03  10486.000000  1.046000e+04
10485.000000
mean       1.429319  5.713363e+02     13.935361  1.355370e+02
9.095149
std       77.273270  3.715967e+04     83.704103  2.599116e+04
31.419096
min     -172.070000  0.000000e+00     -6.590300 -2.336500e+06     -
0.000156
25%        0.000000  5.533150e+00      4.486075  4.073700e+01
3.062800
50%        0.002976  9.952100e+00      6.677300  7.066400e+01
5.139200
75%        0.240320  2.093600e+01     10.587500  1.182200e+02
8.882600
max     7617.300000  3.660200e+06   4470.400000  1.073500e+06
1974.500000

              Attr64
count  10275.000000
mean      35.766800
std      428.298315
min       -0.000102
25%        2.023350
50%        4.059300
75%        9.682750
max    21499.000000

[8 rows x 64 columns]
```

```
feature_names

['net profit / total assets',
 'total liabilities / total assets',
 'working capital / total assets',
 'current assets / short-term liabilities',
 '[(cash + short-term securities + receivables - short-term
liabilities) / (operating expenses - depreciation)] * 365',
 'retained earnings / total assets',
 'EBIT / total assets',
 'book value of equity / total liabilities',
 'sales / total assets',
 'equity / total assets',
 '(gross profit + extraordinary items + financial expenses) / total
assets',
 'gross profit / short-term liabilities',
 '(gross profit + depreciation) / sales',
 '(gross profit + interest) / total assets',
 '(total liabilities * 365) / (gross profit + depreciation)',
 '(gross profit + depreciation) / total liabilities',
 'total assets / total liabilities',
 'gross profit / total assets',
 'gross profit / sales',
 '(inventory * 365) / sales',
 'sales (n) / sales (n-1)',
 'profit on operating activities / total assets',
 'net profit / sales',
 'gross profit (in 3 years) / total assets',
 '(equity - share capital) / total assets',
 '(net profit + depreciation) / total liabilities',
 'profit on operating activities / financial expenses',
 'working capital / fixed assets',
 'logarithm of total assets',
 '(total liabilities - cash) / sales',
 '(gross profit + interest) / sales',
 '(current liabilities * 365) / cost of products sold',
 'operating expenses / short-term liabilities',
 'operating expenses / total liabilities',
 'profit on sales / total assets',
 'total sales / total assets',
 'constant capital / total assets',
 'profit on sales / sales',
 '(current assets - inventory - receivables) / short-term
liabilities',
 'total liabilities / ((profit on operating activities + depreciation)
* (12/365))',
 'profit on operating activities / sales',
 'rotation receivables + inventory turnover in days',
 '(receivables * 365) / sales',
 'net profit / inventory',
```

```
 '(current assets - inventory) / short-term liabilities',
 '(inventory * 365) / cost of products sold',
 'EBITDA (profit on operating activities - depreciation) / total
assets',
 'EBITDA (profit on operating activities - depreciation) / sales',
 'current assets / total liabilities',
 'short-term liabilities / total assets',
 '(short-term liabilities * 365) / cost of products sold)',
 'equity / fixed assets',
 'constant capital / fixed assets',
 'working capital',
 '(sales - cost of products sold) / sales',
 '(current assets - inventory - short-term liabilities) / (sales -
gross profit - depreciation)',
 'total costs / total sales',
 'long-term liabilities / equity',
 'sales / inventory',
 'sales / receivables',
 '(short-term liabilities * 365) / sales',
 'sales / short-term liabilities',
 'sales / fixed assets']
```

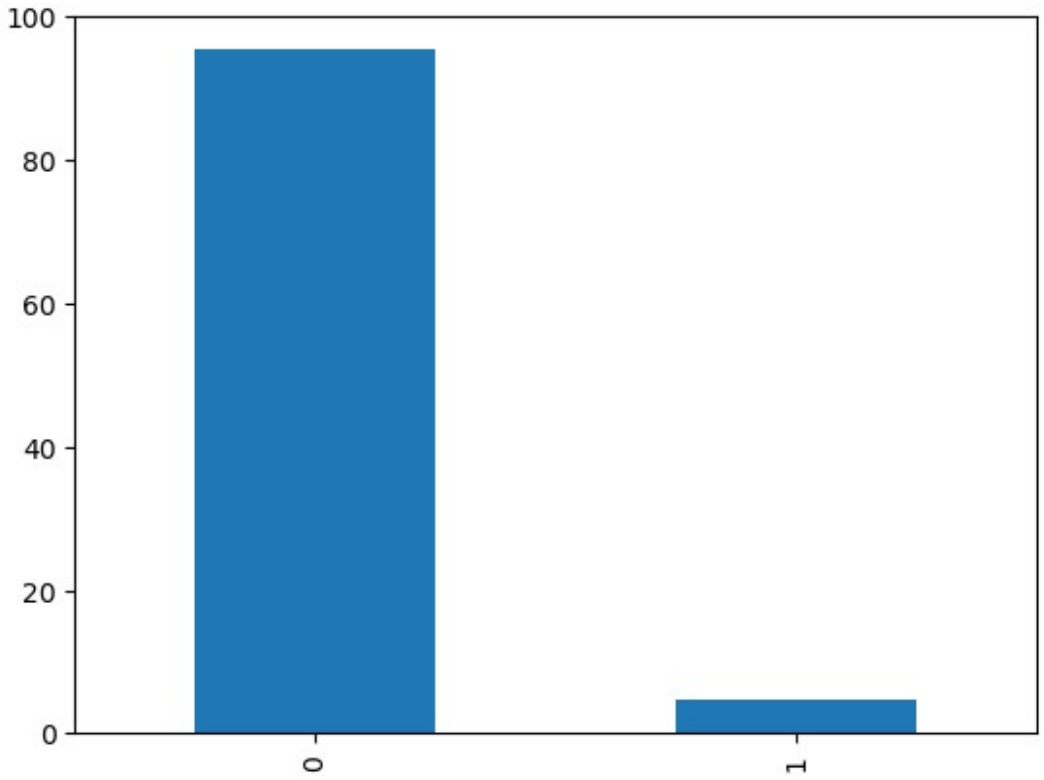
DataFrame zawiera 64 atrybuty numeryczne o zróżnicowanych rozkładach wartości oraz kolumnę `"class"` typu `bytes` z klasami 0 i 1. Wiemy, że mamy do czynienia z klasyfikacją binarną - klasa 0 to brak bankructwa, klasa 1 to bankructwo w ciągu najbliższych 3 lat. Przyjrzyjmy się dokładniej naszym danym.

Zadanie 1 (0.5 punktu)

1. Wyodrębnij klasy jako osobną zmienną typu `pd.Series`, usuwając je z macierzy `X`. Przekonwertuj go na liczby całkowite.
2. Narysuj wykres słupkowy (bar plot) częstotliwości obu klas w całym zbiorze. Upewnij się, że na osi X są numery lub nazwy klas, a oś Y ma wartości w procentach.

```
# your_code
y=X.pop("class").astype(int)
counts= y.value_counts(normalize=True)
percentages= (counts / counts.sum()) * 100
percentages.plot(kind="bar")
```

```
<AxesSubplot:>
```

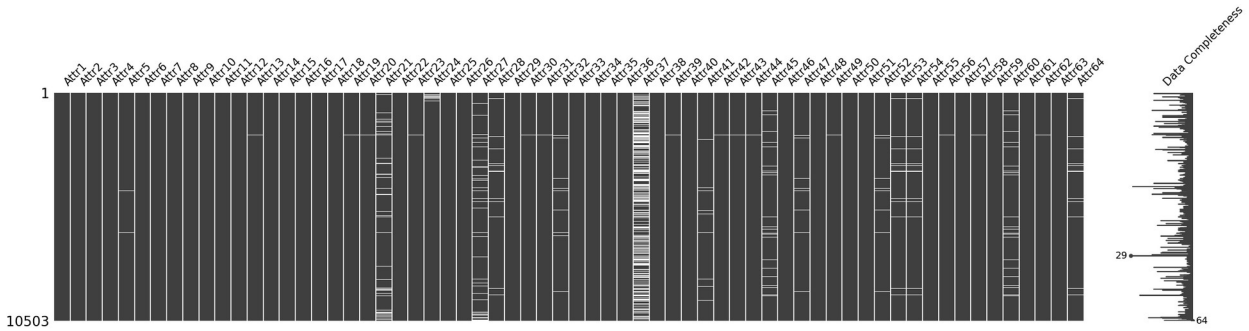


Jak widać, klasa pozytywna jest w znacznej mniejszości, stanowi poniżej 5% zbioru. Taki problem nazywamy **klasyfikacją niezbalansowaną (imbalanced classification)**. Mamy tu **klasę dominującą (majority class)** oraz **klasę mniejszościową (minority class)**. Pechowo prawie zawsze interesuje nas ta druga, bo klasa większościowa jest trywialna. Przykładowo, 99% badanych jest zdrowych, a 1% ma niewykryty nowotwór - z oczywistych przyczyn chcemy wykrywać właśnie sytuację rzadką (problem diagnozy jako klasyfikacji jest zasadniczo zawsze niezbalansowany). W dalszej części laboratorium poznamy szereg konsekwencji tego zjawiska i metody na radzenie sobie z nim.

Mamy sporo cech, wszystkie numeryczne. Ciekawe, czy mają wartości brakujące, a jeśli tak, to ile. Można to policzyć, ale wykres jest często czytelniejszy. Pomoże nam tu biblioteka `missingno`. Zaznacza ona w każdej kolumnie wartości brakujące przeciwnym kolorem.

```
import missingno as msno

msno.matrix(X, labels=True, figsize=(30, 6))
```

```
<AxesSubplot:>
```



Jak widać, cecha 37 ma bardzo dużo wartości brakujących, podczas gdy pozostałe cechy mają raczej niewielką ich liczbę. W takiej sytuacji najlepiej usunąć tę cechę, a pozostałe wartości brakujące **uzupełnić / imputować (impute)**. Typowo wykorzystuje się do tego wartość średnią lub medianę z danej kolumny. Ale uwaga - imputacji dokonuje się dopiero po podziale na zbiór treningowy i testowy! W przeciwnym wypadku wykorzystywalibyśmy dane ze zbioru testowego, co sztucznie zawyżyłoby wyniki. Jest to błąd metodologiczny - **wyciek danych (data leakage)**.

Podział na zbiór treningowy i testowy to pierwszy moment, kiedy niezbalansowanie danych nam przeszkadza. Jeżeli zrobimy to czysto losowo, to są spore szanse, że w zbiorze testowym będzie tylko klasa negatywna - w końcu jest jej aż >95%. Dlatego wykorzystuje się **próbkowanie ze stratyfikacją (stratified sampling)**, dzięki któremu proporcje klas w zbiorze przed podziałem oraz obu zbiorach po podziale są takie same.

Zadanie 2 (0.75 punktu)

1. Usuń kolumnę `"Attr37"` ze zbioru danych.
2. Dokonaj podziału zbioru na treningowy i testowy w proporcjach 80%-20%, z przemieszaniem (`shuffle`), ze stratyfikacją, wykorzystując funkcję `train_test_split` ze Scikit-learn'a.
3. Uzupełnij wartości brakujące średnią wartością cechy z pomocą klasy `SimpleImputer`.

Uwaga:

- pamiętaj o uwzględnieniu stałego `random_state=0`, aby wyniki były **reprodukowalne (reproducible)**
- `stratify` oczekuje wektora klas
- wartości do imputacji trzeba wyestymować na zbiorze treningowym (`.fit()`), a potem zastosować te nauczone wartości na obu podzbiorach (treningowym i testowym)

```
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer


X.pop("Attr37")
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2,stratify=y, random_state=0,shuffle=True)
imputer = SimpleImputer()
imputer.fit(X_train)
X_train = imputer.transform(X_train)
X_test = imputer.transform(X_test)
```

Prosta klasyfikacja

Zanim przejdzie się do modeli bardziej złożonych, trzeba najpierw wypróbować coś prostego, żeby mieć punkt odniesienia. Tworzy się dlatego **modele bazowe (baselines)**.

W naszym przypadku będzie to **drzewo decyzyjne (decision tree)**. Jest to drzewo binarne z decyzjami if-else, prowadzącymi do klasyfikacji danego przykładu w liściu. Każdy podział w drzewie to pytanie postaci "Czy wartość cechy X jest większa lub równa Y?". Trening takiego drzewa to prosty algorytm zachłanny, bardzo przypomina budowę zwykłego drzewa binarnego. W każdym węźle wykonujemy:

1. Sprawdź po kolei wszystkie możliwe punkty podziału, czyli każdą (unikalną) wartość każdej cechy, po kolei.
2. Dla każdego przypadku podziel zbiór na 2 kawałki: niespełniający warunku (lewe dziecko) i spełniający warunek (prawe dziecko).
3. Oblicz jakość podziału według pewnej wybranej funkcji jakości. Im lepiej nasz if/else rozdziela klasy od siebie (im "czystsze" są węzły-dzieci), tym wyższa jakość. Innymi słowy, chcemy, żeby do jednego dziecka poszła jedna klasa, a do drugiego druga.
4. Wybierz podział o najwyższej jakości.

Taki algorytm wykonuje się rekurencyjnie, aż otrzymamy węzeł czysty (pure leaf), czyli taki, w którym są przykłady z tylko jednej klasy. Typowo wykorzystywaną funkcją jakości (kryterium podziału) jest entropia Shannona - im niższa entropia, tym bardziej jednolite są klasy w węźle (czyli wybieramy podział o najniższej entropii).

Powyższe wytłumaczenie algorytmu jest oczywiście nieformalne i dość skrótowe. Doskonałe tłumaczenie, z interaktywnymi wizualizacjami, dostępne jest tutaj. W formie filmów - tutaj oraz tutaj. Dla drzew do regresji - ten film.

Warto zauważyć, że taka konstrukcja prowadzi zawsze do overfittingu. Otrzymanie liści czystych oznacza, że mamy 100% dokładności na zbiorze treningowym, czyli perfekcyjnie przeuczony klasyfikator. W związku z tym nasze predykcje mają bardzo niski bias, ale bardzo dużą wariancję. Pomimo tego drzewa potrafią dać bardzo przyzwoite wyniki, a w celu ich poprawy można je regularyzować, aby mieć mniej "rozrośnięte" drzewo. Film dla zainteresowanych.

W tym wypadku AI to naprawdę tylko zbiór if'ów ;)

Mając wytrenowany klasyfikator, trzeba oczywiście sprawdzić, jak dobrze on sobie radzi. Tu natrafiamy na kolejny problem z klasyfikacją niezbalansowaną - zwykła celność (accuracy) na pewno nie zadziała! Typowo wykorzystuje się AUC, nazywane też AUROC (Area Under Receiver Operating Characteristic), bo metryka ta "widzi" i uwzględnia niezbalansowanie klas. Wymaga ona przekazania prawdopodobieństwa klasy pozytywnej, a nie tylko binarnej decyzji.

Bardzo dobre i bardziej szczegółowe wytłumaczenie, z interktywnymi wizualizacjami, można znaleć tutaj. Dla preferujących filmy - tutaj.

Co ważne, z definicji AUROC, trzeba tam użyć prawdopodobieństw klasy pozytywnej (klasy 1). W Scikit-learn'ie zwraca je metoda `.predict_proba()`, która w kolejnych kolumnach zwraca prawdopodobieństwa poszczególnych klas.

Zadanie 3 (0.75 punktu)

1. Wytrenuj klasyfikator drzewa decyzyjnego (klasa `DecisionTreeClassifier`). Użyj entropii jako kryterium podziału.
2. Oblicz i wypisz AUROC na zbiorze testowym dla drzewa decyzyjnego (funkcja `roc_auc_score`).
3. Skomentuj wynik - czy twoim zdaniem osiągnięty AUROC to dużo czy mało, biorąc pod uwagę możliwy zakres wartości tej metryki?

Uwaga:

- pamiętaj o użyciu stałego `random_state=0`

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import roc_auc_score

model = DecisionTreeClassifier(criterion="entropy",random_state=0)
model.fit(X_train,y_train)
y_pred = model.predict_proba(X_test)

score=roc_auc_score(y_test,y_pred[:,1])

print("AUROC:",score)
```

```
AUROC: 0.7266899766899767
```

Osiągnięty wynik to 0.726 jest mniej wiecej w polowie pomiedzy losowym klasyfikatorem (0.5) a idealnym (1). Jest to wynik w miare zadowalający, ale można go jeszcze poprawić.

Uczenie zespołowe, bagging, lasy losowe

Bardzo często wiele klasyfikatorów działających razem daje lepsze wyniki niż pojedynczy klasyfikator. Takie podejście nazywa się **uczeniem zespołowym (ensemble learning)**. Istnieje wiele różnych podejść do tworzenia takich klasyfikatorów złożonych (ensemble classifiers).

Podstawową metodą jest **bagging**:

1. Wylosuj N (np. 100, 500, ...) próbek boostrapowych (boostrap sample) ze zbioru treningowego. Próbka boostrapowa to po prostu losowanie ze zwracaniem, gdzie dla wejściowego zbioru z M wierszami losujemy M próbek. Będą tam powtórzenia, średnio nawet 1/3, ale się tym nie przejmujemy.
2. Wytrenuj klasyfikator bazowy (base classifier) na każdej z próbek boostrapowych.
3. Stwórz klasyfikator złożony poprzez uśrednienie predykcji każdego z klasyfikatorów bazowych.

Typowo klasyfikatory bazowe są bardzo proste, żeby można było szybko wytrenować ich dużą liczbę. Prawie zawsze używa się do tego drzew decyzyjnych. Dla klasyfikacji uśrednienie wyników polega na głosowaniu - dla nowej próbki każdy klasyfikator bazowy ją klasyfikuje, sumuje się głosy na każdą klasę i zwraca najbardziej popularną decyzję.

Taki sposób ensemblingu zmniejsza wariancję klasyfikatora. Intuicyjnie, skoro coś uśredniamy, to siłą rzeczy będzie mniej rozrzucone, bo dużo ciężej będzie osiągnąć jakąś skrajność. Redukuje to też overfitting.

Lasy losowe (Random Forests) to ulepszenie baggingu. Zaobserwowano, że pomimo losowania próbek boostrapowych, w baggingu poszczególne drzewa są do siebie bardzo podobne (są skorelowane), używają podobnych cech ze zbioru. My natomiast chcemy zróżnicowania, żeby mieć niski bias - redukcją wariancji zajmuje się uśrednianie. Dlatego używa się metody losowej podprzestrzeni (random subspace method) - przy każdym podziale drzewa losuje się tylko pewien podzbiór cech, których możemy użyć do tego podziału. Typowo jest to pierwiastek kwadratowy z ogólnej liczby cech.

Zarówno bagging, jak i lasy losowe mają dodatkowo bardzo przyjemną własność - są mało czułe na hiperparametry, szczególnie na liczbę drzew. W praktyce wystarczy ustawić 500 czy 1000 drzew i będzie dobrze działać. Dalsze dostrajanie hiperparametrów może jeszcze trochę poprawić wyniki, ale nie tak bardzo, jak przy innych klasyfikatorach. Jest to zatem doskonały wybór domyślny, kiedy nie wiemy, jakiego klasyfikatora użyć.

Dodatkowo jest to problem **embarassingly parallel** - drzewa można trenować w 100% równolegle, dzięki czemu jest to dodatkowo wydajna obliczeniowo metoda.

Głębsze wytłumaczenie, z interaktywnymi wizualizacjami, można znaleźć tutaj. Dobrze tłumaczy je też ta seria filmów.

Zadanie 4 (0.5 punktu)

1. Wytrenuj klasyfikator Random Forest (klasa `RandomForestClassifier`). Użyj 500 drzew i entropii jako kryterium podziału.
2. Sprawdź AUROC na zbiorze testowym.
3. Skomentuj wynik w odniesieniu do drzewa decyzyjnego.

Uwaga: pamiętaj o ustawieniu `random_state=0`. Dla przyspieszenia ustaw `n_jobs=-1` (użyje tylu procesów, ile masz dostępnych rdzeni procesora).

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import roc_auc_score

model =
RandomForestClassifier(criterion="entropy",random_state=0,n_jobs=-
1,n_estimators=500)
model.fit(X_train,y_train)
y_pred = model.predict_proba(X_test)

score=roc_auc_score(y_test,y_pred[:,1])

print("AUROC:",score)
```

```
AUROC: 0.8994111948657404
```

Wynik jest dużo lepszy niz w przypadku drzxewa decyzyjnego. AUROC wynosi 0.899, co jest bardzo dobrym rezultatem. Jak widać zastosowanie lasu losowego zamiast pojedynczego drzewa decyzyjnego potrafi znacznie poprawić wartosc wskaznika AUROC.

Jak zobaczymy poniżej, wynik ten możemy jednak jeszcze ulepszyć!

Oversampling, SMOTE

W przypadku zbiorów niezbalansowanych można dokonać **balansowania (balancing)** zbioru. Są tutaj 2 metody:

- **undersampling**: usunięcie przykładów z klasy dominującej
- **oversampling**: wygenerowanie dodatkowych przykładów z klasy mniejszościowej

Undersampling działa dobrze, kiedy niezbalansowanie jest niewielkie, a zbiór jest duży (możemy sobie pozwolić na usunięcie jego części). Oversampling typowo daje lepsze wyniki, istnieją dla niego bardzo efektywne algorytmy. W przypadku bardzo dużego niezbalansowania można zrobić oba.

Typowym algorytmem oversamplingu jest **SMOTE (Synthetic Minority Oversampling TEchnique)**. Działa on następująco:

1. Idź po kolei po przykładach z klasy mniejszościowej
2. Znajdź `k` najbliższych przykładów dla próbki, typowo `k=5`
3. Wylosuj tylu sąsiadów, ile trzeba do oversamplingu, np. jeżeli chcemy zwiększyć klasę mniejszościową 3 razy (o 200%), to wylosuj 2 z 5 sąsiadów
4. Dla każdego z wylosowanych sąsiadów wylosuj punkt na linii prostej między próbką a tym sąsiadem. Dodaj ten punkt jako nową próbkę do zbioru

Taka technika generuje przykłady bardzo podobne do prawdziwych, więc nie zaburza zbioru, a jednocześnie pomaga klasyfikatorom, bo "zagęszcza" przestrzeń, w której znajduje się klasa pozytywna.

Algorytm SMOTE, jego warianty i inne algorytmy dla problemów niezbalansowanych implementuje biblioteka Imbalanced-learn.

Zadanie 5 (1 punkt)

Użyj SMOTE do zbalansowania zbioru treningowego (nie używa się go na zbiorze testowym!) (klasa `SMOTE`). Wytrenuj drzewo decyzyjne oraz las losowy na zbalansowanym zbiorze, użyj tych samych argumentów co wcześniej. Pamiętaj o użyciu wszędzie stałego `random_state=0` i `n_jobs=-1`. Skomentuj wynik.

```
# your_code

from imblearn.over_sampling import SMOTE
```

```
smote = SMOTE(random_state=0,n_jobs=-1)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

decisionTree = DecisionTreeClassifier(criterion="entropy",
random_state=0)
randomForest=
RandomForestClassifier(criterion="entropy",random_state=0,n_jobs=-
1,n_estimators=500)

decisionTree.fit(X_train_smote,y_train_smote)
randomForest.fit(X_train_smote,y_train_smote)

y_pred = decisionTree.predict_proba(X_test)
y_pred_forest= randomForest.predict_proba(X_test)

score_tree=roc_auc_score(y_test,y_pred[:,1])
score_forest=roc_auc_score(y_test,y_pred_forest[:,1])

print("Decision tree AUROC:",score)
print("Random forest AUROC:",score_forest)
```

```
C:\Users\Szczepan\anaconda3\envs\PSI\lib\site-packages\imblearn\
over_sampling\_smote\base.py:345: FutureWarning: The parameter
`n_jobs` has been deprecated in 0.10 and will be removed in 0.12. You
can pass an nearest neighbors estimator where `n_jobs` is already set
instead.
  warnings.warn(

Decision tree AUROC: 0.8994111948657404
Random forest AUROC: 0.9047644274917003
```

Po użyciu smote widac znaczną poprawe w przypadku drzewa decyzyjnego. Natomiast random forest ma praktycznie taki sam wynik. Można wywnioskować z tego że las losowy jest bardziej odporny na niezbalansowanie klas niż pojedyncze drzewo decyzyjne.

W dalszej części laboratorium używaj zbioru po zastosowaniu SMOTE do treningu klasyfikatorów.

Dostrajanie (tuning) hiperparametrów

Lasy losowe są stosunkowo mało czułe na dobór hiperparametrów - i dobrze, bo mają ich dość dużo. Można zawsze jednak spróbować to zrobić, a w szczególności najważniejszy jest parametr

`max_features`, oznaczający, ile cech losować przy każdym podziale drzewa. Typowo sprawdza się wartości z zakresu `[0.1, 0.5]`.

W kwestii szybkości, kiedy dostrajamy hiperparametry, to mniej oczywiste jest, jakiego `n_jobs` użyć. Z jednej strony klasyfikator może być trenowany na wielu procesach, a z drugiej można trenować wiele klasyfikatorów na różnych zestawach hiperparametrów równolegle. Jeżeli nasz klasyfikator bardzo dobrze się uwspółbieżnia (jak Random Forest), to można dać mu nawet wszystkie rdzenie, a za to wypróbowywać kolejne zestawy hiperparametrów sekwencyjnie. Warto ustawić parametr `verbose` na 2 lub więcej, żeby dostać logi podczas długiego treningu i mierzyć czas wykonania. W praktyce ustawia się to metodą prób i błędów.

Zadanie 6 (1 punkt)

1. Dobierz wartość hiperparametru `max_features`:
   - użyj grid search z 5 foldami
   - wypróbuj wartości `[0.1, 0.2, 0.3, 0.4, 0.5]`
   - wybierz model o najwyzszym AUROC (argument `scoring`)
2. Sprawdź, jaka była optymalna wartość `max_features`. Jest to atrybut wytrenowanego `GridSearchCV`.
3. Skomentuj wynik. Czy warto było poświęcić czas i zasoby na tę procedurę?

Uwaga:

- pamiętaj, żeby jako estymatora przekazanego do grid search'a użyć instancji Random Forest, która ma już ustawione `random_state=0` i `n_jobs`

```
from sklearn.model_selection import GridSearchCV

randomForest_model=RandomForestClassifier(criterion="entropy",random_s
tate=0,n_jobs=-1,n_estimators=500,verbose=2)
param_grid = {'max_features': [0.1, 0.2, 0.3, 0.4, 0.5]}
grid_search = GridSearchCV(randomForest_model, param_grid,
cv=5,scoring="roc_auc")
grid_search.fit(X_train_smote, y_train_smote)

print(grid_search.best_params_)
print(grid_search.best_score_)

#wyszło 0.2

#test
#dlatego w ten sposob aby nie włączac grid searcha ponownie
randomForest_model_02=RandomForestClassifier(criterion="entropy",rando
m_state=0,n_jobs=-1,n_estimators=500,max_features=0.2)
randomForest_model_default_max_features=
RandomForestClassifier(criterion="entropy",random_state=0,n_jobs=-
1,n_estimators=500)
randomForest_model_02.fit(X_train_smote,y_train_smote)
randomForest_model_default_max_features.fit(X_train_smote,y_train_smot
```

```
e)

y_pred_02 = randomForest_model_02.predict_proba(X_test)
y_pred_default =
randomForest_model_default_max_features.predict_proba(X_test)

score_02=roc_auc_score(y_test,y_pred_02[:,1])
score_default=roc_auc_score(y_test,y_pred_default[:,1])

print("Random forest AUROC:",score_02)
print("Random forest AUROC:",score_default)



Random forest AUROC: 0.9122619804437986
Random forest AUROC: 0.9047644274917003
```

Grid search zwrócił 0.2 jako najlepszą wartość max_features.Widać że wynik jest lepszy niż dla domyślnej wartości. Warto było poświęcić czas na tę procedurę.

W praktycznych zastosowaniach data scientist wedle własnego uznana, doświadczenia, dostępnego czasu i zasobów wybiera, czy dostrajać hiperparametry i w jak szerokim zakresie. Dla Random Forest na szczęście często może nie być znaczącej potrzeby, i za to go lubimy :)

Random Forest - podsumowanie

1. Model oparty o uczenie zespołowe
2. Kluczowe elementy:
   - bagging: uczenie wielu klasyfikatorów na próbkach boostrapowych
   - metoda losowej podprzestrzeni: losujemy podzbiór cech do każdego podziału drzewa
   - uśredniamy głosy klasyfikatorów
3. Dość odporny na overfitting, zmniejsza wariancję błędu dzięki uśrednianiu
4. Mało czuły na hiperparametry
5. Przeciętnie bardzo dobre wyniki, doskonały wybór domyślny przy wybieraniu algorytmu klasyfikacji

Boosting

Drugą bardzo ważną grupą algorytmów ensemblingu jest **boosting**, też oparty o drzewa decyzyjne. O ile Random Forest trenował wszystkie klasyfikatory bazowe równolegle i je uśredniał, o tyle boosting robi to sekwencyjnie. Drzewa te uczą się na całym zbiorze, nie na próbkach boostrapowych. Idea jest następująca: trenujemy drzewo decyzyjne, radzi sobie przeciętnie i popełnia błędy na części przykładów treningowych. Dokładamy kolejne, ale znające błędy swojego poprzednika, dzięki czemu może to uwzględnić i je poprawić. W związku z tym "boostuje" się dzięki wiedzy od poprzednika. Dokładamy kolejne drzewa zgodnie z tą samą zasadą.

Jak uczyć się na błędach poprzednika? Jest to pewna **funkcja kosztu** (błędu), którą chcemy zminimalizować. Zakłada się jakąś jej konkretną postać, np. squared error dla regresji, albo logistic loss dla klasyfikacji. Później wykorzystuje się spadek wzdłuż gradientu (gradient descent), aby nauczyć się, w jakim kierunku powinny optymalizować kolejne drzewa, żeby zminimalizować błędy poprzednika. Jest to konkretnie **gradient boosting**, absolutnie najpopularniejsza forma boostingu, i jeden z najpopularniejszych i osiągających najlepsze wyniki algorytmów ML.

Tyle co do intuicji. Ogólny algorytm gradient boostingu jest trochę bardziej skomplikowany. Bardzo dobrze i krok po kroku tłumaczy go ta seria filmów na YT. Szczególnie ważne implementacje gradient boostingu to **XGBoost (Extreme Gradient Boosting)** oraz **LightGBM (Light Gradient Boosting Machine)**. XGBoost był prawdziwym przełomem w ML, uzyskując doskonałe wyniki i bardzo dobrze się skalując - był wykorzystany w CERNie do wykrywania cząstki Higgsa w zbiorze z pomiarów LHC mającym 10 milionów próbek. Jego implementacja jest dość złożona, ale dobrze tłumaczy ją inna seria filmików na YT.

Obecnie najczęściej wykorzystuje się LightGBM. Został stworzony przez Microsoft na podstawie doświadczeń z XGBoostem. Został jeszcze bardziej ulepszony i przyspieszony, ale różnice są głównie implementacyjne. Różnice dobrze tłumaczy ta prezentacja z konferencji PyData oraz prezentacja Microsoftu. Dla zainteresowanych - praktyczne aspekty LightGBM.

Zadanie 7 (0.5 punktu)

1. Wytrenuj klasyfikator LightGBM (klasa `LGBMClassifier`). Przekaż `importance_type="gain"` - przyda nam się to za chwilę.
2. Sprawdź AUROC na zbiorze testowym.
3. Skomentuj wynik w odniesieniu do wcześniejszych algorytmów.

Pamiętaj o `random_state` i `n_jobs`.

```
# your_code
from lightgbm import LGBMClassifier


lgbm = LGBMClassifier(importance_type="gain",random_state=0,n_jobs=-1)
lgbm.fit(X_train_smote,y_train_smote)
y_pred = lgbm.predict_proba(X_test)

score=roc_auc_score(y_test,y_pred[:,1])

print("AUROC:",score)


AUROC: 0.9433748070111706
```

Wynik AUROC po zastosowaniu boostingu jest jeszcze lepszy niż poprzedni i jest naprawde bliski idealnego wyniku. Dodatkowo cała procedura jest dużo szybsza.

Boosting dzięki uczeniu na poprzednich drzewach redukuje nie tylko wariancję, ale też bias w błędzie, dzięki czemu może w wielu przypadkach osiągnąć lepsze rezultaty od lasu losowego. Do tego dzięki znakomitej implementacji LightGBM jest szybszy.

Boosting jest jednak o wiele bardziej czuły na hiperparametry niż Random Forest. W szczególności bardzo łatwo go przeuczyć, a większość hiperparametrów, których jest dużo, wiąże się z regularyzacją modelu. To, że teraz poszło nam lepiej z domyślnymi, jest rzadkim przypadkiem.

W związku z tym, że przestrzeń hiperparametrów jest duża, przeszukanie wszystkich kombinacji nie wchodzi w grę. Zamiast tego można wylosować zadaną liczbę zestawów hiperparametrów i tylko je sprawdzić - chociaż im więcej, tym lepsze wyniki powinniśmy dostać. Służy do tego `RandomizedSearchCV`. Co więcej, klasa ta potrafi próbkować rozkłady prawdopodobieństwa, a nie tylko sztywne listy wartości, co jest bardzo przydatne przy parametrach ciągłych.

Hiperparametry LightGBMa są dobrze opisane w oficjalnej dokumentacji: wersja krótsza i wersja dłuższa. Jest ich dużo, więc nie będziemy ich tutaj omawiać. Jeżeli chodzi o ich dostrajanie w praktyce, to przydatny jest oficjalny guide oraz dyskusje na Kaggle.

Zadanie 8 (1.5 punktu)

1. Zaimplementuj random search dla LightGBMa (klasa `RandomizedSearchCV`):
   - użyj tylu prób, na ile pozwalają twoje zasoby obliczeniowe, ale przynajmniej 30
   - przeszukaj przestrzeń hiperparametrów:

```
param_grid = {
    "n_estimators": [400, 500, 600],
    "learning_rate": [0.05, 0.1, 0.2],
    "num_leaves": [31, 48, 64],
    "colsample_bytree": [0.8, 0.9, 1.0],
    "subsample": [0.8, 0.9, 1.0],
}
```

2. Wypisz znalezione optymalne hiperparametry.
3. Wypisz raporty z klasyfikacji (funkcja `classification_report`), dla modelu LightGBM bez i z dostrajaniem hiperparametrów.
4. Skomentuj różnicę precyzji (precision) i czułości (recall) między modelami bez i z dostrajaniem hiperparametrów. Czy jest to pożądane zjawisko w tym przypadku?

Uwaga: pamiętaj o ustawieniu `importance_type`, `random_state=0` i `n_jobs`, oraz ewentualnie `verbose` dla śledzenia przebiegu

```
# your_code

from sklearn.model_selection import RandomizedSearchCV


param_grid = {
    "n_estimators": [400, 500, 600],
    "learning_rate": [0.05, 0.1, 0.2],
```

```
    "num_leaves": [31, 48, 64],
    "colsample_bytree": [0.8, 0.9, 1.0],
    "subsample": [0.8, 0.9, 1.0],
}

lgbm_clf =
LGBMClassifier(importance_type="gain",random_state=0,n_jobs=-1)
random_search =
RandomizedSearchCV(lgbm_clf,param_grid,random_state=0,n_jobs=-
1,n_iter=30)

random_search.fit(X_train_smote,y_train_smote)

lgbm_clf_tuned= random_search.best_estimator_
y_pred_tuned = lgbm_clf_tuned.predict_proba(X_test)

lgbm_clf.fit(X_train_smote,y_train_smote)
y_pred = lgbm_clf.predict_proba(X_test)

score_tuned=roc_auc_score(y_test,y_pred_tuned[:,1])
score=roc_auc_score(y_test,y_pred[:,1])

print("AUROC tuned:",score_tuned)

print("AUROC:",score)

print("Tuned params:",random_search.best_params_)
```

AUROC tuned: 0.9464676737404011 AUROC: 0.9433748070111706 Tuned params: {'subsample': 0.8, 'num_leaves': 31, 'n_estimators': 400, 'learning_rate': 0.2, 'colsample_bytree': 0.9}

```
from sklearn.metrics import classification_report

print("Without tuning")
print(classification_report(y_test, lgbm_clf.predict(X_test)))

print("With tuning")
print(classification_report(y_test, lgbm_clf_tuned.predict(X_test)))
```

Wyniki są bardzo podobne. W przypadku modelu z dostrajaniem hiperparametrów mamy lepszy wynik AUROC.

Model bez strojenia parametrów osiągnął następujące wyniki:

Precyzja wynosi 0.60, co oznacza, że zbankrutowane firmy zostały zidentyfikowane z dokładnością 60%. Czułość wynosi również 0.60, co oznacza, że model poprawnie wykrył 60% rzeczywistych zbankrutowanych firm. Ogólna dokładność (accuracy) wynosi 0.96, co oznacza, że model poprawnie sklasyfikował 96% obserwacji.

Model z dostrajaniem hiperparametrów (tuning) : Precyzja wzrosła do 0.76, co oznacza poprawę identyfikacji zbankrutowanych firm. Czułość wynosi teraz 0.52, co oznacza, że model identyfikuje mniej rzeczywistych zbankrutowanych firm w porównaniu do modelu bez strojenia. Mimo że model z dostrajaniem hiperparametrów ma wyższą precyzję w identyfikacji zbankrutowanych firm, to ma niższą czułość. Ostateczny wybór między tymi dwoma modelami zależy od konkretnego celu biznesowego. Jeśli ważniejsza jest identyfikacja zbankrutowanych firm, to model z dostrajaniem hiperparametrów może być bardziej odpowiedni. Jednak, jeśli istotniejsze jest wykrycie większej liczby rzeczywistych zbankrutowanych firm, to model bez strojenia hiperparametrów może być lepszym wyborem.

Without tuning precision recall f1-score support

```
       0       0.98      0.98      0.98      2002
       1       0.60      0.60      0.60        99

accuracy                           0.96      2101
```

macro avg 0.79 0.79 0.79 2101 weighted avg 0.96 0.96 0.96 2101

With tuning precision recall f1-score support

```
       0       0.98      0.99      0.98      2002
       1       0.76      0.52      0.61        99

accuracy                           0.97      2101
```

macro avg 0.87 0.75 0.80 2101 weighted avg 0.97 0.97 0.97 2101

Boosting - podsumowanie

1. Model oparty o uczenie zespołowe
2. Kolejne modele są dodawane sekwencyjnie i uczą się na błędach poprzedników
3. Nauka typowo jest oparta o minimalizację funkcji kosztu (błędu), z użyciem spadku wzdłuż gradientu
4. Wiodący model klasyfikacji dla danych tabelarycznych, z 2 głównymi implementacjami: XGBoost i LightGBM
5. Liczne hiperparametry, wymagające odpowiednich metod dostrajania

Wyjaśnialna AI

W ostatnich latach zaczęto zwracać coraz większą uwagę na wpływ sztucznej inteligencji na społeczeństwo, a na niektórych czołowych konferencjach ML nawet obowiązkowa jest sekcja "Social impact" w artykułach naukowych. Typowo im lepszy model, tym bardziej złożony, a najpopularniejsze modele boostingu są z natury skomplikowane. Kiedy mają podejmować krytyczne decyzje, to musimy wiedzieć, czemu predykcja jest taka, a nie inna. Jest to poddziedzina uczenia maszynowego - **wyjaśnialna AI (explainable AI, XAI)**.

Taka informacja jest cenna, bo dzięki temu lepiej wiemy, co robi model. Jest to ważne z kilku powodów:

1. Wymogi prawne - wdrażanie algorytmów w ekonomii, prawie etc. ma coraz częściej konkretne wymagania prawne co do wyjaśnialności predykcji
2. Dodatkowa wiedza dla użytkowników - często dodatkowe obserwacje co do próbek są ciekawe same w sobie i dają wiedzę użytkownikowi (często posiadającemu specjalistyczną wiedzę z dziedziny), czasem nawet bardziej niż sam model predykcyjny
3. Analiza modelu - dodatkowa wiedza o wewnętrznym działaniu algorytmu pozwala go lepiej zrozumieć i ulepszyć wyniki, np. przez lepszy preprocessing danych

W szczególności można ją podzielić na **globalną** oraz **lokalną interpretowalność (global / local interpretability)**. Ta pierwsza próbuje wyjaśnić, czemu ogólnie model działa tak, jak działa. Analizuje strukturę modelu oraz trendy w jego predykcjach, aby podsumować w prostszy sposób jego tok myślenia. Interpretowalność lokalna z kolei dotyczy predykcji dla konkretnych próbek - czemu dla danego przykładu model podejmuje dla niego taką, a nie inną decyzję o klasyfikacji.

W szczególności podstawowym sposobem interpretowalności jest **ważność cech (feature importance)**. Wyznacza ona, jak ważne są poszczególne cechy:

- w wariancie globalnym, jak mocno model opiera się na poszczególnych cechach
- w wariancie lokalnym, jak mocno konkretne wartości cech wpłynęły na predykcję, i w jaki sposób

Teraz będzie nas interesować globalna ważność cech. Dla modeli drzewiastych definiuje się ją bardzo prosto. Każdy podział w drzewie decyzyjnym wykorzystuje jakąś cechę, i redukuje z pomocą podziału funkcję kosztu (np. entropię) o określoną ilość. Dla drzewa decyzyjnego ważność to sumaryczna redukcja entropii, jaką udało się uzyskać za pomocą danej cechy. Dla lasów losowych i boostingu sumujemy te wartości dla wszystkich drzew. Alternatywnie można też użyć liczby splitów, w jakiej została użyta dana cecha, ale jest to mniej standardowe.

Warto zauważyć, że taka ważność cech jest **względna**:

- nie mówimy, jak bardzo ogólnie ważna jest jakaś cecha, tylko jak bardzo przydatna była dla naszego modelu w celu jego wytrenowania
- ważność cech można tylko porównywać ze sobą, np. jedna jest 2 razy ważniejsza od drugiej; nie ma ogólnych progów ważności

Ze względu na powyższe, ważności cech normalizuje się często do zakresu [0, 1] dla łatwiejszego porównywania.

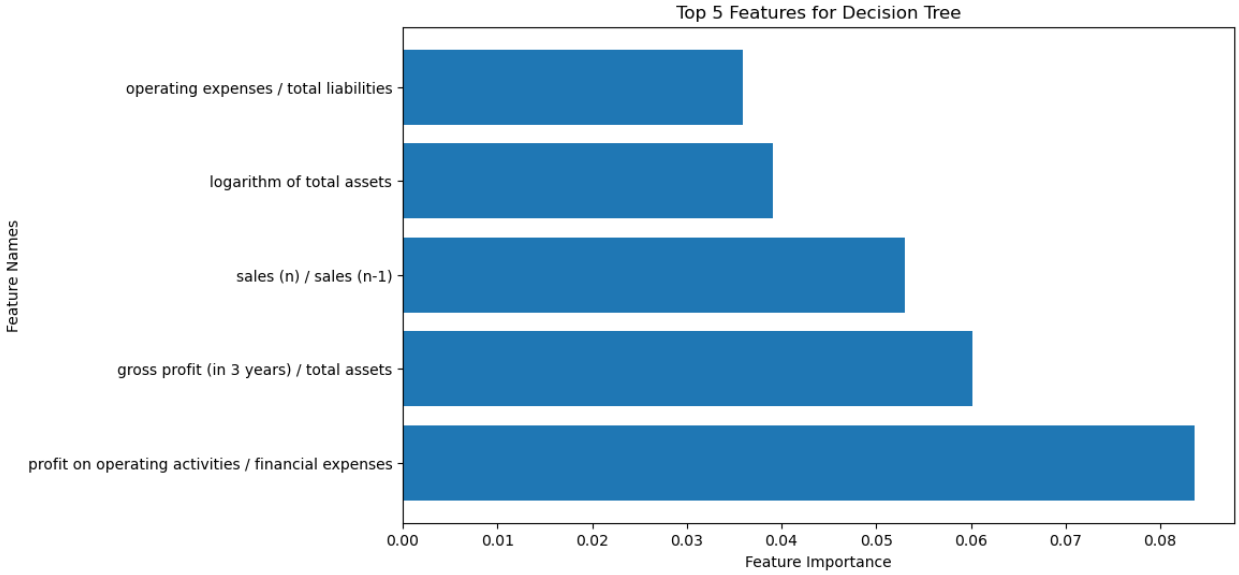
Zadanie 9 (0.5 punktu)

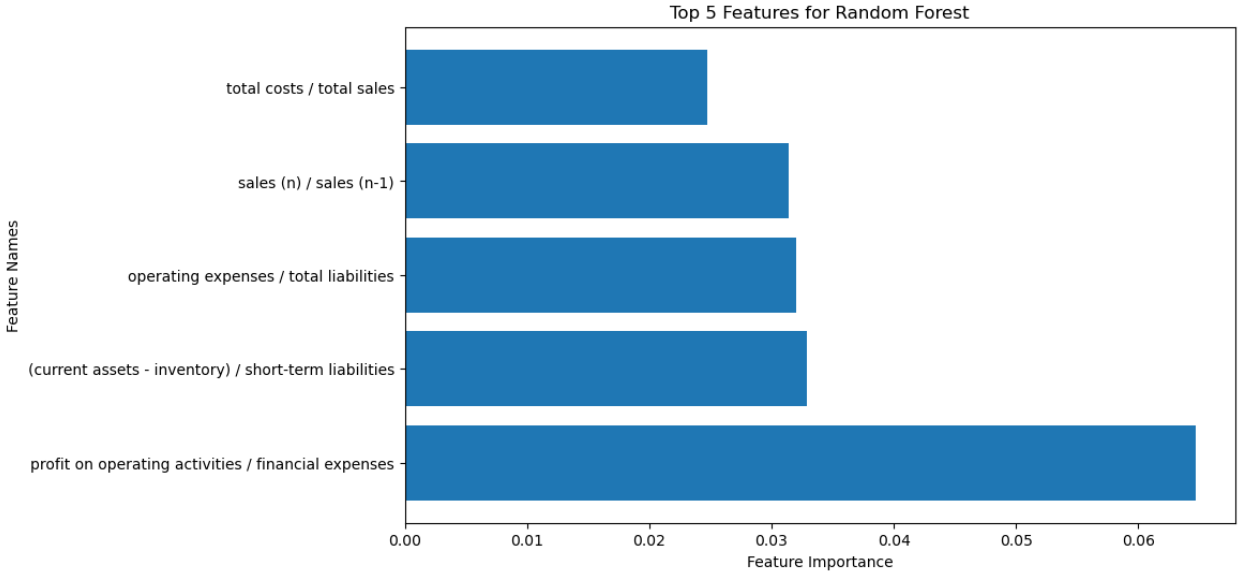
1. Wybierz 5 najważniejszych cech dla drzewa decyzyjnego. Przedstaw wyniki na poziomym wykresie słupkowym. Użyj czytelnych nazw cech ze zmiennej `feature_names`.
2. Powtórz powyższe dla lasu losowego, oraz dla boostingu (tutaj znormalizuj wyniki - patrz uwaga niżej). Wybierz te hiperparametry, które dały wcześniej najlepsze wyniki.
3. Skomentuj, czy wybrane cechy twoim zdaniem mają sens jako najważniejsze cechy.

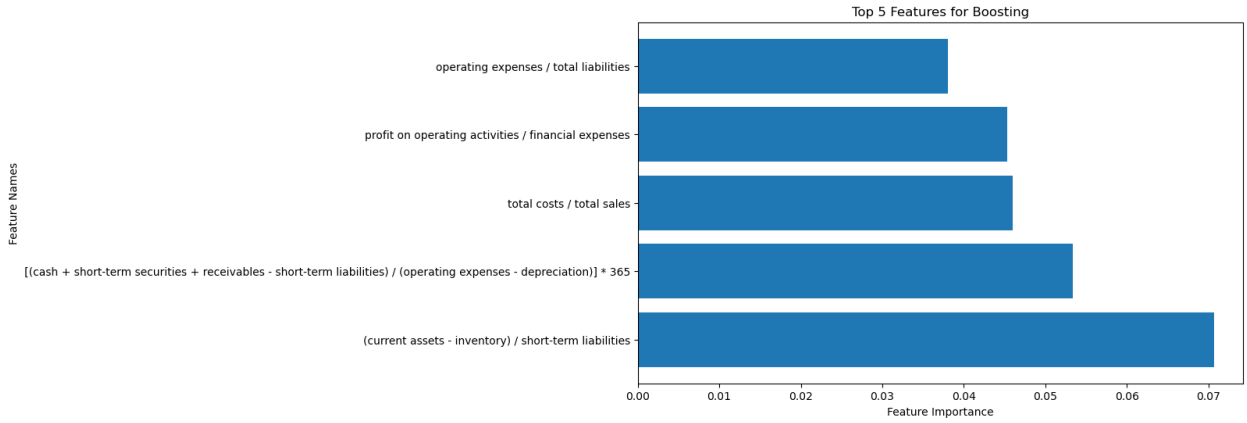
Uwaga: Scikit-learn normalizuje ważności do zakresu [0, 1], natomiast LightGBM nie. Musisz to znormalizować samodzielnie, dzieląc przez sumę.

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

```
def generate_plot(names,importance,model_name):
    plt.figure(figsize=(10, 6))
    plt.barh(names, importance)
    plt.title(f"Top 5 Features for {model_name}")
    plt.xlabel("Feature Importance")
    plt.ylabel("Feature Names")
    plt.show()


tree_model = DecisionTreeClassifier(random_state=0,max_features=0.2)
tree_model.fit(X_train, y_train)
tree_feature_importances = tree_model.feature_importances_

rf_model = RandomForestClassifier(random_state=0,)
rf_model.fit(X_train, y_train)
rf_feature_importances = rf_model.feature_importances_

#'subsample': 0.8, 'num_leaves': 31, 'n_estimators': 400,
'learning_rate': 0.2, 'colsample_bytree': 0.9
boosting_model = LGBMClassifier(n_estimators=50,
random_state=0,subsample=0.8,num_leaves=31,learning_rate=0.2,colsample
_bytree=0.9)
boosting_model.fit(X_train, y_train)
boosting_feature_importances = boosting_model.feature_importances_
boosting_feature_importances_normalized = boosting_feature_importances
/ np.sum(boosting_feature_importances)


t_sorted_idx = np.argsort(tree_feature_importances)[::-1]
top_features_tree = [feature_names[i] for i in t_sorted_idx[:5]]
top_importances_tree = tree_feature_importances[t_sorted_idx][:5]


generate_plot(top_features_tree,top_importances_tree,"Decision Tree")

rf_sorted_idx = np.argsort(rf_feature_importances)[::-1]
top_features_rf = [feature_names[i] for i in rf_sorted_idx[:5]]
top_importances_rf = rf_feature_importances[rf_sorted_idx][:5]


generate_plot(top_features_rf,top_importances_rf,"Random Forest")

b_sorted_idx = np.argsort(boosting_feature_importances_normalized)[::-
1]
top_features_boosting = [feature_names[i] for i in b_sorted_idx[:5]]
top_importances_boosting =
boosting_feature_importances_normalized[b_sorted_idx][:5]

generate_plot(top_features_boosting,top_importances_boosting,"Boosting
")
```


Top 5 Features for Decision Tree
operating expenses / total liabilities
logarithm of total assets
sales (n) / sales (n-1)
gross profit (in 3 years) / total assets
profit on operating activities / financial expenses
Feature Names
0.00
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
Feature Importance


Top 5 Features for Random Forest
total costs / total sales
sales (n) / sales (n-1)
operating expenses / total liabilities
(current assets - inventory) / short-term liabilities
profit on operating activities / financial expenses
Feature Names
0.00
0.01
0.02
0.03
0.04
0.05
0.06
Feature Importance



Wiekszosc cech się powtarza we wszystkich modelach. Najważniejsze cechy to:

- profit on operating activities / financial expenses
- current assets / short-term liabilities
- sales (n)/sales (n-1)
- total cost/total sales

Jak widać najważniejsze cechy to te które mówią o zyskach i kapitale firmy. Mają sens jako najważniejsze cechy.

Dla zainteresowanych

Najpopularniejszym podejściem do interpretowalności lokalnych jest **SHAP (SHapley Additive exPlanations)**, metoda oparta o kooperatywną teorię gier. Traktuje się cechy modelu jak zbiór graczy, podzielonych na dwie drużyny (koalicje): jedna chce zaklasyfikować próbkę jako negatywną, a druga jako pozytywną. O ostatecznej decyzji decyduje model, który wykorzystuje te wartości cech. Powstaje pytanie - w jakim stopniu wartości cech przyczyniły się do wyniku swojej drużyny? Można to obliczyć jako wartości Shapleya (Shapley values), które dla modeli ML oblicza algorytm SHAP. Ma on bardzo znaczące, udowodnione matematycznie zalety, a dodatkowo posiada wyjątkowo efektywną implementację dla modeli drzewiastych oraz dobre wizualizacje.

Bardzo intuicyjnie, na prostym przykładzie, SHAPa wyjaśnia pierwsza część tego artykułu. Dobrze i dość szczegółowo SHAPa wyjaśnia jego autor w tym filmie.

Wyjaśnialna AI - podsumowanie

1. Problem zrozumienia, jak wnioskuje model i czemu podejmuje dane decyzje
2. Ważne zarówno z perspektywy data scientist'a, jak i użytkowników systemu
3. Można wyjaśniać model lokalnie (konkretne predykcje) lub globalnie (wpływ poszczególnych cech)

Zadanie dla chętnych

Dokonaj selekcji cech, usuwając 20% najsłabszych cech. Może się tu przydać klasa `SelectPercentile`. Czy Random Forest i LightGBM (bez dostrajania hiperparametrów, dla uproszczenia) wytrenowane bez najsłabszych cech dają lepszy wynik (AUROC lub innej metryki)?

Wykorzystaj po 1 algorytmie z 3 grup algorytmów selekcji cech:

1. Filter methods - mierzymy ważność każdej cechy niezależnie, za pomocą pewnej miary (typowo ze statystyki lub teorii informacji), a potem odrzucamy (filtrujemy) te o najniższej ważności. Są to np. `chi2` i `mutual_info_classif` z pakietu `sklearn.feature_selection`.
2. Embedded methods - klasyfikator sam zwraca ważność cech, jest jego wbudowaną cechą (stąd nazwa). Jest to w szczególności właściwość wszystkich zespołowych klasyfikatorów drzewiastych. Mają po wytrenowaniu atrybut `feature_importances_`.
3. Wrapper methods - algorytmy wykorzystujące w środku używany model (stąd nazwa), mierzące ważność cech za pomocą ich wpływu na jakość klasyfikatora. Jest to np. recursive feature elimination (klasa `RFE`). W tym algorytmie trenujemy klasyfikator na wszystkich cechach, wyrzucamy najsłabszą, trenujemy znowu i tak dalej.

Typowo metody filter są najszybsze, ale dają najsłabszy wynik, natomiast metody wrapper są najwolniejsze i dają najlepszy wynik. Metody embedded są gdzieś pośrodku.

Dla zainteresowanych, inne znane i bardzo dobre algorytmy:

- Relief (filter method) oraz warianty, szczególnie ReliefF, SURF i MultiSURF (biblioteka `ReBATE`): Wikipedia, artykuł "Benchmarking Relief-Based Feature Selection Methods"
- Boruta (wrapper method), stworzony na Uniwersytecie Warszawskim, łączący Random Forest oraz testy statystyczne (biblioteka `boruta_py`): link 1, link 2