

01

{P and P}



Marzec 29, 2020

Kickstarter ciąg dalszy nastąpił

Dataset
Data manipulation
ML Models
Models comparisons
Training
Coś jeszcze

Naprawdę nie wiedziałem
Co jeszcze dopisać
Product mockups
Więc wstawiłem
Video marketing
Cokolwiek

Problem

PROBLEM 1

Poprzedni release dał nam narzędzie porównawcze, ale chcielibyśmy takie, które daje jasną odpowiedź na pytanie - kickstarterować, czy nie kickstarterować?

PROBLEM 2

Tutaj już nie wystarczy analiza historycznych danych, potrzebny jest model machine learningowy - - byśmy mogli spojrzeć w przyszłość!

PROBLEM 3

Jak pogodzić potrzebę natychmiastowych rezultatów z koniecznością poświęcenia adekwatnego czasu na wyuczenie modeli?

04

ROZWIAZANIE 1

Stwórzmy aplikację, która robi dokładnie to - na podstawie wprowadzonych parametrów kampanii **stwierdzi, czy nowa się uda!**

ROZWIAZANIE 2

Użyjmy modeli **regresji i klasyfikacji**, by spojrzeć w przyszłość i orzec, czy dana kampania ma sens i szansę powodzenia.

ROZWIAZANIE 3

By zmieścić się w czasie musieliszy przygotować sprytny plan działania!

Rozwiązanie

PLANOWANIE

Chwila na zastanowienie, zanim
rzucimy się w wir pracy

DEV_DF

Taki myk, żeby dokonać
komparacji (mądre słowo) modeli

OPTYMALIZACJA

Wiemy, który model, teraz trzeba
dobrać wyposażenie w wersji Sport

ZASTOSOWANIE

My wbijamy model w apkę,
apka wbija użytkownika w siedzenie

Timeline
ETAPY PRAC NAD PROJEKTEM

PLANOWANIE

Wymagało od nas wielu dyskusji, skupienia i zdolności przewidywania problemów, jakie mogą nas po drodze do celu napotkać.

Podstawową kwestią był czas, jaki potrzebny jest do wyuczenia i modeli bazujących na dużym zbiorze danych (ponad 300 tys. rekordów).

PLANOWANIE

DEV_DF

Żeby przyspieszyć prace nad znalezieniem najodpowiedniejszego modelu dla naszego zbioru danych, przygotowaliśmy na jego podstawie podzbior roboczy.

DEV_DF

By jednak mieć pewność, że jest to podzbior reprezentatywny, musieliśmy dokonać analizy porównawczej obu zbiorów.

użyliśmy train_test_split i zadziałało *

df.describe

```
In [5]: rest_df.describe().apply(lambda s: s.apply(lambda x: format(x, 'g')))
```

Out[5]:

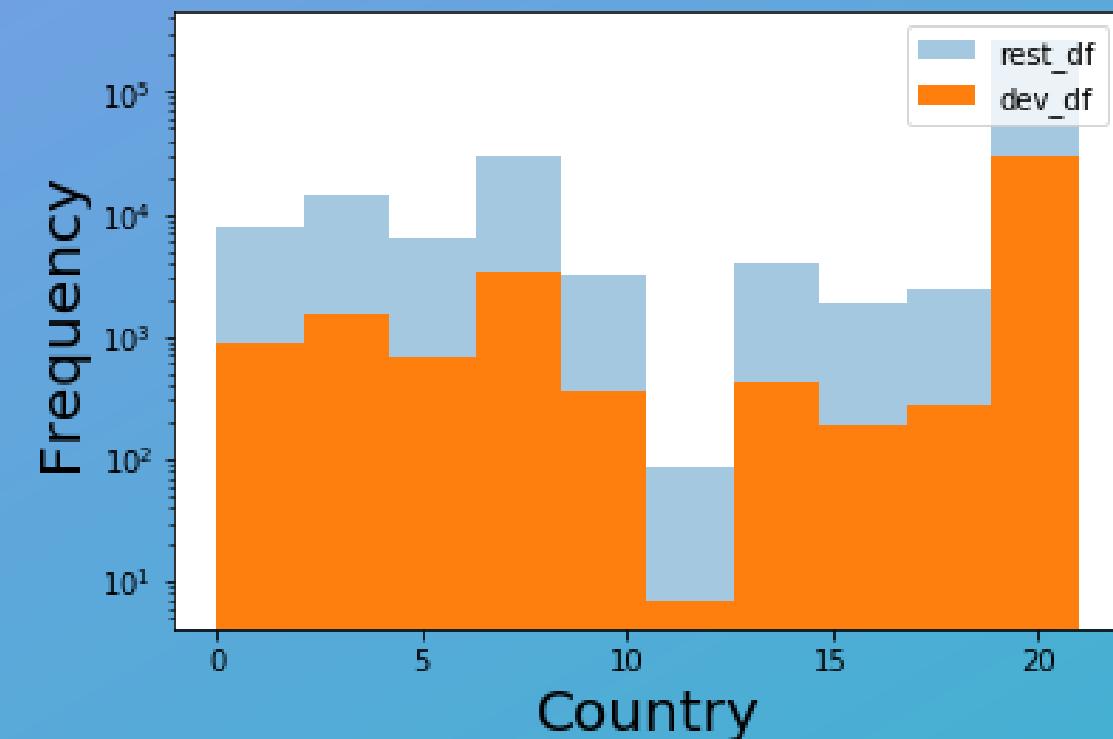
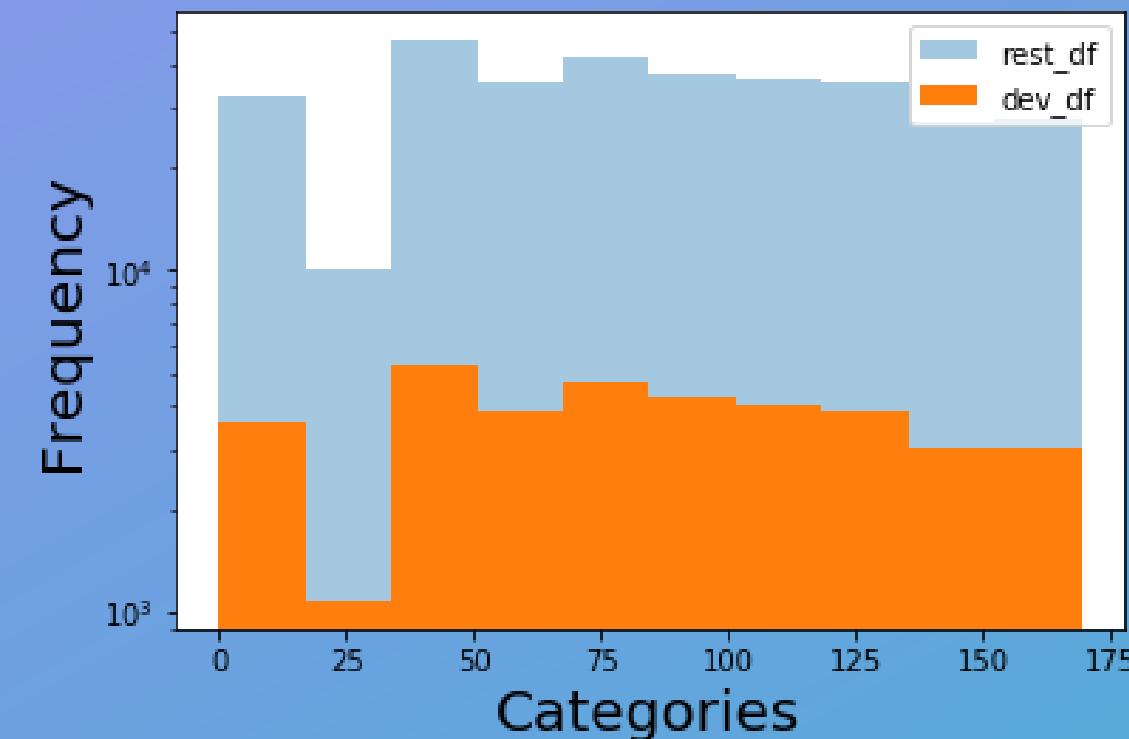
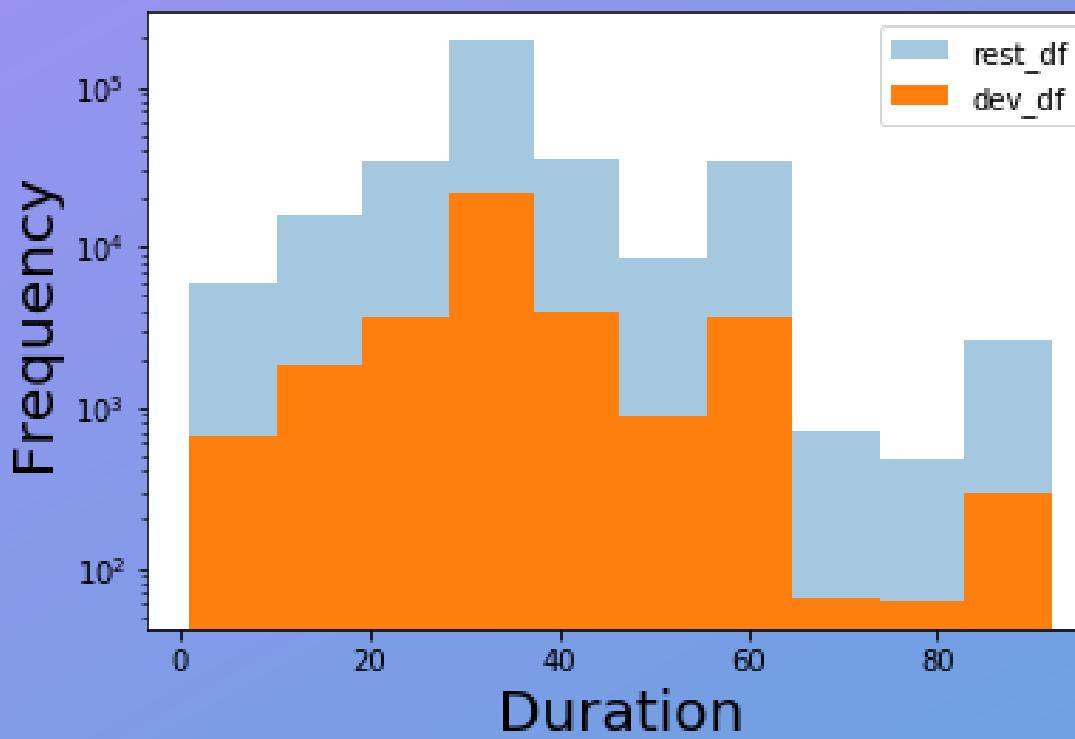
	main_cat_cat	country	duration	currency	goal_in_usd	percentage_of_money_collected	backers	state	pledged_in_usd
count	331414	331414	331414	331414	331414	331414	331414	331414	331414
mean	85.3333	17.9269	34.1812	11.0261	45035.5	3.38915	107.497	0.361098	9153.98
std	45.7419	6.22598	12.7918	3.92691	1.12815e+06	283.165	941.016	0.48032	93340.6
min	0	0	1	0	0.01	0	0	0	0
25%	50	21	30	13	2000	0.00476477	2	0	32
50%	84	21	30	13	5500	0.136	12	0	631
75%	120	21	37	13	16000	1.06667	57	1	4075
max	169	21	92	13	1.66361e+08	104278	219382	1	2.0339e+07

```
In [6]: dev_df.describe().apply(lambda s: s.apply(lambda x: format(x, 'g')))
```

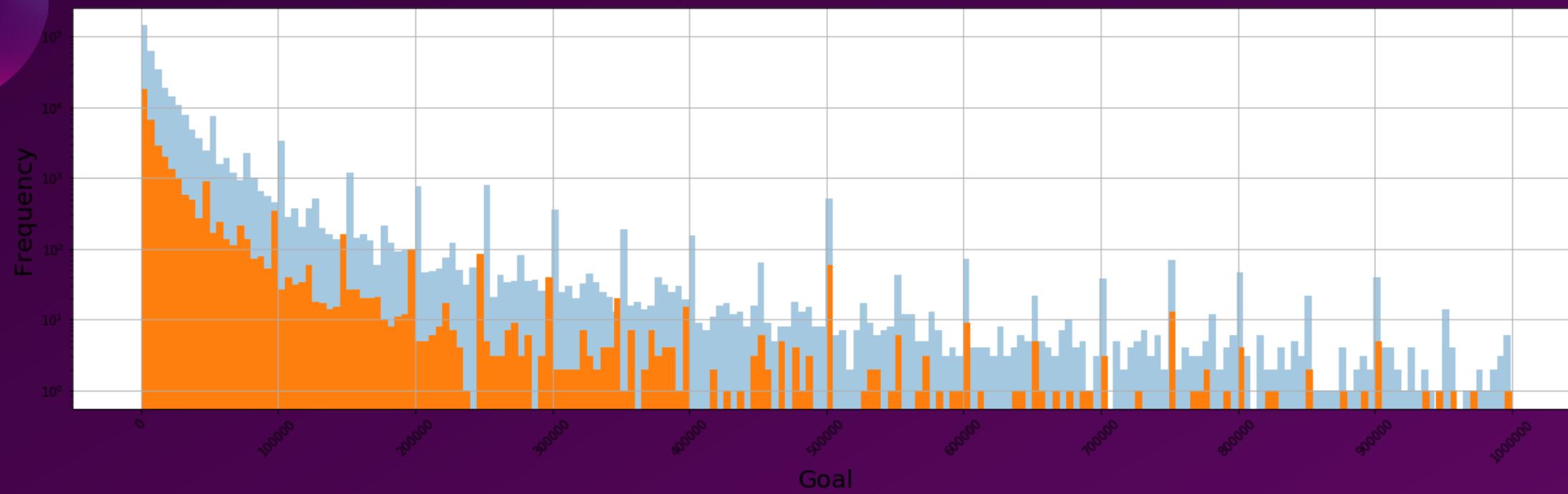
Out[6]:

	main_cat_cat	country	duration	currency	goal_in_usd	percentage_of_money_collected	backers	state	pledged_in_usd
count	36824	36824	36824	36824	36824	36824	36824	36824	36824
mean	85.1896	17.959	34.0792	11.0524	46872.9	2.37795	103.952	0.362481	9203.46
std	45.6717	6.2073	12.7356	3.90838	1.16228e+06	101.813	666.031	0.480723	76577.2
min	0	0	1	0	0.72	0	0	0	0
25%	50	21	30	13	2000	0.0048083	2	0	35
50%	84	21	30	13	5500	0.138301	12	0	633.015
75%	120	21	36	13	15951.7	1.06495	57	1	4113.17
max	169	21	92	13	1.04057e+08	12984	67226	1	9.19206e+06

9 CZĘSTOŚĆ WYSTĘPOWANIA SPECYFICZNYCH WARTOŚCI

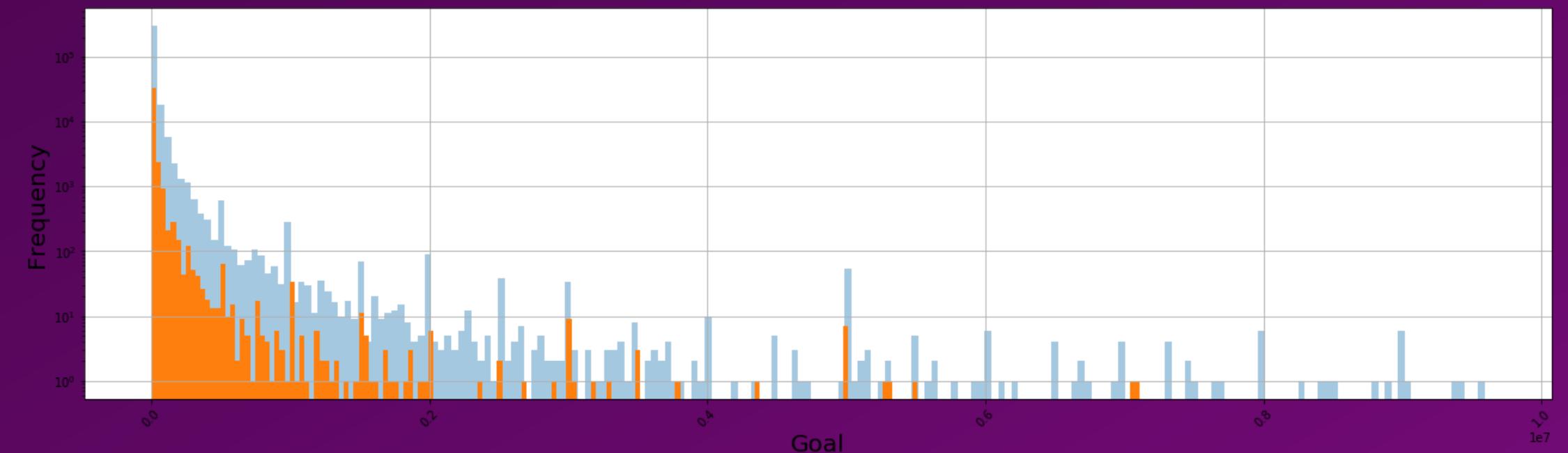


CZĘSTOŚĆ WYSTĘPOWANIA SPECYFICZNYCH WARTOŚCI



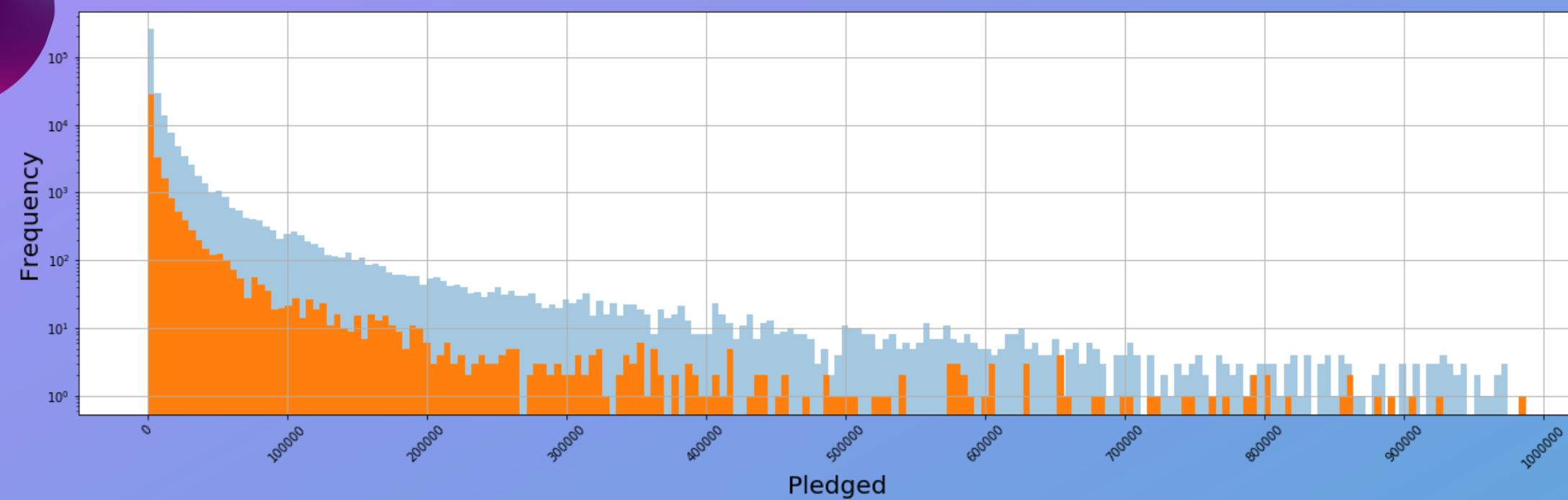
GOAL < 1.000.000\$

GOAL < 10.000.000\$

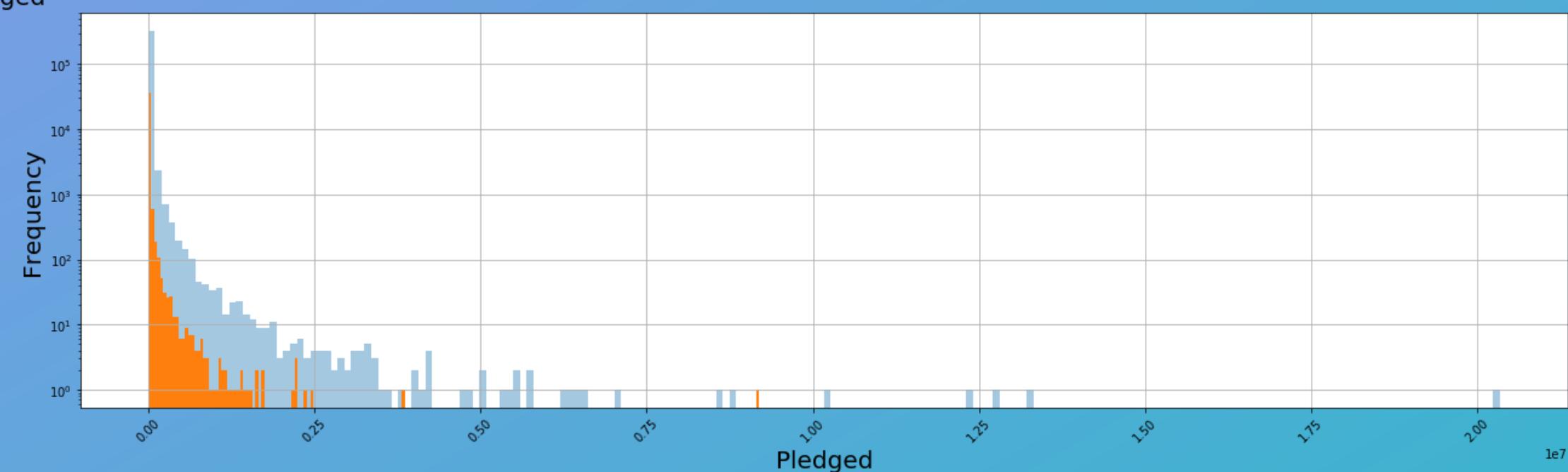


11

CZĘSTOŚĆ WYSTĘPOWANIA SPECYFICZNYCH WARTOŚCI



PLEDGE < 1.000.000\$



PLEDGE < 10.000.000\$

PLANOWANIE

Dysponując sprawdzonym DEV_DF,
mogliśmy przystąpić do przetrenowania
i porównania różnych modeli.

DEV_DF

Dzięki temu, że operowaliśmy na 10%
spośród wszystkich rekordów, poszło
całkiem zgrabnie.

OPTYMALIZACJA

Wliczając w to optymalizację finalnie
obranego modelu!

Szliśmy według planu:

- > Funkcja, która przy pomocy GridSearch szuka najlepszych parametrów
- > Funkcja wyliczająca score z walidacji, r2 i mse dla najlepszych modeli
- > Funkcja potrzebna do KNeighborsRegressor (normalizuje x tak by były od 0 do 1)
- > Podział próbek na treningowe i testowe
- > Stworzenie modeli regresji (do uzupełnienia params dla wszystkich modeli)
- > Upakowanie modeli do słownika
- > Szukanie najlepszych parametrów

```
def best_params(model, params, cv, x_train, y_train):  
    best_model = GridSearchCV(model, params, cv=cv)  
    best_model.fit(x_train, y_train)  
    best_options = best_model.best_params_  
    return best_options, best_model
```

```
def prediction_req(model, x_train, y_train, x_test, y_test, cv):  
    y_pred = model.predict(x_test)  
  
    score_val = np.mean(cross_val_score(model, x_train, y_train, cv=cv))  
    score_r2 = r2_score(y_test, y_pred)  
    score_mse = mean_squared_error(y_test, y_pred)  
    return score_val, score_r2, score_mse
```

```
def normalize_data(x_train, x_test):  
    scaler = MinMaxScaler()  
    scaler.fit(x_train)  
    x_train_norm = scaler.transform(x_train)  
    x_test_norm = scaler.transform(x_test)  
    return x_train_norm, x_test_norm
```

```
# DecisionTreeRegressor  
dtr_model = DecisionTreeRegressor()  
dtr_params = {}
```

```
# KNeighborsRegressor  
knr_model = KNeighborsRegressor()  
knr_params = {}
```

```
# RandomForestRegressor  
rfr_model = RandomForestRegressor()  
rfr_params = {}
```

```
# SVR  
svm_model = svm.SVR()  
svm_params = {}
```

```
# Xgboost  
xgb_model = xgb.XGBRegressor()  
xgb_params = {}
```

```
models={'DecisionTreeRegressor': (dtr_model,dtr_params),  
'KNeighborsRegressor': (knr_model, knr_params),  
'RandomForestRegressor' : (rfr_model, rfr_params),  
'SVR' : (svm_model, svm_params),  
'Xgboost' : (xgb_model, xgb_params)}
```

```
results=[]  
for key in models.keys():  
    if key=='KNeighborsRegressor':  
        best_options, best_model = best_params(models[key][0], models[key][1], 3, x_train_norm, y_train)  
        score_val, score_r2, score_mse = prediction_req(best_model, x_train_norm, y_train, x_test_norm, y_test, cv=3)  
    else:  
        best_options, best_model = best_params(models[key][0], models[key][1], 3, x_train, y_train)  
        score_val, score_r2, score_mse = prediction_req(best_model, x_train, y_train, x_test, y_test, cv=3)  
  
    results.append([key, best_options, score_val, score_r2, score_mse])  
print(key+' - done!')
```

Szliśmy według planu:

- > Funkcja, która przy pomocy GridSearch szuka najlepszych parametrów
- > Funkcja wyliczająca score z walidacji, r2 i mse dla najlepszych modeli
- > Funkcja potrzebna do KNeighborsRegressor (normalizuje x tak by były od 0 do 1)
- > Podział próbek na treningowe i testowe
- > Stworzenie modeli regresji (do uzupełnienia params dla wszystkich modeli)
- > Upakowanie modeli do słownika
- > Szukanie najlepszych parametrów

```
def best_params(model, params, cv, x_train, y_train):  
    best_model = GridSearchCV(model, params, cv=cv)  
    best_model.fit(x_train, y_train)  
    best_options = best_model.best_params_  
    return best_options, best_model
```

```
def prediction_reg(model, x_train, y_train, x_test, y_test, cv=3):  
    y_pred = model.predict(x_test)  
  
    score_val = np.mean(cross_val_score(model, x_train, y_train, cv=cv))  
    score_r2 = r2_score(y_test, y_pred)  
    score_mse = mean_squared_error(y_test, y_pred)  
    return score_val, score_r2, score_mse
```

```
def normalize_data(x_train, x_test):  
    scaler = MinMaxScaler()  
    scaler.fit(x_train)  
    x_train_norm = scaler.transform(x_train)  
    x_test_norm = scaler.transform(x_test)  
    return x_train_norm, x_test_norm
```

```
# DecisionTreeRegressor  
dtr_model = DecisionTreeRegressor()  
dtr_params = {}
```

```
# KNeighborsRegressor  
knr_model = KNeighborsRegressor()  
knr_params = {}
```

```
# RandomForestRegressor  
rfr_model = RandomForestRegressor()  
rfr_params = {}
```

```
# SVR  
svm_model = svm.SVR()  
svm_params = {}
```

```
# Xgboost  
xgb_model = xgb.XGBRegressor()  
xgb_params = {}
```

```
models = {'DecisionTreeRegressor': (dtr_model, dtr_params),  
          'KNeighborsRegressor': (knr_model, knr_params),  
          'RandomForestRegressor': (rfr_model, rfr_params),  
          'SVR': (svm_model, svm_params),  
          'Xgboost': (xgb_model, xgb_params)}
```

```
for key in models.keys():  
    if key == 'KNeighborsRegressor':  
        best_options, best_model = best_params(models[key][0], models[key][1], 3, x_train, y_train)  
        score_val, score_r2, score_mse = prediction_reg(best_model, x_train, y_train, x_test, y_test, cv=3)  
  
    results.append([key, best_options, score_val, score_r2, score_mse])  
    print(key + ' - done!')
```

ale daliśmy radę

```
▶ results_df = pd.DataFrame(results, columns=["model", "best_options", "validation", "r2", "mse"])
results_df
```

	model	best_options	validation	r2	mse
0	DecisionTreeRegressor	{}	-0.106433	-0.849711	6.391444e+09
1	KNeighborsRegressor	{}	0.087521	0.191412	2.793975e+09
2	RandomForestRegressor	{}	0.164173	0.127827	3.013683e+09
3	SVR	{}	-0.014351	-0.020496	3.526196e+09
4	Xgboost	{}	0.248432	0.116466	3.052940e+09

WYBRALIŚMY
XGBOOST

PLANOWANIE

Być może nie było tego widać,

ale na potrzeby uczenia modeli musieliśmy
enkodować i normalizować dane wsadowe.

DEV_DF

Musieliśmy też jakoś

umieścić wszystko w aplikacji.

OPTYMALIZACJA

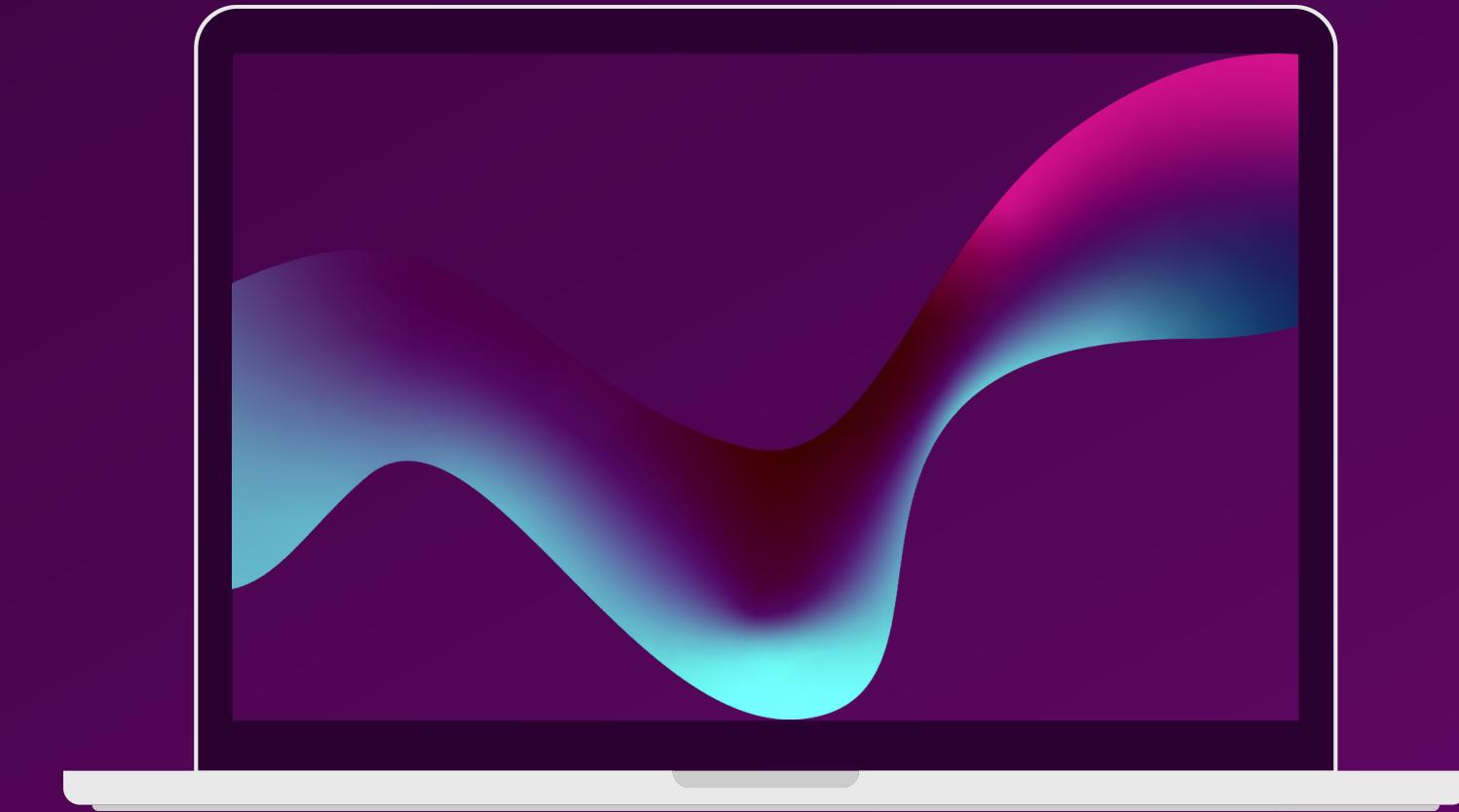
Z pomocą przyszły ogórki konserwowe ;)

Wiwat Pickle i możliwość zapisania

zarówno enkoderów, jak i modelu
do pythonowego obiektu!

ZASTOSOWANIE

Aplikacja



Zobaczmy, jak działa...

PRZYSZŁOŚĆ



KOSZYKOWANIE

Jeszcze lepsze dopasowanie do specyfiki zestawu danych - - wiele bardzo odległych outliersów, charakterystyczne tendencje dla różnych kategorii, itp.



PRECYZJA

Jeszcze lepsza optymalizacja parametrów, jeszcze większa trafność predykcji, jeszcze większa pewność odniesienia sukcesu.



INTERFEJS

Jeszcze piękniejsza, jeszcze szybsza, jeszcze bardziej przyjazna, jeszcze bardziej intuicyjna aplikacja.

TEAM { PANDP }

Dzięki!



NINA UTRATA



PAWEŁ DAWICKI



BARTOSZ STASIAK



PAWEŁ ALICKI



MATEUSZ ZAWADZKI