

Inżynieria Uczenia Maszynowego

Studenci:

Bartłomiej Krawczyk

Mateusz Brzozowski

Temat

“Jakiś czas temu wprowadziliśmy konta premium, które uwalniają użytkowników od słuchania reklam. Nie są one jednak jeszcze zbyt popularne – czy możemy się dowiedzieć, które osoby są bardziej skłonne do zakupu takiego konta?”

```
import itertools
import numpy as np
import pandas as pd
import pickle
import requests
import seaborn as sns

from IPython.display import display
from matplotlib import pyplot as plt
from math import sqrt
from scipy.stats import uniform
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression
from sklearn.dummy import DummyClassifier
from sklearn.model_selection import RandomizedSearchCV, train_test_split
from sklearn.metrics import (
    confusion_matrix,
    roc_auc_score
)
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from statistics import stdev, mean
from typing import Any, Dict, Optional
from xgboost import XGBClassifier

from microservice import IUMModel
from utility import Model
```

Cechy i funkcje celu

Do trenowania naszych modeli przygotowaliśmy następujące cechy wygenerowane na podstawie dostarczonych danych:

- `number_of_advertisements`, ilość odtworzonych reklam w danym miesiącu
- `number_of_tracks`, ilość przesłuchanych utworów w danym miesiącu
- `number_of_skips`, ilość pominiętych utworów w danym miesiącu
- `number_of_likes`, liczba danych lików w danym miesiącu
- `number_of_liked_tracks_listened`, liczba przesłuchanych utworów w danym miesiącu, które w momencie odtworzenia były polubione
- `number_of_tracks_in_favourite_genre`, liczba przesłuchanych utworów z ulubionego gatunku w danym miesiącu
- `total_number_of_favourite_genres_listened`, liczba przesłuchanych gatunków w danym miesiącu należących do ulubionych użytkownika
- `average_popularity_in_favourite_genres`, średnia popularność utworów wśród ulubionych gatunków w danym miesiącu

- `total_tracks_duration_ms`, całkowity czas przesłuchanych utworów w danym miesiącu
- `number_of_different_artists`, ilość przesłuchanych artystów w danym miesiącu
- `average_release_date`, średnia data przesłuchanych piosenek w danym miesiącu
- `average_duration_ms`, średni czas trwania utworów przesłuchanych w danym miesiącu
- `explicit_tracks_ratio`, ułamek "wulgarnych" utworów przesłuchanych w danym miesiącu
- `average_popularity`, średnia popularność przesłuchanych utworów w danym miesiącu
- `average_acousticness`, średnia akustyka przesłuchanych utworów w danym miesiącu
- `average_danceability`, średnia taneczność przesłuchanych utworów w danym miesiącu
- `average_energy`, średnia moc przesłuchanych utworów w danym miesiącu
- `average_instrumentalness`, średnia ilość utworów z wokalem przesłuchanych w danym miesiącu
- `average_liveness`, średnie brzmienie utworów na żywo przesłuchanych w danym miesiącu
- `average_loudness`, średnia głośność przesłuchanych utworów w danym miesiącu
- `average_speechiness`, średnia ilość wokalu w utworach przesłuchanych w danym miesiącu
- `average_tempo`, średnia prędkość przesłuchanych utworów w danym miesiącu
- `average_valence`, średnia emocjonalność przesłuchanych utworów w danym miesiącu
- `average_track_name_length`, średnia długość nazwy utworów przesłuchanych w danym miesiącu
- `average_daily_cost`, średni koszt utrzymania przesłuchanych piosenek w danym miesiącu

Posiadamy również dwie funkcje celu:

- `premium_user_numerical`, która określa czy użytkownik kiedykolwiek kupi premium
- `will_buy_premium_next_month_numerical` przedstawiająca to czy użytkownik zakupi premium w przeciągu następnych 30 dni

```
FEATURE_VERSION = 'v1'
FEATURE_PATH = f'features/{FEATURE_VERSION}/feature.csv'
```

```
FEATURES = [
    'number_of_advertisements',
    'number_of_tracks',
    'number_of_skips',
    'number_of_likes',
    'number_of_liked_tracks_listened',
    'number_of_tracks_in_favourite_genre',
    'total_number_of_favourite_genres_listened',
    'average_popularity_in_favourite_genres',
    'total_tracks_duration_ms',
    'number_of_different_artists',
    'average_release_date',
    'average_duration_ms',
    'explicit_tracks_ratio',
    'average_popularity',
    'average_acousticness',
    'average_danceability',
    'average_energy',
    'average_instrumentalness',
    'average_liveness',
    'average_loudness',
    'average_speechiness',
    'average_tempo',
    'average_valence',
    'average_track_name_length',
    'average_daily_cost',
]
```

```
TARGETS = [
    'premium_user_numerical',
    'will_buy_premium_next_month_numerical'
```

```
]
TARGET_AND_FEATURES = TARGETS + FEATURES

data_frame = pd.read_csv(FEATURE_PATH)
```

Przykładowe wartości cech oraz funkcji celu

```
data_frame.head()
```

```

    user_id  year  month  premium_user_numerical  \
0      212  2020     2                1
1      212  2020     7                1
2      212  2020     1                1
3      212  2020     3                1
4      212  2020     8                1

    will_buy_premium_next_month_numerical  number_of_premium  \
0                0                0
1                0                0
2                0                0
3                0                0
4                0                0

    number_of_advertisements  number_of_tracks  number_of_skips  \
0                10                30                16
1                 5                16                 8
2                14                39                21
3                 3                20                 9
4                13                40                22

    number_of_likes  ...  average_danceability  average_energy  \
0                 8  ...          0.542767          0.600467
1                 2  ...          0.499000          0.675250
2                12  ...          0.487000          0.691667
3                 6  ...          0.516700          0.637150
4                12  ...          0.513525          0.642350

    average_instrumentalness  average_liveness  average_loudness  \
0              0.094653          0.179113          -10.158167
1              0.150966          0.226225           -8.795062
2              0.119363          0.235690           -9.491795
3              0.069641          0.189545           -8.409150
4              0.112645          0.285203           -9.562400

    average_speechiness  average_tempo  average_valence  \
0              0.049070        121.964467          0.588167
1              0.061812        123.069063          0.452131
2              0.064918        123.745769          0.564369
3              0.053715        121.415600          0.586000
4              0.055282        120.922325          0.510517

    average_track_name_length  average_daily_cost
0              22.233333          0.012134
1              22.687500          0.014780
2              19.461538          0.012399
```

3	21.250000	0.012706
4	24.000000	0.013076

[5 rows x 31 columns]

Macierz korelacji cech z wartościami przewidywanymi

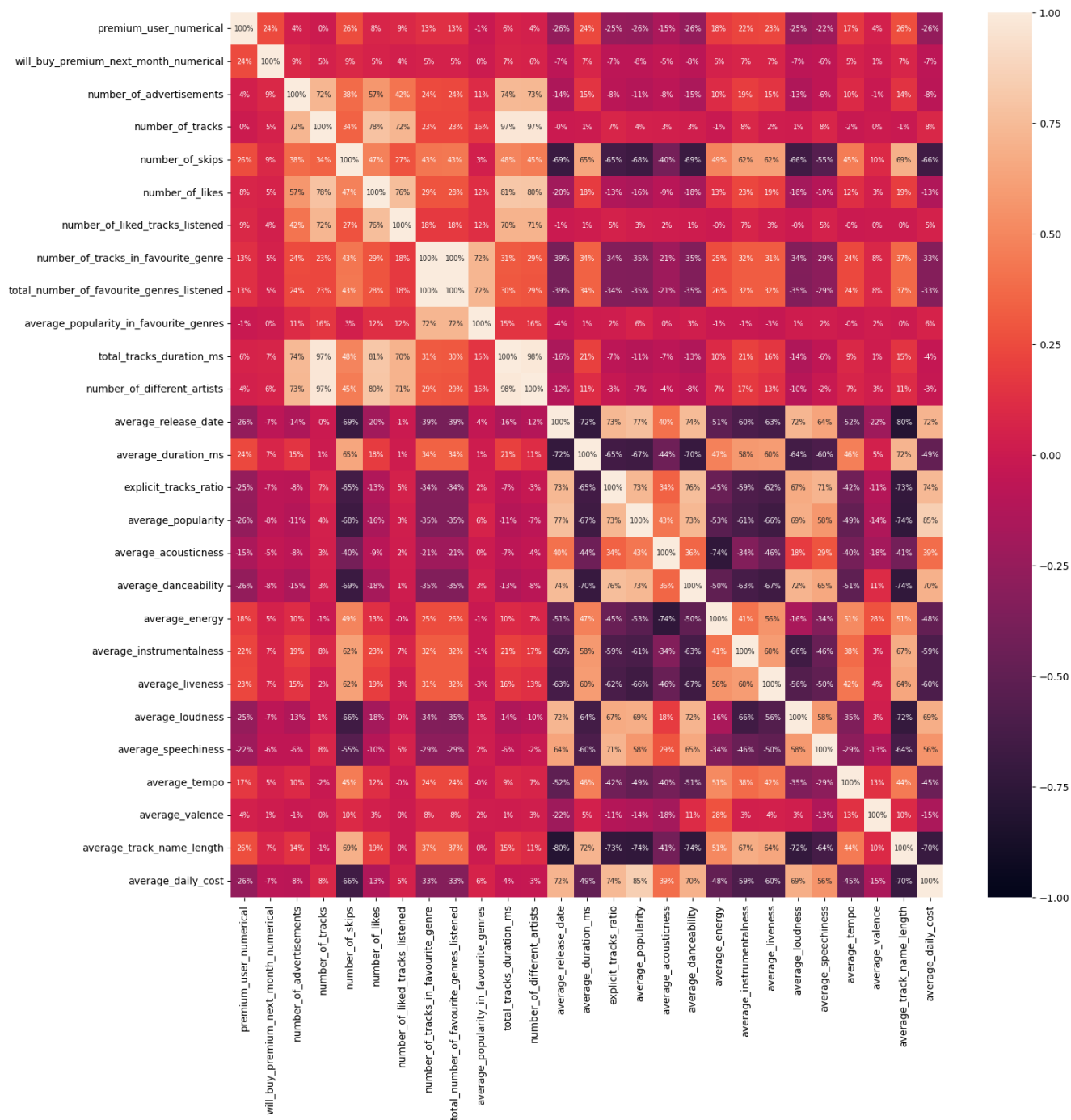
Sprawdzamy korelację cech, które nie są zbyt skorelowane między sobą, a za to są skorelowane z targetem.

```
correlation_matrix = data_frame.loc[:, TARGET_AND_FEATURES] \
    .corr(method='spearman')

plt.figure(figsize=(16, 16))

sns.heatmap(
    correlation_matrix,
    xticklabels=correlation_matrix.columns, # type: ignore
    yticklabels=correlation_matrix.columns, # type: ignore
    annot=True,
    annot_kws={"fontsize": 7},
    fmt=".0%",
    vmin=-1,
    vmax=1,
)

plt.show()
```



Definiujemy pipeline do uzupełnienia danych pustych oraz przeskalowania danych

```
pipeline = Pipeline([
    ("simple_imputer", SimpleImputer()),
    ("standard_scaler", StandardScaler())
])
```

Dzielimy dane na dane trenujące oraz testowe do późniejszych eksperymentów A/B

```
TRAINING_UP_TO = 2023
TRAIN_DATA: pd.DataFrame = data_frame.loc[data_frame.year < TRAINING_UP_TO, :]
TEST_DATA: pd.DataFrame = data_frame.loc[data_frame.year >= TRAINING_UP_TO, :]
TEST_SIZE = 0.33
```

Pipeline uczony jest na podstawie samych danych testowych

```
X_train_temp, X_test_temp, Y_train, Y_test = train_test_split(
    TRAIN_DATA[FEATURES],
    TRAIN_DATA[TARGETS],
    test_size=TEST_SIZE
)
X_train_temp: pd.DataFrame
X_test_temp: pd.DataFrame
Y_train: pd.DataFrame
Y_test: pd.DataFrame

train_data = pipeline.fit_transform(X_train_temp)
test_data = pipeline.transform(X_test_temp)
X_train = pd.DataFrame(train_data, columns=FEATURES)
X_test = pd.DataFrame(test_data, columns=FEATURES)
```

Cechy przetworzone przez pipeline

```
X_train.head()
```

	number_of_advertisements	number_of_tracks	number_of_skips	\
0	0.386860	-0.485850	0.830546	
1	-1.356432	-1.658003	-0.880549	
2	0.635902	0.979341	-0.538330	
3	2.130152	1.370059	0.146108	
4	-1.356432	0.002547	-0.709440	

	number_of_likes	number_of_liked_tracks_listened	\
0	0.583458	0.074885	
1	-1.370041	-1.287405	
2	1.141601	0.269498	
3	0.025315	0.853336	
4	0.304387	0.853336	

	number_of_tracks_in_favourite_genre	\
0	0.863598	
1	-0.641583	
2	-0.453435	
3	-0.453435	
4	-0.641583	

	total_number_of_favourite_genres_listened	\
0	0.698899	
1	-0.608941	
2	-0.445461	
3	-0.445461	
4	-0.608941	

	average_popularity_in_favourite_genres	total_tracks_duration_ms	\
0	-0.022313	-0.435282	
1	-1.010011	-1.626428	
2	1.421246	0.666986	
3	1.421246	1.140734	
4	-1.010011	-0.253779	

number_of_different_artists	...	average_danceability	average_energy	\
-----------------------------	-----	----------------------	----------------	---

0	-0.501295	...	-1.659065	1.342909
1	-1.697606	...	0.922672	-0.572892
2	1.293171	...	0.854068	0.003184
3	1.053909	...	0.382945	-0.246091
4	-0.022771	...	0.888335	-0.085594

	average_instrumentalness	average_liveness	average_loudness	\
0	2.449911	0.998213	-0.832375	
1	-1.008311	-1.123035	0.964094	
2	-0.380334	-0.359967	0.729621	
3	-0.668866	-0.455298	0.475303	
4	-0.349358	-0.589086	1.084777	

	average_speechiness	average_tempo	average_valence	\
0	-1.197827	1.422952	0.127103	
1	1.752697	0.335856	-0.193103	
2	0.770775	-0.575114	1.033064	
3	0.411620	-1.015395	-0.639460	
4	1.112326	0.215161	-0.084959	

	average_track_name_length	average_daily_cost
0	1.804459	-1.246425
1	-1.031041	0.386880
2	-0.687234	1.524497
3	-1.118217	1.030821
4	-0.747340	0.052786

[5 rows x 25 columns]

```
Y_train.head()
```

	premium_user_numerical	will_buy_premium_next_month_numerical
64816	1	0
59448	0	0
95390	0	0
77685	0	0
115377	0	0

Modele

Do porównywania wybraliśmy cztery modele:

- **Dummy** - naiwny model, który zawsze przewiduje najczęściej występującą klasę
- **Logistic Regression** - model regresji logistycznej z domyślnymi parametrami
- **XGB Classifier** - model XGBoost z domyślnymi parametrami
- **XGB Classifier with Randomized Search** - model XGBoost z Randomized Search. Randomized Search to metoda optymalizacji hiperparametrów, która polega na losowym testowaniu zdefiniowane wartości hiperparametrów i ustaleniu ich najlepszej kombinacji. W ten sposób można znaleźć dobre parametry modelu bez konieczności przeszukiwania całej przestrzeni hiperparametrów. Dodatkowo, aby przeciwdziałać niezbalansowanym danym ustawiliśmy parametr `scale_pos_weight` według zaleceń na stosunek liczby negatywnych rekordów (0) do liczby pozytywnych (1).

```
DUMMY = 'dummy'
LOGISTIC_REG = 'logistic_regression'
XGB = 'xgb_classifier'
XGB_BEST_ESTIMATOR = 'xgb_classifier_best_estimator'
RANDOM = 'randomized_search'
```

```

MODEL_TYPES = [DUMMY, LOGISTIC_REG, XGB, XGB_BEST_ESTIMATOR]

def construct_dummy(X_train: pd.DataFrame,
                   y_train: pd.DataFrame,
                   params: Optional[Dict[str, Any]] = None) -> DummyClassifier:
    return DummyClassifier().fit(X_train, y_train)

def construct_logistic_regression(X_train: pd.DataFrame,
                                 y_train: pd.DataFrame,
                                 params: Optional[Dict[str, Any]] = None) -> LogisticRegression:
    return LogisticRegression().fit(X_train, y_train)

def construct_xgb_classifier(X_train: pd.DataFrame,
                             y_train: pd.DataFrame,
                             params: Optional[Dict[str, Any]] = None) -> XGBClassifier:
    return XGBClassifier().fit(X_train, y_train)

def construct_xgb_classifier_with_randomized_search(
    X_train: pd.DataFrame,
    y_train: pd.DataFrame,
    params: Optional[Dict[str, Any]] = None
) -> XGBClassifier:
    if params:
        return XGBClassifier(**params).fit(X_train, y_train)
    scale = y_train.value_counts()
    model = XGBClassifier(scale_pos_weight=scale[0] / scale[1])

    randomized_search_cv = RandomizedSearchCV(
        estimator=model,
        param_distributions={
            'max_depth': range(3, 25),
            'eta': uniform(0, 0.3),
            'gamma': uniform(0, 1),
            'n_estimators': range(10, 100),
        },
        n_iter=120,
        scoring='roc_auc',
        n_jobs=-1,
        verbose=3,
    )
    estimator = randomized_search_cv.fit(X_train, y_train)
    best_estimator = estimator.best_estimator_
    print('Best parameters:', best_estimator.get_params())
    return best_estimator # type: ignore

MODEL_CONSTRUCTORS = {
    DUMMY: construct_dummy,
    LOGISTIC_REG: construct_logistic_regression,
    XGB: construct_xgb_classifier,
    XGB_BEST_ESTIMATOR: construct_xgb_classifier_with_randomized_search
}

```



```

}
MODELS: Dict[str, Dict[str, Model]] = {}

for type in MODEL_TYPES:
    MODELS[type] = {
        target: MODEL_CONSTRUCTORS[type](X_train, Y_train[target])
        for target in TARGETS
    }

```

Ocena modeli

Posiadamy niezbalansowane dane, dlatego też do oceny modeli wykorzystaliśmy metrykę ROC-AUC, która jest miarą jakości klasyfikatora binarnego.

ROC-AUC mierzy zdolność modelu do rozróżnienia między dwiema klasami poprzez obliczenie powierzchni pod krzywą ROC. Krzywa ROC przedstawia zależność między wskaźnikiem True Positive Rate = $TP / (TP + FN)$ (czułość) a False Positive Rate = $FP / (FP + TN)$ (specyficzność). Wyższa wartość ROC-AUC oznacza lepszą zdolność modelu do rozróżniania klas.

Nie wykorzystaliśmy metryki *accuracy*, ponieważ w przypadku niezbalansowanych danych, może ona być myląca. Przykładowo, jeśli mamy 1000 obserwacji, z czego 900 należy do klasy 0, a 100 do klasy 1, to model, który zawsze zwraca 0, będzie miał *accuracy* 90%.

```

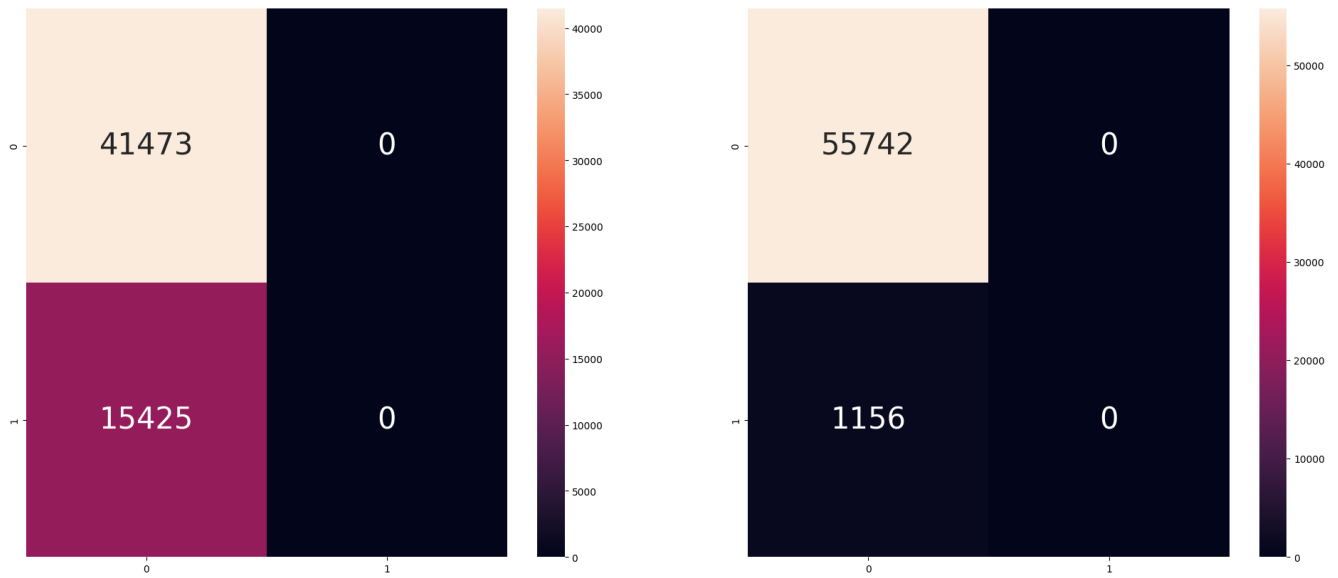
for type in MODEL_TYPES:
    print(type.upper())
    _, axs = plt.subplots(1, 2, figsize=(24, 10)) # type: ignore
    for i, target in enumerate(TARGETS):
        model = MODELS[type][target]
        y_predicted = model.predict(X_test)
        y_true = Y_test[target]
        roc_auc_score_value = roc_auc_score(y_true, y_predicted)
        print(f"ROC AUC score for {target}: {roc_auc_score_value}")
        matrix = confusion_matrix(y_true, y_predicted)
        sns.heatmap(
            matrix,
            annot=True,
            annot_kws={"fontsize": 30},
            fmt='g',
            xticklabels=["0", "1"], # type: ignore
            yticklabels=["0", "1"], # type: ignore
            ax=axs[i] # type: ignore
        )
    plt.show()

```

DUMMY

ROC AUC score for premium_user_numerical: 0.5

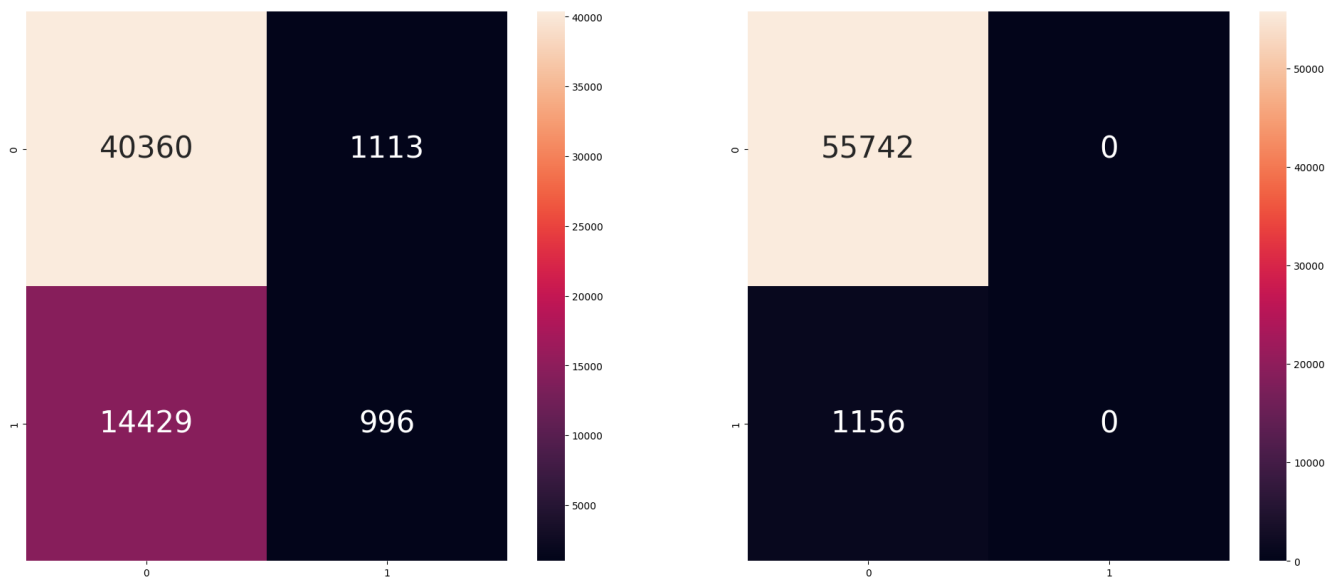
ROC AUC score for will_buy_premium_next_month_numerical: 0.5



LOGISTIC_REGRESSION

ROC AUC score for premium_user_numerical: 0.5188668826384125

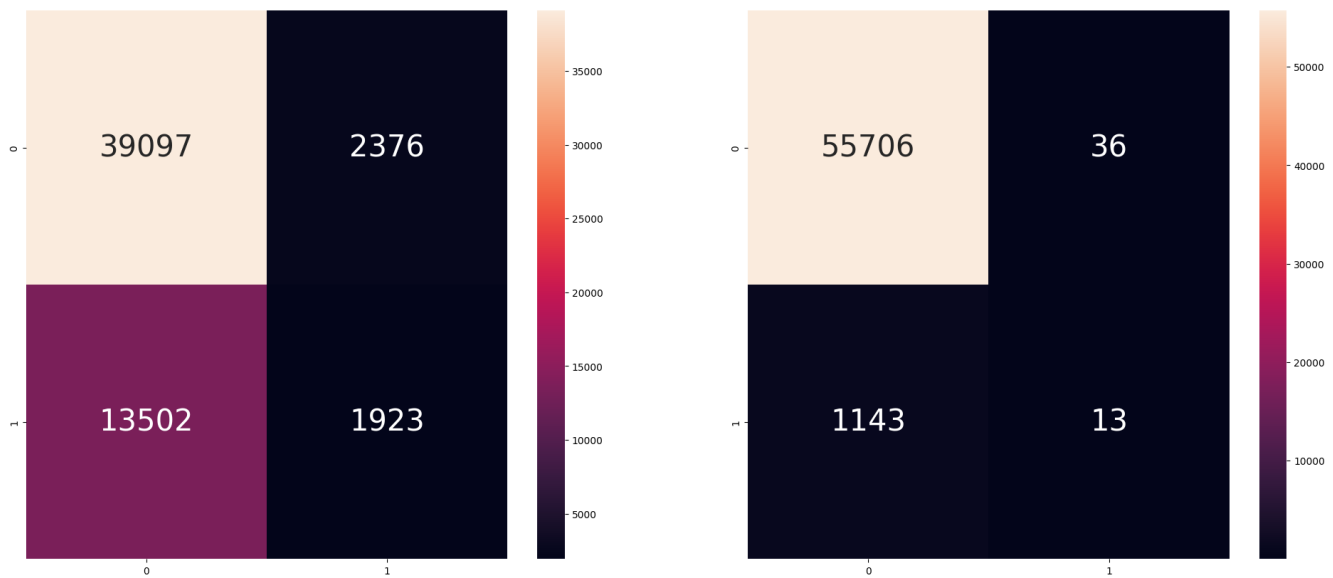
ROC AUC score for will_buy_premium_next_month_numerical: 0.5



XGB_CLASSIFIER

ROC AUC score for premium_user_numerical: 0.5336887309589363

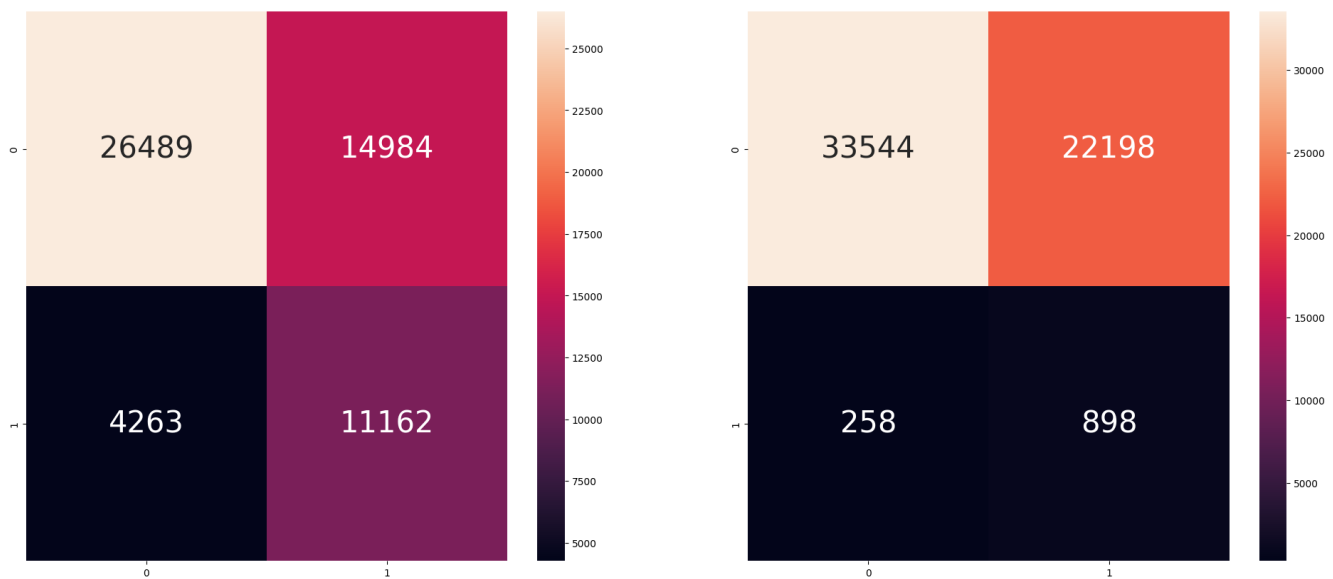
ROC AUC score for will_buy_premium_next_month_numerical: 0.5052999210773211



XGB_CLASSIFIER_BEST_ESTIMATOR

ROC AUC score for premium_user_numerical: 0.6811675847296093

ROC AUC score for will_buy_premium_next_month_numerical: 0.6892945303243974



Analizując wyniki możemy zauważyć, że dla przewidywania `premium_user_numerical` (czy użytkownik kiedykolwiek zakupi premium) najgorzej poradził sobie model naiwny `Dummy`, który każdemu przypisuje klasę większościową. Nieznacznie lepsze wyniki na podobnym poziomie, osiągnęły modele `Logistic Regression` oraz `XGB Classifier`. Najlepsze wyniki osiągnął model `XGB Classifier with Randomized Search`, dzięki optymalizacji hiperparametrów. W przypadku `will_buy_premium_next_month_numerical` (czy użytkownik kupi premium w przeciągu miesiąca) modele `Dummy` oraz `Logistic Regression` w każdym przypadku przewidywały klasę większościową. Model `XGB Classifier` był nieznacznie lepszy. Jedynie model `XGB Classifier with Randomized Search` osiągnął lepszy wynik 0.69, jednak kosztem przypisania większej ilości błędnych predykcji w przypadku klasy większościowej.

Istotność parametrów

```
def retrieve_weights(model: Model) -> np.ndarray[np.float64]:
    if isinstance(model, LogisticRegression):
        return model.coef_[0]
    if isinstance(model, XGBClassifier):
```

```

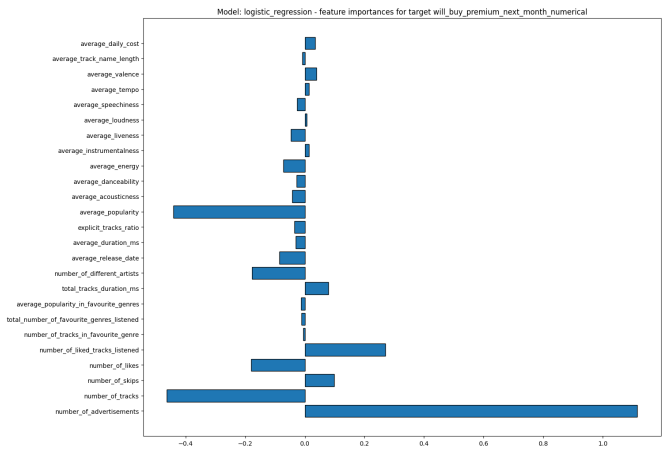
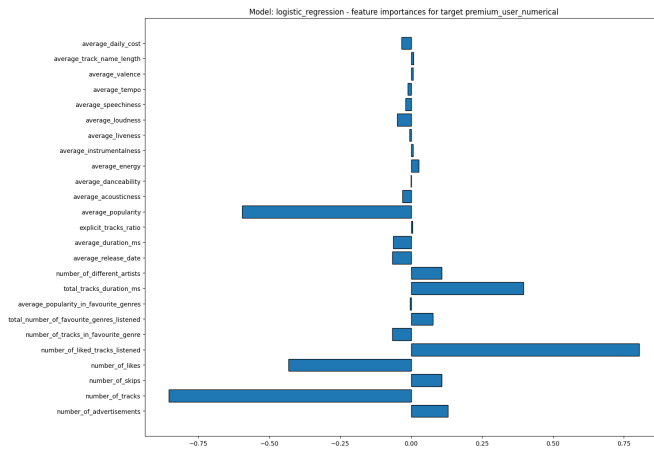
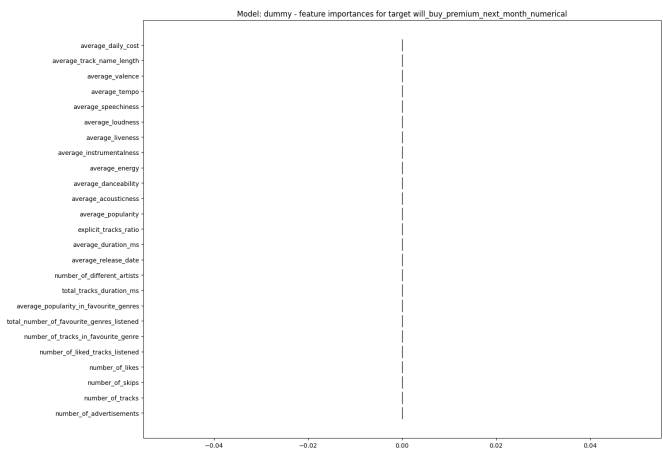
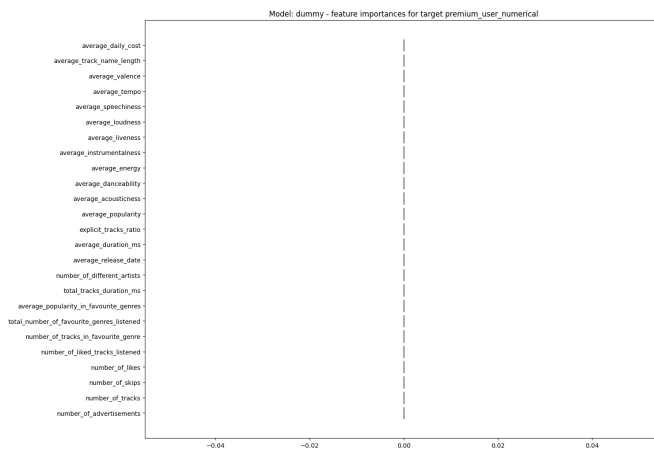
    return model.feature_importances_
return np.zeros(len(FEATURES))

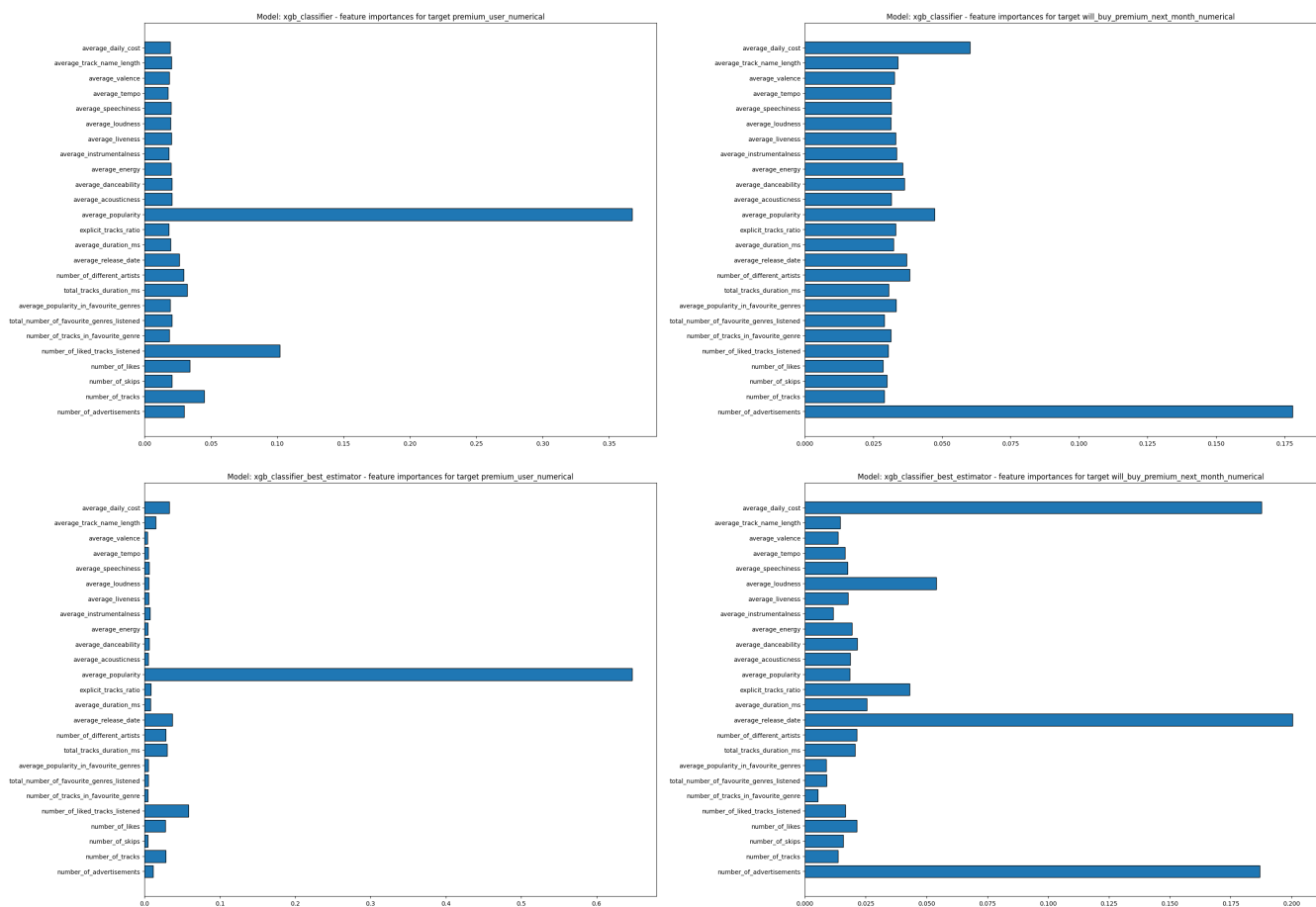
```

```

for type in MODEL_TYPES:
    _, axs = plt.subplots(1, len(TARGETS), figsize=(
        30, 10), constrained_layout=True)
    for i, target in enumerate(TARGETS):
        model = MODELS[type][target]
        columns = FEATURES
        weights = retrieve_weights(model)
        axs[i].barh(y=columns, width=weights, edgecolor="black")
        axs[i].set_title(
            f"Model: {type} - feature importances for target {target}")
plt.show()

```





Analizując ważność parametrów możemy zauważyć, że większość parametrów jest brana pod uwagę przez modele, jednakże kilka z nich wyróżnia się na tle pozostałych. W przypadku przewidywania dla tego, czy użytkownik zakupi premium w przeciągu miesiąca najważniejszym parametrem jest **number_of_advertisements**, czyli liczba wyświetlanych reklam. Dodatkowo ostatni wytrenowany model uwzględnia jeszcze **average_release_date** oraz **average_daily_cost**. Możemy z tego wnioskować, że aby użytkownik jak najszybciej zakupił premium powinniśmy manipulować ilością wyświetlanych mu reklam. Natomiast w przypadku przewidywania tego, czy użytkownik kiedykolwiek zakupi premium ważniejsze okazuje się **average_popularity** oraz **number_of_liked_tracks_listened** co oznacza, że użytkownicy, którzy słuchają popularniejszych utworów oraz słuchają polubionych utworów są bardziej skłonni do zakupu premium. Może oznaczać, to, że w długofalowej perspektywie ważniejsze może być proponowanie użytkownikowi utworów, które są popularne oraz utworów, które użytkownik polubił, niż wyświetlanie reklam.

Eksperymenty A/B

Trenujemy wszystkie modele na danych do 2023, a wyniki dla wszystkich modeli zapisujemy do plików pkl. Uruchamiamy mikroserwis, który wczytuje te modele. Następnie dane użytkowników korzystających w roku 2023 dzielimy na różne rzeczywistości i dla każdej z tych grup wykonujemy predykcję z wykorzystaniem naszego mikroserwisu, a następnie przeprowadzamy porównanie za pomocą testu t-studenta.

```
def get_params(model: Model) -> Optional[Dict[str, Any]]:
    if isinstance(model, XGBClassifier):
        return model.get_params()
    return None
```

```
X_train = pd.DataFrame(
    pipeline.fit_transform(TRAIN_DATA[FEATURES]),
    columns=FEATURES
)
```

```

Y_train = TRAIN_DATA[TARGETS]
for type in MODEL_TYPES:
    estimators = {}
    for target in TARGETS:
        y_train = Y_train[target]
        estimators[target] = MODEL_CONSTRUCTORS[type](
            X_train, y_train, get_params(MODELS[type][target])
        )
    model = IUMModel(pipeline, estimators)
    with open(f'models/{type}.pkl', 'wb') as f:
        pickle.dump(model, f)

```

```

random_ordered_ids = np.random.permutation(TEST_DATA['user_id'].unique())
size = len(random_ordered_ids) // len(MODEL_TYPES)

```

```

REALITIES: Dict[str, pd.DataFrame] = {}

```

```

for i, type in enumerate(MODEL_TYPES):
    ids = random_ordered_ids[i * size:(i + 1) * size]
    mask = TEST_DATA['user_id'].isin(ids)
    REALITIES[type] = TEST_DATA.loc[mask]

```

```

for type in MODEL_TYPES:
    display(REALITIES[type].head())

```

	user_id	year	month	premium_user_numerical	\
942	11621	2023	3	1	
949	11621	2023	1	1	
2378	9065	2023	3	1	
2836	17778	2023	1	1	
2867	583	2023	3	1	

	will_buy_premium_next_month_numerical	number_of_premium	\
942	1	1	
949	0	0	
2378	1	0	
2836	1	0	
2867	1	0	

	number_of_advertisements	number_of_tracks	number_of_skips	\
942	15	37	1	
949	13	36	2	
2378	11	36	0	
2836	9	32	1	
2867	11	31	15	

	number_of_likes	...	average_danceability	average_energy	\
942	5	...	0.666946	0.577703	
949	8	...	0.667389	0.615833	
2378	8	...	0.678528	0.589583	
2836	10	...	0.616688	0.566781	
2867	11	...	0.441774	0.660419	

	average_instrumentalness	average_liveness	average_loudness	\
942	0.015345	0.137381	-6.718757	
949	0.029088	0.171839	-6.333861	
2378	0.030580	0.149925	-6.894583	

2836	0.097120	0.207247	-7.540094
2867	0.066486	0.241710	-8.348548

	average_speechiness	average_tempo	average_valence \
942	0.084957	108.457351	0.506541
949	0.090872	115.756556	0.561761
2378	0.083008	114.375944	0.515417
2836	0.089953	113.637219	0.393766
2867	0.060265	123.085516	0.419452

	average_track_name_length	average_daily_cost
942	13.189189	0.023551
949	13.583333	0.023168
2378	14.194444	0.022362
2836	12.125000	0.024966
2867	22.258065	0.013564

[5 rows x 31 columns]

	user_id	year	month	premium_user_numerical \
328	7643	2023	1	1
331	7643	2023	1	1
457	18501	2023	1	1
2943	10173	2023	2	1
3612	16245	2023	3	1

	will_buy_premium_next_month_numerical	number_of_premium \
328	0	0
331	0	0
457	1	0
2943	1	0
3612	1	0

	number_of_advertisements	number_of_tracks	number_of_skips \
328	10	26	11
331	7	26	14
457	13	28	13
2943	11	36	2
3612	9	34	16

	number_of_likes	...	average_danceability	average_energy \
328	11	...	0.481962	0.644192
331	9	...	0.499385	0.719962
457	10	...	0.478964	0.671143
2943	5	...	0.611306	0.557056
3612	13	...	0.463882	0.683147

	average_instrumentalness	average_liveness	average_loudness \
328	0.122086	0.228292	-9.241962
331	0.071406	0.198385	-8.612577
457	0.065366	0.224946	-8.977607
2943	0.008194	0.135256	-7.189000
3612	0.080322	0.310679	-8.806382

	average_speechiness	average_tempo	average_valence \
328	0.066550	125.406462	0.468627
331	0.056546	124.025731	0.591512

457	0.057068	127.819821	0.572821
2943	0.059411	113.502861	0.495778
3612	0.084926	124.870147	0.498176

	average_track_name_length	average_daily_cost
328	25.692308	0.013864
331	22.192308	0.012140
457	21.571429	0.012536
2943	14.416667	0.022563
3612	23.264706	0.012730

[5 rows x 31 columns]

	user_id	year	month	premium_user_numerical	\
41	4074	2023	3	1	
47	4074	2023	1	1	
474	1944	2023	1	1	
483	1944	2023	1	1	
2905	8001	2023	1	1	

	will_buy_premium_next_month_numerical	number_of_premium	\
41	1	0	
47	0	0	
474	1	0	
483	1	0	
2905	1	0	

	number_of_advertisements	number_of_tracks	number_of_skips	\
41	1	6	0	
47	13	37	3	
474	7	27	13	
483	5	13	7	
2905	14	37	19	

	number_of_likes	...	average_danceability	average_energy	\
41	2	...	0.626167	0.560333	
47	9	...	0.673216	0.623946	
474	10	...	0.505333	0.657370	
483	7	...	0.488308	0.625231	
2905	4	...	0.440108	0.691568	

	average_instrumentalness	average_liveness	average_loudness	\
41	0.005269	0.148200	-6.978667	
47	0.022808	0.185284	-6.816027	
474	0.132346	0.151230	-9.610593	
483	0.189436	0.269831	-9.640538	
2905	0.107200	0.302508	-9.375378	

	average_speechiness	average_tempo	average_valence	\
41	0.128867	134.761333	0.527667	
47	0.093181	115.215946	0.511486	
474	0.081237	129.688852	0.569074	
483	0.051131	125.086231	0.607846	
2905	0.073959	117.028243	0.418997	

	average_track_name_length	average_daily_cost
41	7.333333	0.019577

47	11.405405	0.026657
474	19.925926	0.011697
483	31.461538	0.012755
2905	24.783784	0.014111

[5 rows x 31 columns]

	user_id	year	month	premium_user_numerical	\
378	14848	2023	3	1	
443	18214	2023	1	1	
2129	5945	2023	1	1	
2611	15895	2023	3	1	
3385	15047	2023	1	1	

	will_buy_premium_next_month_numerical	number_of_premium	\
378	1	0	
443	1	0	
2129	1	1	
2611	1	0	
3385	1	1	

	number_of_advertisements	number_of_tracks	number_of_skips	\
378	15	36	2	
443	4	16	9	
2129	17	32	0	
2611	2	5	3	
3385	14	39	19	

	number_of_likes	...	average_danceability	average_energy	\
378	5	...	0.648306	0.512556	
443	3	...	0.480187	0.652187	
2129	9	...	0.654094	0.547312	
2611	2	...	0.513200	0.439800	
3385	14	...	0.487282	0.719385	

	average_instrumentalness	average_liveness	average_loudness	\
378	0.027164	0.155794	-7.668833	
443	0.045305	0.223931	-9.035750	
2129	0.045521	0.120609	-7.680031	
2611	0.025822	0.216700	-12.256600	
3385	0.127397	0.201549	-8.873872	

	average_speechiness	average_tempo	average_valence	\
378	0.087006	106.619583	0.463817	
443	0.043562	115.447125	0.474750	
2129	0.067134	111.153844	0.473512	
2611	0.040380	129.347000	0.483400	
3385	0.063238	125.130744	0.527464	

	average_track_name_length	average_daily_cost
378	13.333333	0.023180
443	24.375000	0.014265
2129	12.312500	0.023636
2611	34.600000	0.011271
3385	23.615385	0.012737

[5 rows x 31 columns]

```

result = {
    type: {
        target: pd.DataFrame({
            "guess": [],
            "ground_truth": [],
            "model": [],
            "year": [],
            "month": [],
            "user_id": [],
        })
        for target in TARGETS
    }
    for type in MODEL_TYPES
}

for type in MODEL_TYPES:
    url = f'http://127.0.0.1:5000/predict/{type}'
    for i in range(0, len(TEST_DATA)):
        row = TEST_DATA.iloc[i].to_dict()
        response = requests.post(url, json=row).json()
        for target in TARGETS:
            current = pd.DataFrame({
                "guess": [1 if response[target] else 0],
                "ground_truth": [row[target]],
                "model": [type],
                "year": [row['year']],
                "month": [row['month']],
                "user_id": [row['user_id']],
            })
            result[type][target] = pd.concat(
                [result[type][target], current], ignore_index=True
            )

```

```

for type in MODEL_TYPES:
    print(type.upper())
    for target in TARGETS:
        print(target)
        print()
        print(result[type][target].guess.value_counts())
        print()
        print(result[type][target].ground_truth.value_counts())
        roc_auc_score_value = roc_auc_score(
            result[type][target].ground_truth, result[type][target].guess
        )
        print()
        print('ROC AUC score = ', roc_auc_score_value)
        print()

```

DUMMY

premium_user_numerical

0.0 9972

Name: guess, dtype: int64

0.0 9707

1.0 265

Name: ground_truth, dtype: int64

```

ROC AUC score = 0.5

will_buy_premium_next_month_numerical

0.0    9972
Name: guess, dtype: int64

0.0    9808
1.0     164
Name: ground_truth, dtype: int64

ROC AUC score = 0.5

LOGISTIC_REGRESSION
premium_user_numerical

0.0    9809
1.0     163
Name: guess, dtype: int64

0.0    9707
1.0     265
Name: ground_truth, dtype: int64

ROC AUC score = 0.5303700305750956

will_buy_premium_next_month_numerical

0.0    9972
Name: guess, dtype: int64

0.0    9808
1.0     164
Name: ground_truth, dtype: int64

ROC AUC score = 0.5

XGB_CLASSIFIER
premium_user_numerical

0.0    9600
1.0     372
Name: guess, dtype: int64

0.0    9707
1.0     265
Name: ground_truth, dtype: int64

ROC AUC score = 0.5389876202934665

will_buy_premium_next_month_numerical

0.0    9967
1.0      5
Name: guess, dtype: int64

```

```
0.0    9808
1.0     164
Name: ground_truth, dtype: int64
```

ROC AUC score = 0.5090443838777703

```
XGB_CLASSIFIER_BEST_ESTIMATOR
premium_user_numerical
```

```
0.0    6858
1.0    3114
Name: guess, dtype: int64
```

```
0.0    9707
1.0     265
Name: ground_truth, dtype: int64
```

ROC AUC score = 0.6380987460906447

```
will_buy_premium_next_month_numerical
```

```
0.0    6630
1.0    3342
Name: guess, dtype: int64
```

```
0.0    9808
1.0     164
Name: ground_truth, dtype: int64
```

ROC AUC score = 0.6737021545378585

```
for type in MODEL_TYPES:
    for target in TARGETS:
        result[type][target].to_csv(f'ab_experiment/{type}-{target}.csv')
```

Ustalamy hipotezę zerową H_0 mówiącą, że pierwszy model nie jest lepszy od drugiego oraz hipotezę alternatywną H_1 głoszącą, że model pierwszy jest lepszy od modelu drugiego.

Na podstawie tabeli rozkładu t-studenta, przyjętego istotności statystycznej jako 0.05 oraz stopni swobody $12+12-2 = 22$ ustaliliśmy wartość parametru t_α jako 2.074

$$t = \frac{\bar{q}_A - \bar{q}_B}{s_p \sqrt{\frac{1}{n_A} + \frac{1}{n_B}}}$$

$$s_p = \sqrt{\frac{(n_A - 1)\sigma^2 + (n_B - 1)\sigma^2}{n_A + n_B - 2}}$$

```
BUCKETS = 12
T_ALPHA = 2.074
```

```
def s_p(sigma_A: float, sigma_B: float) -> float:
    return sqrt(
        (BUCKETS - 1) * (sigma_A ** 2) + (BUCKETS - 1) * (sigma_B ** 2)
        / (BUCKETS + BUCKETS - 2)
```

```

)

def t(q_A: float, q_B: float, s_p_value: float) -> float:
    return (q_A - q_B) / (s_p_value * sqrt(1 / BUCKETS + 1 / BUCKETS))

for type_A, type_B in itertools.product(MODEL_TYPES, MODEL_TYPES):
    if type_A == type_B:
        continue
    print(f'{type_A} vs {type_B}'.upper())
    print()
    for target in TARGETS:
        print(target)
        reality_A: pd.DataFrame = result[type_A][target]
        reality_B: pd.DataFrame = result[type_B][target]

        data = pd.concat([reality_A, reality_B])
        random_ordered_ids = np.random.permutation(
            data['user_id'].unique()
        )
        size = len(random_ordered_ids) // BUCKETS

        reality_A_score = []
        reality_B_score = []

        for bucket in range(BUCKETS):
            ids = random_ordered_ids[bucket * size:(i + 1) * size]
            mask = data['user_id'].isin(ids)
            bucket_data = data.loc[mask]
            reality_A_data = bucket_data.loc[bucket_data['model'] == type_A]
            reality_B_data = bucket_data.loc[bucket_data['model'] == type_B]

            reality_A_score.append(
                roc_auc_score(
                    reality_A_data['ground_truth'],
                    reality_A_data['guess'],
                )
            )
            reality_B_score.append(
                roc_auc_score(
                    reality_B_data['ground_truth'],
                    reality_B_data['guess'],
                )
            )

        s_p_value = s_p(stdev(reality_A_score), stdev(reality_B_score))
        if s_p_value != 0:
            t_value = t(mean(reality_A_score), mean(
                reality_B_score), s_p_value)
        else:
            t_value = 0

        if t_value > T_ALPHA:
            print(f'{type_A} is better than {type_B}')
        else:
            print(f'We can\'t say that {type_A} is better than {type_B}')

```

```
print()
```

DUMMY VS LOGISTIC_REGRESSION

```
premium_user_numerical
```

```
We can't say that dummy is better than logistic_regression
```

```
will_buy_premium_next_month_numerical
```

```
We can't say that dummy is better than logistic_regression
```

DUMMY VS XGB_CLASSIFIER

```
premium_user_numerical
```

```
We can't say that dummy is better than xgb_classifier
```

```
will_buy_premium_next_month_numerical
```

```
We can't say that dummy is better than xgb_classifier
```

DUMMY VS XGB_CLASSIFIER_BEST_ESTIMATOR

```
premium_user_numerical
```

```
We can't say that dummy is better than xgb_classifier_best_estimator
```

```
will_buy_premium_next_month_numerical
```

```
We can't say that dummy is better than xgb_classifier_best_estimator
```

LOGISTIC_REGRESSION VS DUMMY

```
premium_user_numerical
```

```
We can't say that logistic_regression is better than dummy
```

```
will_buy_premium_next_month_numerical
```

```
We can't say that logistic_regression is better than dummy
```

LOGISTIC_REGRESSION VS XGB_CLASSIFIER

```
premium_user_numerical
```

```
We can't say that logistic_regression is better than xgb_classifier
```

```
will_buy_premium_next_month_numerical
```

```
We can't say that logistic_regression is better than xgb_classifier
```

LOGISTIC_REGRESSION VS XGB_CLASSIFIER_BEST_ESTIMATOR

```
premium_user_numerical
```

```
We can't say that logistic_regression is better than xgb_classifier_best_estimator
```

```
will_buy_premium_next_month_numerical
```

```
We can't say that logistic_regression is better than xgb_classifier_best_estimator
```

XGB_CLASSIFIER VS DUMMY

```
premium_user_numerical
```

```
We can't say that xgb_classifier is better than dummy
```

```
will_buy_premium_next_month_numerical
```

We can't say that xgb_classifier is better than dummy

XGB_CLASSIFIER VS LOGISTIC_REGRESSION

premium_user_numerical

We can't say that xgb_classifier is better than logistic_regression

will_buy_premium_next_month_numerical

We can't say that xgb_classifier is better than logistic_regression

XGB_CLASSIFIER VS XGB_CLASSIFIER_BEST_ESTIMATOR

premium_user_numerical

We can't say that xgb_classifier is better than xgb_classifier_best_estimator

will_buy_premium_next_month_numerical

We can't say that xgb_classifier is better than xgb_classifier_best_estimator

XGB_CLASSIFIER_BEST_ESTIMATOR VS DUMMY

premium_user_numerical

xgb_classifier_best_estimator is better than dummy

will_buy_premium_next_month_numerical

xgb_classifier_best_estimator is better than dummy

XGB_CLASSIFIER_BEST_ESTIMATOR VS LOGISTIC_REGRESSION

premium_user_numerical

We can't say that xgb_classifier_best_estimator is better than logistic_regression

will_buy_premium_next_month_numerical

xgb_classifier_best_estimator is better than logistic_regression

XGB_CLASSIFIER_BEST_ESTIMATOR VS XGB_CLASSIFIER

premium_user_numerical

We can't say that xgb_classifier_best_estimator is better than xgb_classifier

will_buy_premium_next_month_numerical

xgb_classifier_best_estimator is better than xgb_classifier

Z przeprowadzonego eksperymentu wynika, że model XGB Classifier with Randomized Search jest lepszy od reszty modeli w przewidywaniu wartości will_buy_premium_next_month_numerical oraz jest lepszy od modelu naiwnego Dummy w przypadku przewidywania wartości premium_user_numerical.