Politechnika Krakowska
im. Tadeusza Kościuszki

Wydział Informatyki i Telekomunikacji

Kajetan Mikrut

Numer albumu: 138098

Wykorzystanie metod sztucznej inteligencji do prognozowania cen nieruchomości.

The use of artificial intelligence methods in forecasting real estate prices.

Praca inżynierska
na kierunku Informatyka

Praca wykonana pod kierunkiem:
dr hab. inż. Michał Piotr Bereta , prof. PK

Kraków. 2024

SPIS TREŚCI

1. WSTĘP

Obecnie na rynku dostępne są różne rozwiązania kalkulujące ceny nieruchomości – zarówno takie płatne, jak i darmowe, niektóre korzystają z ludzkiej analizy, inne używają metod uczenia maszynowego/sztucznej inteligencji, zdarza się również spotykać mieszane metody. Nie są one jednak pozbawione wad – czasem konieczne jest czekanie 24 godziny na wynik szacunku, jak również niektóre z nich są wysoce kosztowne. Ponadto nie zawsze wiadomo, jak działają te serwisy – z jakich źródeł korzystają, na bazie czego wykonywane są kalkulacje, jakich algorytmów używają. Ostatecznie, niemal wszystkie z nich wymagają podania danych niepotrzebnych stricte do oceny wartości danej nieruchomości, takich jak e-mail, czy numer telefonu osoby zainteresowanej, jak również odpowiedzi na pytania związane z planami dotyczącymi nieruchomości.

Celem tej pracy jest stworzenie modelu do kalkulacji cen nieruchomości, pozbawionego wyżej wymienionych wad, będącego otwartym i darmowym narzędziem. Najważniejszym aspektem tworzenia takiego modelu jest odpowiedni algorytm kalkulujący ceny na bazie podanych przez użytkownika parametrów, takich jak lokalizacja, metraż, stan lokalu etc., zatem konieczne będzie przetestowanie różnych funkcji do tworzenia takich modeli, z różnymi parametrami, w celu wyłonienia algorytmu cechującego się największą dokładnością predykcji. Równie ważne jest pozyskanie odpowiednich danych do szkolenia danego modelu – bazy danych na temat ogłoszeń nieruchomości z ich lokalizacjami, parametrami i ceną.

2. CEL I ZAKRES PRACY

W celu stworzenia odpowiedniego modelu kalkulującego ceny nieruchomości należy pozyskać odpowiednią bazę danych z możliwie największą liczbą parametrów – tak, aby móc sprawdzić, jaka ich liczba jest odpowiednia do najlepszego, tj. najbardziej precyzyjnego przewidzenia wartości danego mieszkania. Proces pozyskania i opis bazy danych przedstawiony jest w rozdziale 3. Pozyskanie danych. Konieczne jest przetestowanie wykorzystania różnej liczby tych parametrów. Bardzo istotnym elementem jest analiza takiej bazy danych, zbadanie jej za pomocą wykresów czy map cieplnych w celu determinacji, które cechy mają najważniejszy wpływ na cenę nieruchomości, a które można pominąć, co opisane jest rozdziale 5. Przygotowanie środowiska i wczytanie danych – podrozdziały 5.2 i 5.3.

Ponadto należy zastanowić się nad odpowiednim wyborem środowiska, języka programowania używanego w projekcie i bibliotek sztucznej inteligencji – w tym przypadku jest to *Jupyter Notebook*[5], język *Python*[4] i biblioteka *scikit-learn*[6]. Trzeba przetestować różne algorytmy szkolące modele w celu wyłonienia jednego, cechującego się największą dokładnością predykcji, używając różnych parametrów przy ich tworzeniu. Konieczne jest także rozważenie różnych podejść w przypadku brakujących danych – nieuniknione jest natknięcie się na pewne braki w bazie danych, co stanowi problem dla wielu algorytmów uczących. Należy zatem rozważyć usunięcie takich rekordów kosztem utraty cennych zasobów, bądź wypełnienie brakujących danych na bazie całego zbioru. Tematyce tej poświęcony jest rozdział 6. Tworzenie i analiza modeli, w którym testowane są różne próby stworzenia modelu przewidującego ceny nieruchomości.

Po przeprowadzonych badaniach należy wyciągnąć wnioski i zastanowić się nad ewentualną kontynuacją prac nad projektem, możliwościami rozwoju bądź rozszerzeniem badań o alternatywne podejścia - poświęcone temu jest podsumowanie pracy.

3. POZYSKANIE DANYCH

Niewątpliwie pierwszym, podstawowym elementem stworzenia takiego programu jest pozyskanie bazy danych do wyszkolenia modelu sztucznej inteligencji. W tym celu najprościej byłoby pozyskać takie dane z któregoś z popularnych portali aukcyjnych, niestety żaden z nich nie pozwala na wykorzystywanie ich danych. Są jednak dostępne bazy archiwalne, udostępniane na różnych licencjach. Baza używana w tym projekcie pochodzi z serwisu Kaggle[1]. Jest ona udostępniona na licencji Apache 2.0[2], pozwalającej na użycie jej do tego typu projektu.

Baza ta zawiera informacje na temat ogłoszeń sprzedaży nieruchomości, konkretnie mieszkań w 15 największych polskich miast z sierpnia, września października i listopada 2023 roku. Poza parametrami pozyskanymi z samych ogłoszeń jest ona uzupełniona także o dane z serwisu Open Street Map[3] wskazujące dystans od danego lokalu do najbliższych interesujących miejsc z perspektywy osoby kupującej (ang. *points of interest*), takich jak szkoły, apteki, sklepy, restauracje itp. Każda tabela składa się z 28 kolumn, w tym 26 możliwych kategorii dla każdej pozycji:

1) *id* – unikalny identyfikator każdego ogłoszenia, wyrażony w ciągu znaków;
2) *city* – miasto, w którym znajduje się dany lokal;
3) *type* – typ budynku, w jakim znajduje się nieruchomość – *blockOfFlats, tenement,* lub *apartmentBuilding* (blok mieszkalny, kamienica lub apartamentowiec);
4) *squareMeters* – metraż mieszkania;
5) *rooms* – liczba pokoi w mieszkaniu;
6) *floor* – piętro, na którym znajduje się lokal;
7) *floorCount* – całkowita liczba pięter bloku/kamienicy;
8) *buildYear* – data wybudowania budynku, w którym znajduje się nieruchomość;
9) *latitude* – szerokość geograficzna lokalizacji mieszkania;
10) *longitude* – długość geograficzna lokalizacji mieszkania;
11) *centre distance* – odległość od centrum miasta w kilometrach;
12) *poiCount* – liczba punktów zainteresowania w promieniu 500 metrów od nieruchmości;
13) *[punkt_zainteresowania]Distance* – siedem kolumn wyrażających dystans w kilometrach od najbliższych punktów zainteresowania takich jak: szkoła, przychodnia, punkt pocztowy, przedszkole, restauracja, szkoła wyższa czy apteka;
14) *ownership* – typ własności lokalu – *condominium* lub *cooperative*;
15) *buildingMaterial* – materiał, z jakiego wykonany jest budynek – *brick* lub *concreteSlab* czyli cegła lub „wielka płyta”;
16) *condition* – stan mieszkania, *low* lub *premium*;
17) *has[właściwość]* – pięć kolumn opisujących dodatkowe cechy mieszkania – obecność miejsca parkingowego, balkonu, windy, ochrony i piwnicy/komórki lokatorskiej, określone wartościami *yes* lub *no*;
18) *price* – cena nieruchomości w ogłoszeniu.

Warto zauważyć, że nie każdy rekord w bazie danych ma uzupełnione wszystkie te pola – zdarzają się puste komórki. Na rysunkach 3.1, 3.2 i 3.3 widoczna struktura bazy danych dla pierwszych dziesięciu rekordów.

id	city	type	squareMeters	rooms	floor	floorCount	buildYear	latitude	longitude
f8524536d4b09a0c8ccc0197ec9d7bde	szczecin	blockOfFlats	63	3	4	10	1980	53.378933	14.6252957
accbe77d4b360fea9735f138a50608dd	szczecin	blockOfFlats	36	2	8	10		53.442692	14.5596899
8373aa373dbc3fe7ca3b7434166b8766	szczecin	tenement	73.02	3	2	3		53.452222	14.5533333
0a68cd14c44ec5140143ece75d739535	szczecin	tenement	87.6	3	2	3		53.4351	14.5329
f66320e153c2441edc0fe293b54c8aeb	szczecin	blockOfFlats	66	3	1	3		53.410278	14.5036111
2e190fcd6934978ca36d86ba41e842fc	szczecin	blockOfFlats	63.3	3	2	4	1997	53.4631	14.5728
ec27024bfcd012728617a35dad2cb6b8	szczecin	blockOfFlats	47.45	2	2	10	1974	53.450233	14.5626247
d3e0e36529df3360849ec40168c10755	szczecin	apartmentBuilding	60.08	2	3	4	2009	53.454685	14.5515204
7e1981e920d763d6237c5bdcf13cf5b7	szczecin	blockOfFlats	47.76	2	8	12	1980	53.458869	14.5364034
4a04a9c54d8281e3ec23df031e538d85	szczecin	tenement	72.09	4	2	3	1890	53.435092	14.5596122

Rys. 3.1. Struktura bazy danych – część pierwsza

centreDistance	poiCount	schoolDistance	clinicDistance	postOfficeDistance	kindergartenDistance	restaurantDistance
6.53	9	0.118	1.389	0.628	0.105	1.652
2.15	16	0.273	0.492	0.652	0.291	0.348
3.24	9	0.275	0.672	0.367	0.246	0.3
2.27	32	0.175	0.259	0.223	0.359	0.101
4.07	1	0.218	1.69	0.504	0.704	0.501
4.48	10	0.079	1.224	0.737	0.26	1.102
2.99	18	0.327	0.378	0.234	0.262	0.244
3.53	8	0.572	0.784	0.305	0.435	0.257
4.27	6	0.345	0.52	0.336	0.5	0.265
1.3	22	0.232	0.292	0.356	0.301	0.199

Rys. 3.2. Struktura bazy danych – część druga

collegeDistance	pharmacyDistance	ownership	buildingMaterial	condition	hasParkingSpace	hasBalcony	hasElevator	hasSecurity	hasStorageRoom	price
	0.413	condominium	concreteSlab		yes	yes	yes	no	yes	415000
1.404	0.205	cooperative	concreteSlab		no	yes	yes	no	yes	395995
1.857	0.28	condominium	brick		no	no	no	no	no	565000
0.31	0.087	condominium	brick		yes	yes	no	no	yes	640000
2.138	0.514	condominium			no	no	no	no	no	759000
0.377	0.745	cooperative	concreteSlab		yes	yes	no	no	yes	499000
1.736	0.277	condominium	concreteSlab	low	no	no	yes	no	yes	370000
1.945	0.155	condominium	brick	premium	no	yes	yes	no	no	629000
1.879	0.42	condominium	concreteSlab		no	yes	yes	no	yes	399000
0.653	0.199	condominium	brick	low	yes	no	no	no	yes	325000

Rys. 3.3. Struktura bazy danych – część trzecia

Baza zawiera około 19 000 rekordów dla każdego miesiąca, co jest stosunkowo niewielką liczbą w kontekście szkolenia modeli sztucznej inteligencji, jednak powinna ona być wystarczająca dla badań przeprowadzanych w tym projekcie. Jej niewątpliwą zaletą jest bardzo duża liczba parametrów nieruchomości, dzięki czemu możliwe jest przetestowanie wielu opcji szkolenia modelu, zbadanie jaka liczba parametrów jest najlepsza dla stworzenia algorytmu cechującego się największą dokładnością predykcji.

4. WYKORZYSTANE TECHNOLOGIE

Kolejnym istotnym krokiem jest wybór technologii do przetwarzania pozyskanych danych i stworzenia modelu. Podczas wyboru technologii istotnymi czynnikami są oczywiście oferowane możliwości, lecz także szybkość działania i dostęp do dokumentacji oraz stabilność.

4.1 Python

Jako język używany do pisania projektu został wybrany Python w wersji 3.11.5. Język ten cechuje się prostotą składni, co wysoce ułatwia pisanie kodu, a jednocześnie oferuje wsparcie dla wielu narzędzi przydatnych do analizy i przetwarzania danych, jak również do sztucznej inteligencji/uczenia maszynowego. Wersja 3.11.5 z uwagi na to, że jest to stabilna, przetestowana wersja, z dostępną dokumentacją, która jednocześnie oferuje wsparcie dla wszystkich bibliotek używanych w projekcie. Nie jest to wszakże najnowsza wersja, ta jednak nie jest kompatybilna z niektórymi narzędziami, które wykorzystane są w procesie tworzenia modelu.

4.2 Jupyter Notebook

Środowisko zastosowane do stworzenia projektu. Notatniki Jupyter są interaktywne i iteratywne – mogą być używane krok po kroku, co jest niezwykle ważne biorąc pod uwagę eksploracyjny charakter projektu – dzięki temu można w szybki sposób testować różne kombinacje modeli bez konieczności uruchamiania całego skryptu na nowo. Ponadto umożliwiają one łączenie kodu, tekstu i wyników w jednym dokumencie, co znacząco upraszcza proces analizy danych, zrozumienia modeli i interpretacji wyników bez konieczności korzystania z wielu plików. Mogą być używane z wieloma językami (wsparcie dla ponad 30 języków), w tym z Pythonem.

4.3 Scikit-learn

Jest to popularna biblioteka do uczenia maszynowego w języku Python. Jest otwarto-źródłowa, zawiera narzędzia i algorytmy do przetwarzania i analizy danych oraz do uczenia maszynowego. Posiada szeroką społeczność programistyczną, dzięki czemu jest na bieżąco aktualizowana, wszelkie błędy są na bieżąco usuwane, a ponadto dostępna jest obszerna dokumentacja. Oferuje ona szeroką gamę algorytmów uczenia maszynowego, które są intuicyjne w użyciu, z bardzo dobrym opisem błędów. Dodatkowo jest dobrze zintegrowana z innymi bibliotekami, takimi jak *Pandas*[7], *Matplotlib*[8], czy *Seaborn*[9] co ułatwia pracę z danymi.

4.4 Pandas

Biblioteka w języku Python przeznaczona do manipulacji i analizy danych. Jest bardzo użytecznym narzędziem do pracy z danymi takimi jak arkusze kalkulacyjne czy bazy danych. Wprowadza strukturę danych *DataFrame* zbliżoną do tabeli, dostarcza także wiele wbudowanych funkcji i narzędzi do przetwarzania danych, oferuje także wiele metod wizualizacji danych. Ponadto również dobrze integruje się z pozostałymi bibliotekami wykorzystywanymi w projekcie.

4.5 Matplotlib

Biblioteka do wizualizacji danych w języku Python. Umożliwia tworzenie różnego rodzaju wykresów oraz posiada wszechstronny interfejs pozwalający użytkownikowi na kontrolę nad wyglądem tworzonych wizualizacji. W tym projekcie będzie używana głównie do konfiguracji wykresów – nadawania etykiet oraz kontroli nad rozmiarami i proporcjami wykresów.

4.6 Seaborn

Biblioteka w języku Python oparta na *Matplotlib*, dostarcza wyższego poziomu interfejsu do wizualizacji danych – umożliwia tworzenie zaawansowanych wykresów statystycznych, co jest szczególnie pomocne w fazie eksploracji danych.

5. PRZYGOTOWANIE ŚRODOWISKA I WCZYTANIE DANYCH

Pierwsze, podstawowe czynności przy rozpoczęciu projektu, to przygotowanie środowiska, w tym przypadku jest to notatnik Jupyter, który nie wymaga zaawansowanej konfiguracji, następnie należy wczytać dane i odpowiednio je przygotować do dalszej pracy. Ważnym aspektem jest także analiza danych, jak podaje Baldominos A. szczególnie istotne jest zrozumienie zależności między poszczególnymi parametrami a ostateczną ceną nieruchomości, gdyż to one mogą mieć największy wpływ na kształtowanie modelu[12].

5.1 Przygotowanie środowiska

Przy tworzeniu nowego notatnika Jupyter konieczny jest jedynie wybór jądra – domyślnie dostępne są *Calysto Scheme 3* oraz *Python 3 (ipykernel)*, jądra można także dodawać, jednak na potrzeby tego projektu wystarczy *Python 3*. Kolejnym krokiem jest zainstalowanie i import wszelkich potrzebnych bibliotek/klas. Większość bibliotek była domyślnie zainstalowana razem z środowiskiem *Python*, zainstalowania wymagała jedynie biblioteka *Seaborn*, co też można zrobić poleceniem *pip install*, następnie konieczne było także dodanie ścieżki do miejsca jej instalacji do zmiennej *$PATH*, aby później notatnik mógł ją zaimportować. Teraz można przejść do importu wszystkich potrzebnych bibliotek i klas, co przedstawione jest na listingu 5.1.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
import mpl_toolkits
from sklearn import ensemble
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
```

Listing 5.1. Import bibliotek i klas do projektu

Większość często używanych bibliotek importowana jest z aliasem, co pozwala na skrócenie kodu. W przypadku biblioteki *scikit-learn* pobrane są konkretne klasy, a nie cała biblioteka. Jest to również działanie mające na celu uproszczenie kodu – zamiast pisać *sklearn.preprocessing.LabelEncoder* można użyć samego *LabelEncoder*, co przy wielokrotnym użyciu poszczególnych klas niewątpliwie przyspiesza proces i wzmaga czytelność kodu.

Kolejnym krokiem w celu utrzymania spójności badań jest upewnienie się, że wszelkie wykresy będą wyświetlać się w notatniku bezpośrednio pod komórkami z kodem, który je generuje, a nie w osobnych plikach. Dzięki temu zależnie od potrzeby możliwy będzie powrót i ponowna analiza takich wykresów, a całość będzie spójna. W tym celu użyta jest funkcja magiczna *%matplotlib*, co zamieszczono na listingu 5.2.

```
%matplotlib inline
```

Listing 5.2. Użycie funkcji *%matplotlib*

Funkcja ta jako argument przyjmuje backend do wyświetlania wykresów, jeśli użyje się jej zaś bez parametrów wszelkie rysunki będą wyświetlać się w osobnym okienku (domyślny backend *matplotlib*). Użycie argumentu *inline* dostępne jest jedynie dla notatników Jupyter i *Jupyter QtConsole*, pozwala on na wyświetlanie wszelkich wykresów bezpośrednio pod komórką z kodem, który je tworzy.

Z racji tego, że tabele, które będą wczytywane do projektu mają dużą liczbę kolumn należy także zmienić ustawienia biblioteki *Pandas* tak, aby przy wyświetlaniu danych widoczna była cała szerokość tabeli – niekiedy może to być kluczowe w celu odczytania konkretnych parametrów.

```
pd.set_option('display.max_columns', 30)
```

Listing 5.3. Ustawnie maksymalnej liczby wyświetlanych kolumn

Konieczna jest zatem modyfikacja zmiennej *display.max_columns* – domyślnie jest ona ustawiona na 20, a tabele używane w projekcie mają po 28 kolumn, wobec czego wartość została ustawiona na 30 za pomocą funkcji *set_option*, jak widać na listingu 5.3, aby zostawić zapas na ewentualne dodane kolumny. Z tak skonfigurowanym środowiskiem można przejść do kolejnego etapu.

5.2 Wczytanie, wstępna analiza i konfiguracja danych

Po pomyślnym skonfigurowaniu środowiska można przystąpić do wczytania i wstępnej konfiguracja danych. Z racji tego, że będą testowane różne podejścia, dane będą na późniejszych etapach zmieniane na różne sposoby, niemniej już na początku można pozbyć elementów, które się nie przydadzą na żadnym stadium, jak również dodać pewne stałe. Pliki są w formacie csv, wczytywane do struktury *DataFrame* z biblioteki *Pandas* za pomocą funkcji *read_csv*, co ukazano na listingu 5.4.

```
data_august = pd.read_csv("apartments_pl_2023_08.csv", true_values=['yes'], false_values=['no'])
data_september = pd.read_csv("apartments_pl_2023_09.csv", true_values=['yes'], false_values=['no'])
data_october = pd.read_csv("apartments_pl_2023_10.csv", true_values=['yes'], false_values=['no'])
data_november = pd.read_csv("apartments_pl_2023_11.csv", true_values=['yes'], false_values=['no'])
```

Listing 5.4. Wczytanie danych

Jako pierwszy argument podaje się ścieżkę do pliku, będącą w tym przypadku jednocześnie nazwą pliku, z racji tego, że notatnik znajduje się w tym samym folderze co baza danych. Ustawione są tutaj także parametry *true_values* i *false_values*, które zamieniają wartości przypisane do nich na zmienne typu *Boolean* – *True* i *False*. W plikach znajdują się kolumny *has[właściwość]*, w których znajdują się wartości „*yes*” lub „*no*”, będące wartościami tekstowymi, dla środowiska odczytywane jako *string*. Funkcje szkolące modele używane w tym projekcie przyjmują jedynie wartości liczbowe, wobec czego taka zamiana jest konieczna.

Kolejnym krokiem jest dodanie do każdej z tabel pola na miesiąc, z którego pochodzą dane – początkowo dane były w różnych plikach, wobec czego nie było takiej kolumny w tabelach, jednak chcąc połączyć je w jeden plik konieczne jest dodanie

takiego pola z uwagi na to, że ceny nieruchomości mogą się zmieniać z miesiąca na miesiąc, wobec czego może być to ważny parametr dla modelu przewidującego ich ceny. Wykonanie widoczne jest na listingu 5.5.

```
data_august['month'] = 8
data_september['month'] = 9
data_october['month'] = 10
data_november['month'] = 11
```

Listing 5.5. Dodanie kolumny z numerem miesiąca, z którego pochodzą dane

Tak przygotowane dane można złączyć w jeden plik, celem posiadania jak największej liczby rekordów do szkolenia modelu. W tym celu użyta jest funkcja *concat*, zamieszczona na listingu 5.6.

```
data = pd.concat([data_august, data_september, data_october, data_november], ignore_index = True)
```

Listing 5.6. Złączenie danych

Jako parametry podana została lista obiektów do złączenia oraz *ignore_index* ustawione na *True*. Dzięki temu obecne indeksy zostaną zignorowane, nadana będzie nowa numeracja, wszystkie dane przypisane są do zmiennej *data*. Aby zobaczyć, jak wygląda struktura zaimportowanych i połączonych danych można użyć funkcji *head* wyświetlającej domyślnie 5 pierwszych rekordów.

	id	city	type	squareMeters	rooms	floor	floorCount	buildYear	latitude	longitude	centreDistance	poiCount
0	f8524536d4b09a0c8ccc0197ec9d7bde	szczecin	blockOfFlats	63.00	3.0	4.0	10.0	1980.0	53.378933	14.625296	6.53	9.0
1	accbe77d4b360fea9735f138a50608dd	szczecin	blockOfFlats	36.00	2.0	8.0	10.0	NaN	53.442692	14.559690	2.15	16.0
2	8373aa373dbc3fe7ca3b7434166b8766	szczecin	tenement	73.02	3.0	2.0	3.0	NaN	53.452222	14.553333	3.24	9.0
3	0a68cd14c44ec5140143ece75d739535	szczecin	tenement	87.60	3.0	2.0	3.0	NaN	53.435100	14.532900	2.27	32.0
4	f66320e153c2441edc0fe293b54c8aeb	szczecin	blockOfFlats	66.00	3.0	1.0	3.0	NaN	53.410278	14.503611	4.07	1.0

Rys. 5.1. Użycie funkcji *head* – część pierwsza

schoolDistance	clinicDistance	postOfficeDistance	kindergartenDistance	restaurantDistance	collegeDistance	pharmacyDistance	ownership	buildingMaterial
0.118	1.389	0.628	0.105	1.652	NaN	0.413	condominium	concreteSlab
0.273	0.492	0.652	0.291	0.348	1.404	0.205	cooperative	concreteSlab
0.275	0.672	0.367	0.246	0.300	1.857	0.280	condominium	brick
0.175	0.259	0.223	0.359	0.101	0.310	0.087	condominium	brick
0.218	1.690	0.504	0.704	0.501	2.138	0.514	condominium	NaN

Rys. 5.2. Użycie funkcji *head* – część druga

condition	hasParkingSpace	hasBalcony	hasElevator	hasSecurity	hasStorageRoom	price	month
NaN	True	True	True	False	True	415000	8
NaN	False	True	True	False	True	395995	8
NaN	False	False	False	False	False	565000	8
NaN	True	True	False	False	True	640000	8
NaN	False	False	False	False	False	759000	8

Rys. 5.3. Użycie funkcji *head* – część trzecia

Po analizie rysunków 5.1 - 5.3 można uznać, że import przebiegł poprawnie, wszystkie dane wyświetlają się w należyty sposób, jak również widać dodaną kolumnę *month*. Można także zauważyć, że w pustych komórkach wyświetla się *NaN* (eng. *Not a Number*) – znacznik wskazujący, że wartość znajdująca się w tym miejscu nie ma wizualnej reprezentacji. Jest to problem, do którego trzeba będzie powrócić w późniejszym momencie projektu, z uwagi na to, że większość funkcji budujących modele nie radzi sobie z brakującymi wartościami, jednak istnieje parę strategii z nimi związanych, co też zostanie przetestowane. W celu dalszej analizy danych można użyć funkcji *describe* – pokaże ona podstawową analizę zawartości kolumn.

	squareMeters	rooms	floor	floorCount	buildYear	latitude	longitude	centreDistance	poiCount	schoolDistance
count	68894.000000	68894.000000	56422.000000	67938.000000	56959.000000	68894.000000	68894.000000	68894.000000	68894.000000	68825.000000
mean	58.944435	2.685212	3.305431	5.221172	1984.586527	52.045462	19.501237	4.342607	20.337460	0.416781
std	21.279593	0.911641	2.506487	3.270862	34.058699	1.339131	1.781489	2.873857	23.816947	0.479851
min	25.000000	1.000000	1.000000	1.000000	1850.000000	49.978999	14.447127	0.020000	0.000000	0.004000
25%	44.492500	2.000000	2.000000	3.000000	1965.000000	51.114026	18.523780	1.990000	7.000000	0.175000
50%	55.000000	3.000000	3.000000	4.000000	1991.000000	52.195312	19.899315	3.940000	13.000000	0.290000
75%	69.000000	3.000000	4.000000	6.000000	2014.000000	52.440563	20.994734	6.120000	24.000000	0.468000
max	150.000000	6.000000	29.000000	29.000000	2023.000000	54.606460	23.207128	16.940000	208.000000	4.818000

Rys. 5.4. Użycie funkcji *describe* – część pierwsza

clinicDistance	postOfficeDistance	kindergartenDistance	restaurantDistance	collegeDistance	pharmacyDistance	price	month
68575.000000	68799.000000	68826.000000	68701.000000	66873.000000	68787.000000	6.889400e+04	68894.000000
0.983966	0.523196	0.376271	0.357266	1.449895	0.366447	7.151848e+05	9.441098
0.905907	0.513988	0.463293	0.484109	1.106158	0.478425	3.515693e+05	1.126327
0.001000	0.001000	0.002000	0.001000	0.006000	0.001000	1.500000e+05	8.000000
0.359000	0.239000	0.158000	0.117000	0.584000	0.145000	4.700000e+05	8.000000
0.681000	0.392000	0.266000	0.234000	1.121000	0.241000	6.490000e+05	9.000000
1.255000	0.628000	0.421000	0.416000	2.070000	0.407000	8.600000e+05	10.000000
4.998000	4.968000	4.960000	4.985000	5.000000	4.992000	2.500000e+06	11.000000

Rys. 5.5. Użycie funkcji *describe* – część druga

Funkcja ta domyślnie wyświetla 8 kategorii dla każdej numerycznej kolumny w obiekcie *DataFrame*:

1) *count* – liczba zapełnionych komórek w danej kolumnie;
2) *mean* – średnia wartość danych z danej kolumny;
3) *std* – odchylenie standardowe danych;
4) *min, max* – skrajne wartości występujące w określonej kolumnie;
5) *25%, 50%, 75%* - dane w konkretnych percentylach.

Dzięki analizie rysunków 5.4 i 5.5 można zaobserwować pewne anomalie w bazie danych, jak na przykład fakt, że 75% nieruchomości ma metraż do 69m^2, zaś największa ma aż 150m^2 – takie odchylenia mogą negatywnie wpłynąć na uczenie i późniejszą efektywność modelu, wobec czego na razie dane te pozostaną w bazie, jednak w przypadku negatywnych wyników będzie się można zastanowić nad usunięciem danych odbiegających od normy. Ponadto obserwując wartości *count* dla każdej kolumny widać, że niektóre kolumny posiadają dużo pustych rekordów – w szczególności kolumny *floor* i *buildYear* – cała struktura posiada obecnie 68894 rekordy, zaś te kolumny posiadają odpowiednio 56422 i 56959 wypełnionych pól, czyli około 19% pól jest pustych. Stanowi

to duży problem szczególnie w przypadku kolumny *floor* określającej piętro, na którym znajduje się nieruchomość z uwagi na to, że z pewnością jest to wartość mająca wysoki wpływ na ostateczną cenę nieruchomości, w związku z czym nie można usunąć całej tej kolumny, zaś pozbycie się rekordów z pustymi wartościami wiąże się z utratą co najmniej około 20% wierszy z bazy.

Aby wyświetlić analizę kolumn zawierających wartości nienumeryczne (w tym przypadku są to wyłącznie wartości tekstowe) należy ponownie użyć funkcji *describe*, tym razem jednak ustawiając parametr *include* na wartość *object* – wskazuje on, aby funkcja wyświetliła wartości typu *object*, czyli na przykład wartości tekstowe (lecz nie tylko, biblioteka *Pandas* określa w ten sposób także wartości typu *timestamp* i inne).

	id	city	type	ownership	buildingMaterial	condition	hasElevator
count	68894	68894	53616	68894	42103	16970	65349
unique	39028	15	3	3	2	2	2
top	98bd9e22e76cf1b940267b08127c69be	warszawa	blockOfFlats	condominium	brick	premium	False
freq	4	20256	32104	61814	32519	9351	34587

Rys. 5.6. Użycie funkcji *describe* dla wartości typu *object*

Przy takim ustawieniu funkcja wyświetla 4 parametry dla każdej kolumny:

1) *count* – licznik niepustych pól;
2) *unique* – liczba unikalnych wartości w danej kolumnie;
3) *top* – najpopularniejsza wartość w konkretnej kolumnie;
4) *freq* – liczba wystąpień najpopularniejszej wartości.

Obserwując rysunek 5.6 można zauważyć, że w bazie znajdują się ogłoszenia dla 15 różnych miast, z których Warszawa pojawia się najczęściej – aż 20256 razy, jak również ponownie widać dwie problematyczne kolumny pod względem liczby pustych pól – *buildingMaterial* i *Condition*. Reszta kolumn ma po dwie, lub trzy wartości unikalne, wszystkie z nich trzeba będzie zamienić na wartości liczbowe przed szkoleniem modelu, gdyż funkcje szkolące nie przyjmują wartości tekstowych. Kolumnę „id" można zaś usunąć – są to jedynie identyfikatory ogłoszeń, nie mające żadnego wpływu na ceny nieruchomości. Widać natomiast, że identyfikatory się powtarzają, co oznacza, że niektóre z nieruchomości były widniały przez ponad miesiąc na portalach aukcyjnych. Nie powinno mieć to negatywnego wpływu na szkolenie, jednak w razie niepowodzenia będzie można rozważyć usunięcie powtarzających się rekordów.

```
data = data.drop(['id'], axis=1)
```

Listing 5.7. Usunięcie kolumny *id* z wykorzystaniem funkcji *drop*

Do usunięcia kolumny wykorzystano funkcję *drop*, jak widać na listingu 5.7, w której pierwszy parametr wskazuje na usuwaną kolumnę, zaś jako drugi argument podano *axis*, czyli orientację w jakiej użytkownik chce usunąć dane – 1 oznacza usunięcie w pionie, czyli kolumnę, zaś 0 – w poziomie, czyli wiersz.

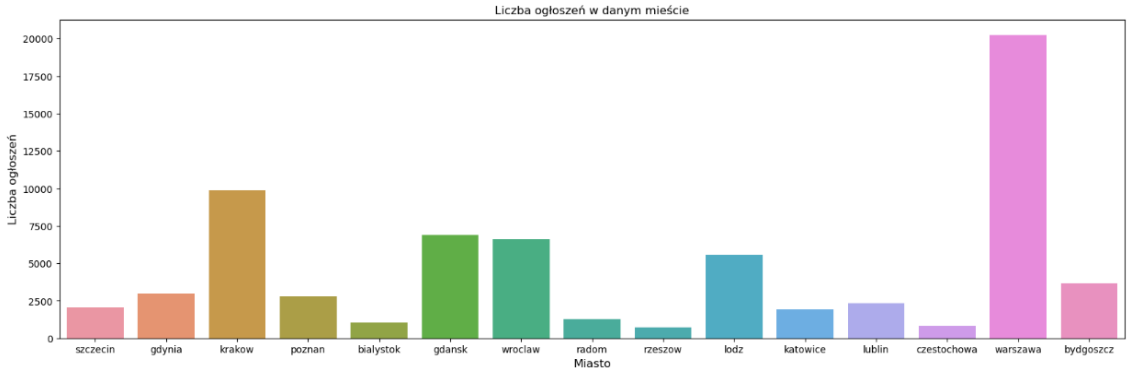
5.3 Analiza konkretnych kolumn bazy danych w poszukiwaniu anomalii

Biblioteka *Pandas* dostarcza wiele innych funkcji do analizy struktury *DataFrame*, jednak wykonana dotychczas analiza wstępna jest wystarczająca, pozwala na zrozumienie ogólnej struktury bazy oraz na zauważenie potencjalnych problemów. Można zatem skupić się na analizie konkretnych cech bazy danych. Pierwszym etapem będzie analiza liczby ogłoszeń w różnych miastach, z uwagi na obserwację jak przeważająca liczba rekordów jest z Warszawy. Można zatem stworzyć wykres obrazujący tę zależność. W tym celu można wykorzystać bibliotekę *Seaborn* do stworzenia wykresu oraz *Matplotlib* do ustawienia wielkości obszaru rysowania i ustawienia tytułów osi, co przedstawiono na listingu 5.8.

```
plt.figure(figsize=(20,6))
sns.countplot(x = 'city', data = data).set_title("Liczba ogłoszeń w danym mieście")
plt.xlabel("Miasto", fontsize=12)
plt.ylabel("Liczba ogłoszeń", fontsize=12)
plt.show()
```

Listing 5.8. Stworzenie wykresu liczby ogłoszeń w danym mieście

W celu stworzenia wykresu używana jest funkcja *figure* do stworzenia przestrzeni na wykres i określenia jej wymiarów, następnie za pomocą funkcji *countplot* określane jest źródło danych i kolumna, z której dane będą znajdować się na osi *x*. Następnie ustawiane są tytuły osi, a funkcja *show* powoduje wyświetlenie wykresu.



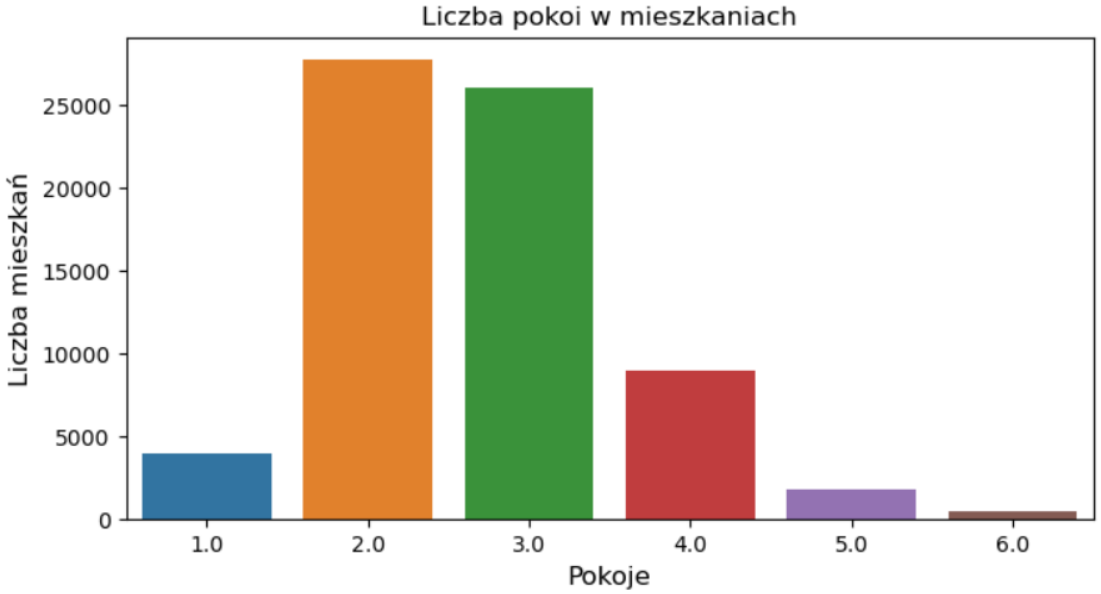
Rys. 5.7. Wykres liczby ogłoszeń w danym mieście

Z rysunku 5.7 można wynieść następujące wnioski:

1) Przeważająca liczba ogłoszeń jest z Warszawy, Krakowa, Gdańska, Wrocławia i Łodzi;
2) Bardzo niewiele ogłoszeń jest z Radomia, Rzeszowa, Częstochowy i Białegostoku.

W szczególności ważne są miasta, z których pochodzi niewiele ogłoszeń, z racji tego, że mogą zaburzać proces uczenia modelu – jak wiadomo, w każdym mieście ceny nieruchomości kształtują się w inny sposób, a zatem niewielka liczba rekordów z odbiegającymi parametrami może mieć negatywny wpływ na późniejszą efektywność

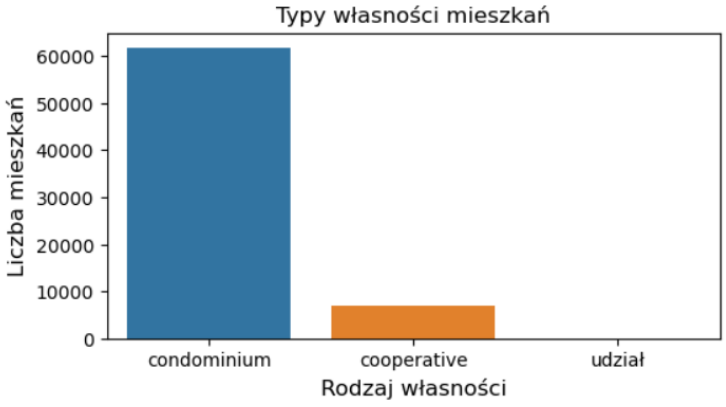
modelu. Kolejnym krokiem w procesie szukania anomalii jest zbadanie liczby pokoi w mieszkaniach – w tym celu można stworzyć kolejny wykres w procesie zbliżonym do poprzedniego.



Rys. 5.8. Wykres liczby pokoi w mieszkaniach

Proces tworzenia wykresu jest analogiczny do poprzedniego. Obserwując rysunek 5.8 można stwierdzić, że przeważają mieszkania dwu- i trzypokojowe, natomiast mieszkania sześciopokojowe są skrajnością, mogącą zaburzać stabilność danych.

Kolejną kolumną wymagającą zbadania jest *ownership*. Jak widać z rysunku 5.6 niemal 62 tysiące wartości to *condominium*, wobec czego należy sprawdzić jaki jest rozkład pozostałych wartości.




```
ownership
condominium    61814
cooperative     7077
udział             3
Name: count, dtype: int64
```

Rys. 5.9. Wykres i liczby wystąpień poszczególnych rodzajów własności mieszkań

Poza stworzeniem wykresu, jak widać na rysunku 5.9 zostały także wypisane konkretne liczby wystąpień poszczególnych rodzajów własności za pomocą funkcji *value_counts* wypisującej wystąpienia unikalnych wartości w kolumnach. Można zauważyć, że występują jedynie 3 rekordy o wartości *„udział”* – wygląda to jak błąd, niewątpliwie należy usunąć te wiersze. Wystąpień wartości *„cooperative”* jest około 7 tysięcy, co również jest stosunkowo niską liczbą w porównaniu do ponad 60 tysięcy wystąpień wartości *condominium*, niemniej nie jest to na tyle mała liczba, aby od razu usuwać te wartości.

```
data = data.drop(data[data['ownership'] == 'udział'].index)
```

Listing 5.9. Usunięcie nieprawidłowych wartości funkcją *drop*

Wartości zostały usunięte za pomocą funkcji *drop*, co ukazano na listingu 5.9. Nie usuwa ona tych wartości bezpośrednio, wobec czego należy zmodyfikowaną wersję przypisać do oryginalnego zbioru *data*.

Ostatnią kolumną do rozpatrzenia jest *condition*, gdyż jak wynika z rysunku 5.6 jedynie niecałe 17 tysięcy rekordów ma w tej kategorii przypisaną wartość. Przyjmuje ona 2 wartości: *premium* i *low*.

```
data['condition'].value_counts()
```

```
condition
premium    9351
low        7616
Name: count, dtype: int64
```

Listing 5.10. Wypisanie rozkładu wartości w kolumnie *condition*

Sam rozkład wartości widoczny na listingu 5.10 nie budzi podejrzeń, natomiast można sprawdzić, jaki wpływ mają te wartości na ostateczną cenę nieruchomości, aby ocenić na ile kluczowa jest to kolumna w kontekście całego zbioru. W tym celu należy wypełnić wartości *NaN*, aby móc je również zobaczyć na wykresie, co zaprezentowano na listingu 5.11.

```
data['condition'] = data['condition'].fillna('unsigned')
```

Listing 5.11. Wypełnienie pustych wartości kolumny *condition*

W tym celu można użyć funkcji *fillna*, która zapełnia wszystkie pola o wartości *NaN* zmienną podaną w argumencie. Z punktu widzenia bazy danych nie ma to znaczenia, gdyż dalej we wszystkich tych polach znajduje się identyczna wartość, zmienia się jedynie jej treść tak, aby można ją było bez przeszkód ukazać graficznie. Mając tak przygotowane dane można stworzyć wykres zależności ceny od parametru *condition*.

```
plt.figure(figsize=(10, 3))
sns.boxplot(x='price', y='condition', data=data)
plt.title('Zależność ceny od stanu mieszkania')
plt.xlabel('Cena', fontsize=12)
plt.ylabel('Stan mieszkania', fontsize=12)
plt.show()
```

Listing 5.12. Stworzenie wykresu zależności ceny od stanu mieszkania

Rys. 5.10. Wykres zależności ceny od stanu mieszkania

Jest to wykres pudełkowy, najlepiej ilustrujący tę zależność. Tworzy się go funkcją *boxplot* z biblioteki *Seaborn*, co zaprezentowano na listingu 5.12. Z rysunku 5.10 wynika, że choć faktycznie nieruchomości o niskim stanie mają delikatnie niższą cenę, a te o stanie premium – wyższą wartość, to oba te przedziały się pokrywają, mediany różnią się tylko o około 150 tysięcy złotych, co w przedziale od 150 tysięcy do 2,5 miliona złotych nie jest przeważającą wartością, zaś oferty o nieprzypisanym parametrze *condition* znajdują się najczęściej pomiędzy *low* i *premium*, w każdej kategorii jednak wartości skrajne sięgają niemal 2,5 mln złotych. Wobec tego po sprawdzeniu, jaka jest wydajność modelu z obecnością tych danych, będzie można sprawdzić jak model radziłby sobie bez tej kolumny, czy faktycznie jest ona niezbędna do wyceny nieruchomości.

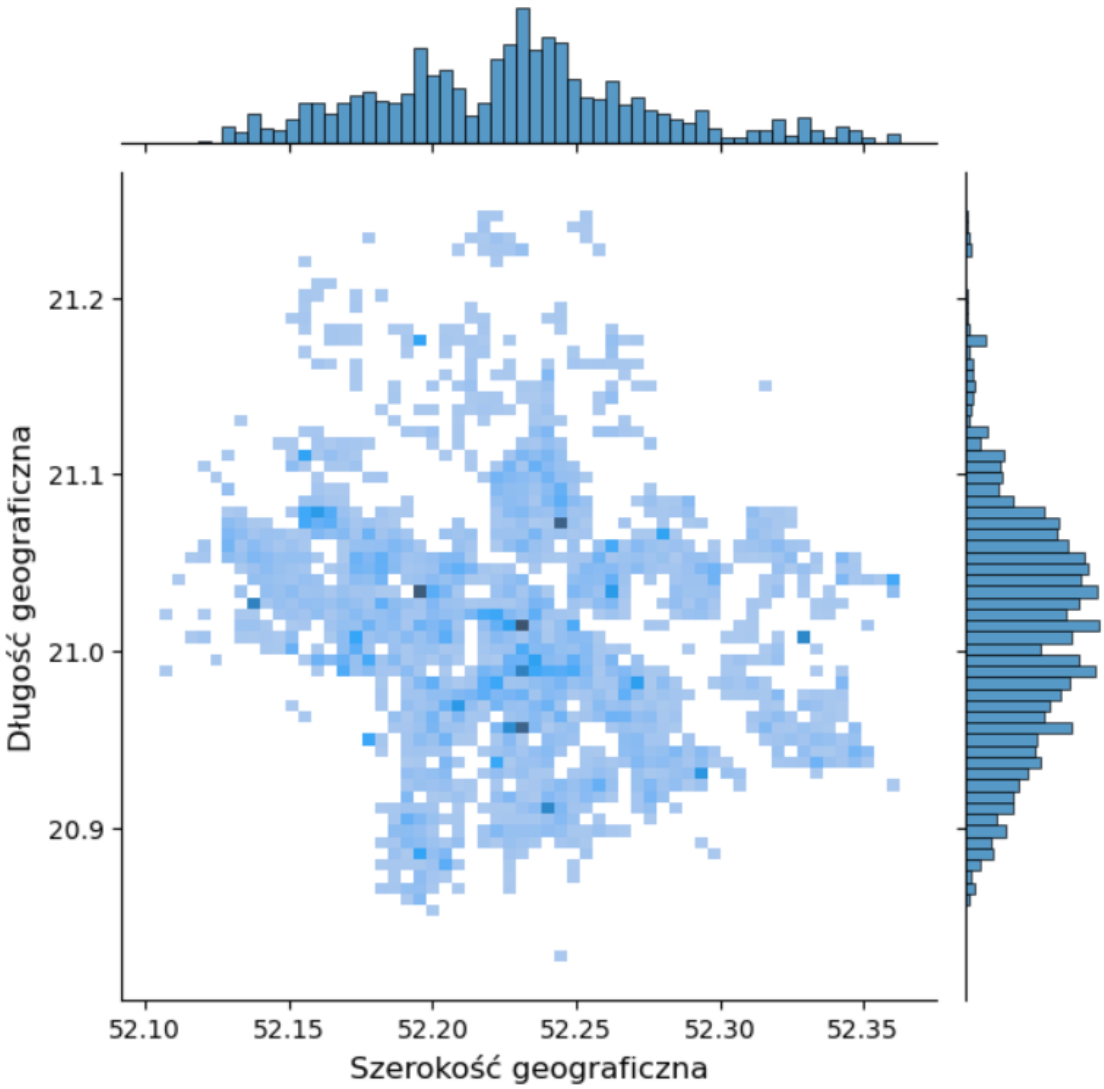
Po zbadaniu anomalii można przejść do stworzenia wykresów rozmieszczenia nieruchomości w różnych miastach używając szerokości i długości geograficznych, aby stwierdzić, czy nieruchomości są równomiernie rozłożone w miastach, czy są one odpowiednią reprezentacją do wyszkolenia modelu. Można by stworzyć jeden wykres dla całej bazy, jednak baza obejmuje 15 miast z całej Polski, wobec czego trudno byłoby cokolwiek zobaczyć na takim wykresie. Proces tworzenia wykresu ukazano na listingu 5.13.

```
warsaw_data = data[data['city'] == 'warszawa']
```

```
plt.figure(figsize=(6,6))
sns.jointplot(x=warsaw_data.latitude.values, y=warsaw_data.longitude.values, kind = 'hist')
plt.ylabel('Długość geograficzna', fontsize=12)
plt.xlabel('Szerokość geograficzna', fontsize=12)
plt.show()
```

Listing 5.13. Kod do wygenerowania wykresu rozkładu nieruchomości w Warszawie

W celu stworzenia wykresu należy wybrać rekordy z jednego miasta, następnie funkcją *jointplot* stworzyć wykres, jako argumenty podając wartości z kolumn *latitude* i *longitude*. Ustawiając wartość zmiennej *kind* można wybrać typ wykresu, w tym przypadku jest to histogram.



Rys. 5.11. Rozmieszczenie nieruchomości w Warszawie

W centralnej części wykresu widocznego na rysunku 5.11 można zobaczyć rozmieszczenie nieruchomości w Warszawie, kolorystycznie zaznaczone jest ich zagęszczenie na danym obszarze. U góry i po prawej stronie wykresu widoczna jest liczba

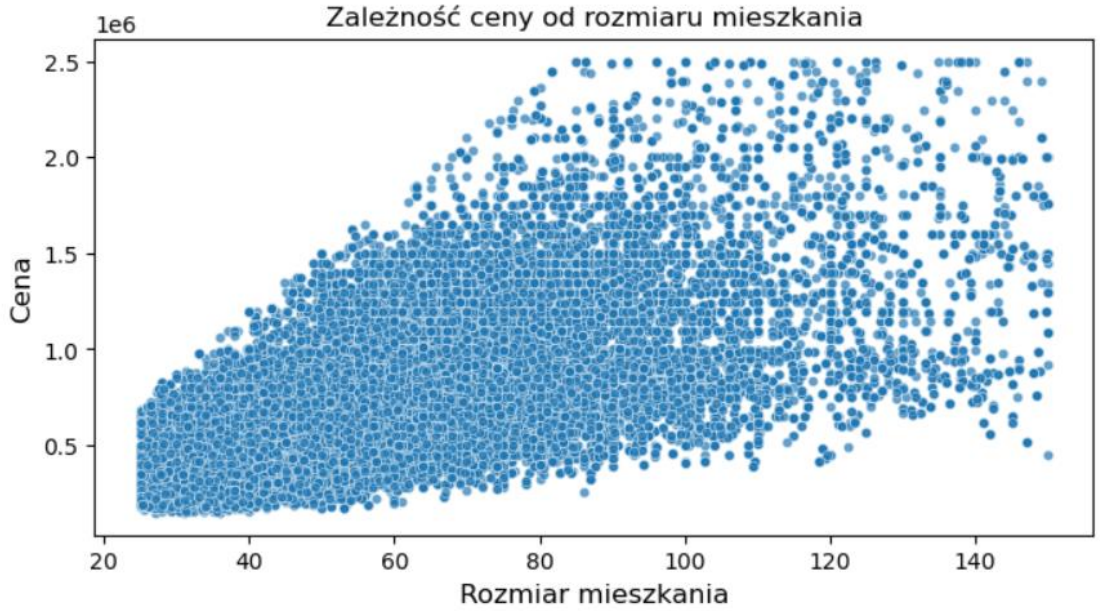
rekordów w danym pasie długości i szerokości geograficznych. Po analizie wykresu można stwierdzić, że nieruchomości są rozmieszczone odpowiednio aby stanowić reprezentację modelu. Podobna analiza została wykonana dla wszystkich miast, aby uniknąć późniejszych błędów.

Ostatnim elementem do analizy jest zależność pomiędzy poszczególnymi parametrami mieszkań, a ich ostateczną ceną. Te obserwacje pomogą na późniejszym etapie dobrać odpowiedni algorytm do szkolenia modelu. Pierwszym parametrem może być metraż – przy nim jest największa szansa dostrzeżenia zależności.

```
plt.figure(figsize=(8, 4))
sns.scatterplot(x='squareMeters', y='price', data=data, s=20, alpha=0.7)
plt.title('Zależność ceny od rozmiaru mieszkania')
plt.xlabel('Rozmiar mieszkania', fontsize=12)
plt.ylabel('Cena', fontsize=12)
plt.show()
```

Listing 5.14. Stworzenie wykresu zależności ceny od rozmiaru mieszkania

Taki wykres można wygenerować funkcją *scatterplot* z biblioteki *Seaborn*, co zaprezentowano na listingu 5.14. Ustawienie parametrów *s* i *alpha* zmienia rozmiar i przezroczystość punktów na wykresie, wspomagając jego czytelność.



Rys. 5.12. Wykres zależności ceny od rozmiaru mieszkania

Przy analizie rysunku 5.12 można zauważyć liniową zależność między metrażem, a ceną mieszkania. Nie jest to oczywiście w pełni liniowa zależność – jest zbyt dużo innych parametrów, aby tak było, natomiast widać podobieństwo do takiej zależności. W celu dalszego zbadania tej zależności policzony został współczynnik korelacji liniowej Pearsona między ceną i rozmiarem mieszkania, co widać na listingu 5.15.

```
correlation_coefficient, p_value = pearsonr(data['squareMeters'], data['price'])
print(f"Współczynnik korelacji Pearsona: {correlation_coefficient}")
print(f"P-wartość: {p_value}")
```

```
Współczynnik korelacji Pearsona: 0.6225598142744349
P-wartość: 0.0
```

Listing 5.15. Współczynnik Pearsona między ceną a rozmiarem mieszkania

Współczynnik ten można policzyć funkcją *pearsonr* z biblioteki *pyplot[10]*, która poza podaniem współczynnika liczy także prawdopodobieństwo, że dane nie są ze sobą powiązane. Otrzymane wyniki jasno wskazują na silną korelację pomiędzy tymi dwoma parametrami (wartość współczynnika korelacji między 0,5 a 0,7 jest najczęściej traktowana jako silna korelacja) i zerowe prawdopodobieństwo, żeby takie ułożenie miało być przypadkowe.

6. TWORZENIE I ANALIZA MODELI

Ta część projektu poświęcona jest tworzeniu i analizie modeli przewidujących ceny nieruchomości z biblioteki *scikit-learn*. W tym celu konieczne będzie przetworzenie bazy danych tak, aby była możliwa do odczytania przez funkcje szkolące oraz aby dane były sformatowane w sposób pozwalający na uzyskanie jak największej dokładności predykcji. Pierwszym krokiem będzie zastąpienie wartości tekstowych liczbowymi i przetestowanie różnych strategii działania z pustymi komórkami. Następnie tak przygotowane dane będzie można podstawić do wybranych funkcji szkolących, przetestować ich efektywność z różnymi argumentami, ewentualnie możliwe będzie usunięcie niektórych kolumn z bazy celem sprawdzenia, jaka liczba parametrów jest najlepsza dla szkolenia modelu.

6.1 Strategie zarządzania pustymi komórkami

Modele używane w projekcie nie mają natywnego wsparcia dla obsługi brakujących danych, wobec czego należy zastanowić się jak sprawić, aby we wszystkich komórkach znajdowały się wartości. Można tu wyróżnić dwa podejścia:

1) Usunięcie brakujących danych – jest to najprostsze rozwiązanie, polegające na usunięciu rekordów z brakującymi wartościami bądź kolumn, w których znajduje się wyjątkowo dużo pustych pól; niewątpliwą zaletą tego podejścia jest prostota implementacji oraz pewność, że wszystkie dane znajdujące się w przetworzonej bazie będą prawdziwe, dzieje się to jednak kosztem utraty części danych, a co za tym idzie – mniejszym zbiorem do wyszkolenia modelu, co może skutkować niższą wydajnością stworzonego modelu;

2) Zastąpienie brakujących danych wartościami przybliżonymi – kolejnym rozwiązaniem jest uzupełnienie brakujących danych, w tym celu można sprawdzić średnie wartości dla danej kolumny i nimi wypełnić puste komórki, kolejną opcją jest wypełnienie tych wartości różnymi popularnymi danymi, lub wartościami pasującymi do innych parametrów danego rekordu na bazie pozostałych, wypełnionych rekordów; główną zaletą tego rozwiązania jest zachowanie rozmiaru bazy danych, może się natomiast okazać, że uzupełnione wartości nie są prawidłowe i zaburzają ogólne zależności między parametrami, a ceną nieruchomości;

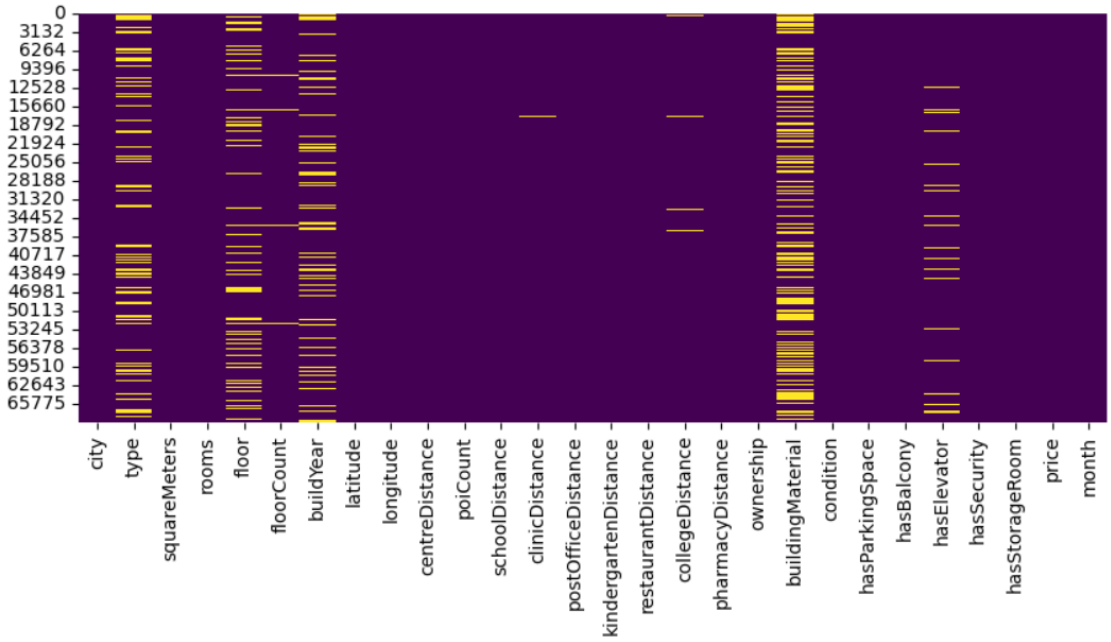
Oba podejścia mają istotne wady i zalety, trudno jest zatem wyłonić lepszą strategię bez przetestowania i porównania obydwu tych rozwiązań, co zostanie wykonane w następnych częściach pracy.

6.1.1 Usunięcie brakujących danych

W celu przygotowania zbioru z usuniętymi danymi należy skopiować przygotowaną wcześniej bazę do nowej zmiennej, aby nie naruszać oryginału. Następnie konieczne jest sprawdzenie, jak dużo komórek jest pustych i czy lepiej usunąć konkretne rekordy z brakującymi danymi, czy całe kolumny, w których występują duże braki.

```
plt.figure(figsize=(10, 4))
sns.heatmap(data_delete_nan.isna(), cmap='viridis', cbar=False)
plt.show()
```

Listing 6.1. Tworzenie mapy cieplnej



Rys. 6.1. Mapa cieplna brakujących rekordów w poszczególnych kolumnach

Taką mapę można stworzyć funkcją *heatmap* z biblioteki *Seaborn*, używając funkcji *isna* do znalezienia komórek o wartościach *NaN*, co pokazano na listingu 6.1. Obserwując rysunek 6.1 można zauważyć, że największe braki danych występują w kolumnach *buildingMaterial*, *type*, *floor* i *buildYear*. Widać również, że braki te występują na przestrzeni całego zbioru danych. Aby dowiedzieć się, jakie konkretnie liczby brakujących komórek znajdują się w konkretnych kolumnach można użyć funkcji *isna* na zbiorze w połączeniu z funkcją *sum*, co ukazano na listingu 6.2.

```
nan_counts = data_delete_nan.isna().sum()
print(nan_counts)
```

```
city                       0
type                   15278
squareMeters               0
rooms                      0
floor                  12472
floorCount               956
buildYear              11935
latitude                   0
longitude                  0
centreDistance             0
poiCount                   0
schoolDistance            69
clinicDistance           319
postOfficeDistance        95
kindergartenDistance      68
restaurantDistance       193
collegeDistance         2021
pharmacyDistance         107
ownership                  0
buildingMaterial       26791
condition                  0
hasParkingSpace            0
hasBalcony                 0
hasElevator             3545
hasSecurity                0
hasStorageRoom             0
price                      0
month                      0
dtype: int64
```

Listing 6.2. Liczby braków danych w poszczególnych kolumnach

Największe braki występują w kolumnie *buildingMaterial*, są one na tyle duże, że nie można usunąć wszystkich rekordów, które nie mają przypisanej wartości dla tego parametru. Po sprawdzeniu za pomocą wykresu, że zależności z tej kolumny nie wpływają silnie na cenę nieruchomości, można ją usunąć.

```
data_delete_nan = data_delete_nan.drop('buildingMaterial', axis=1)
```

Listing 6.3. Usunięcie kolumny *buildingMaterial*

W tym celu należy zastosować funkcję *drop* z parametrem *axis* ustawionym na 1, wykonanie pokazano na listingu 6.3. Po pozbyciu się kolumny *buildingMaterial* można sprawdzić, jak dużo rekordów w bazie ma wartości *NaN* w którejkolwiek kolumnie – pozwoli to ocenić, czy należy usunąć jeszcze jakąś kolumnę, czy też można przystąpić do usuwania rekordów.

```
count_nan_records = data_delete_nan.isna().any(axis=1).sum()
print(count_nan_records)
```

```
33372
```

Listing 6.4. Sprawdzenie liczby rekordów z wartościami *NaN*

Jak wynika z listingu 6.4 wierszy z brakami jest w bazie ponad 33 tysiące, czyli niemal połowa rekordów zawiera wartości *NaN*, co jest zbyt dużą wartością, aby je wszystkie usunąć – mogłaby to być zbyt duża strata do wyszkolenia odpowiedniego modelu. Należy zatem znaleźć kolejną kolumnę, którą można usunąć, po analizie na rysunku 6.1 można sprawdzić parametr *type*, gdyż również posiada dużo wartości *NaN*. Po upewnieniu się, że nie ma on liniowej zależności z ceną nieruchomości został on usunięty i pozostalo około 25 tysięcy rekordów z pustymi polami, lecz kolumny w których występują duże braki, czyli *floor* i *buildYear* mają zbyt duży wpływ na ostateczną cenę nieruchomości, aby je usuwać. Zatem należy usunąć resztę rekordów z brakującymi danymi.

```
data_delete_nan = data_delete_nan.dropna()
```

Listing 6.5. Usunięcie rekordów z brakującymi danymi

W tym celu można skorzystać z funkcji *dropna* z biblioteki *Pandas*, co zostało pokazane na listingu 6.5. W ten sposób otrzymany zbiór jest pozbawiony wartości *NaN*.

6.1.2 Zastąpienie brakujących danych

Teraz można przystąpić do budowy zbioru, który nie straci żadnych rekordów, dzięki zastąpieniu wartości *NaN* wartościami przybliżonymi. Dane na starcie posiadają takie braki, jak na rysunku 6.1 i listingu 6.2. Należy zatem uzupełnić dane w kolumnach *type*, *floor*, *floorCount*, *buildYear*, wszystkich kolumnach typu *[punkt_zainteresowania]Distance* oraz *buildingMaterial i hasElevator*. W procesie tym należy uważać, aby nie zaburzyć bazy, zatem należy położyć szczególny nacisk na uzupełnienie tych danych zgodnie z dynamiką kolumn i ich zależności w stosunku do ceny nieruchomości.

Proces ten można podzielić na dwa etapy – uzupełnianie wartości tekstowych i numerycznych – będą cechować się one innymi zasadami. W przypadku danych tekstowych są to w większości 2/3 wartości unikalne występujące w danej kolumnie. Trudno jest zatem wyciągnąć z nich wartość średnią i uzupełnić nią brakujące pola, a zapełnienie ich najpopularniejszą wartością, czy losowymi wartościami mogłoby zaburzyć stabilność modelu. Można zatem dodać w tych kolumnach wartość *unsigned*, dzięki czemu rozkład obecnych wartości nie zostanie zaburzony, a wartości *NaN* będą wyeliminowane.

```
columns_to_fill = ['type', 'buildingMaterial']
data_replace_nan[columns_to_fill] = data_replace_nan[columns_to_fill].fillna('unsigned')
```

Listing 6.6. Uzupełnienie brakujących wartości tekstowych

Takiego uzupełnienia moża dokonać wybierając kolumny do uzupełnienia, a następnie wypełniając je funkcją *fillna* – zostało to pokazane na listingu 6.6.

Na tym etapie do uzupełnienia pozostają jedynie kolumny z wartościami numerycznymi oraz jedna kolumna z wartościami typu *Boolean*. Pierwszym krokiem będzie uzupełnienie pustych komórek w kolumnach *[punkt_zainteresowania]Distance* określających dystans od najbliższego punktu zainteresowania. W tym przypadku można założyć, że jeśli komórka w takiej kolumnie jest pusta, nie ma w pobliżu określonego punktu zainteresowania. Jednak z racji tego, że potrzeba danych liczbowych, nie można wpisać tam wartości *none* lub *brak*, można zamiast tego wpisać odpowiednio duży dystans. W tym celu należy zerknąć na rysunki 5.4 i 5.5, z których jasno wynika, że maksymalne wartości w tych kolumnach zbliżone są do 5.0. Zatem w brakujących polach można wpisać wartość 7.0, która będzie odstawać od obecnych wartości tych kolumn, a jednocześnie zastąpi braki. Wykonanie tego etapu jest widoczne na listingu 6.7.

```
columns_to_fill = ['schoolDistance', 'clinicDistance', 'postOfficeDistance', 'kindergartenDistance',
                   'restaurantDistance', 'collegeDistance', 'pharmacyDistance']
data_replace_nan[columns_to_fill] = data_replace_nan[columns_to_fill].fillna(7.0)
```

Listing 6.7. Uzupełnienie brakujących danych w kolumnach typu *[punkt_zainteresowania]Distance*

Następnie można przejść do wypełnienia wartości w kolumnie *hasElevator*. Ma ona typ *Boolean*, wobec czego znalezienie średniej wartości jest również niemożliwe, można jednak dodać wartość liczbową oznaczającą brak informacji – interpreter odczytuje wartości *True* i *False* jako zera i jedynki, wobec czego można wartości *NaN* zastąpić liczbą 2, co też zostało wykonane na listingu 6.8.

```
data_replace_nan[['hasElevator']] = data_replace_nan[['hasElevator']].fillna(2)
```

Listing 6.8. Uzupełnienie kolumny hasElevator

Do wypełnienia pozostają jeszcze kolumny *floor, floorCount i buildYear*. Są to wartości stricte liczbowe, wobec nich najbezpieczniejszym podejściem będzie sprawdzenie średniej wartości i wypełnienie nią brakujących rekordów. Aby zachować lepszą stabilność bazy można te dane obliczyć dla konkretnych miast, ponieważ mogą się one różnić w zależności od lokalizacji.

```
columns = data_replace_nan[['floor', 'floorCount', 'buildYear']]
mean_values_by_city = columns.groupby(data_replace_nan['city']).mean()
print(mean_values_by_city)
```

```
               floor  floorCount    buildYear
city
bialystok   3.431700    5.196041  1987.273438
bydgoszcz   2.784920    4.049033  1973.272074
czestochowa 2.900576    4.606618  1975.584362
gdansk      3.099744    4.965366  1986.548628
gdynia      2.817483    4.451256  1987.586376
katowice    3.633991    5.876557  1973.316434
krakow      3.050501    4.796379  1989.247046
lodz        3.226527    5.023749  1974.880478
lublin      3.088573    4.481755  1983.919064
poznan      3.373262    5.272154  1981.731443
radom       3.514178    5.155294  1974.464837
rzeszow     3.792359    5.949650  2000.041667
szczecin    3.143605    4.453922  1966.097263
warszawa    3.724031    6.105095  1988.858480
wroclaw     3.062637    4.871830  1981.141323
```

Listing 6.9. Sprawdzenie średnich wartości wybranych kolumn dla miast

Można tego dokonać wybierając konkretne kolumny, następnie za pomocą metody *groupby* uzależnić je od miast i funkcją *mean* sprawdzić średnie wartości, jak na listingu 6.9. Mając tak dostępne wartości średnie można przystąpić do wypełniania tych kolumn.

```
def fillna_by_city(row):
    city = row['city']
    for col in ['floor', 'floorCount', 'buildYear']:
        if pd.isna(row[col]):
            row[col] = mean_values_by_city.loc[city, col]
    return row
```

Listing 6.10. Funkcja wypełniająca puste wartości numeryczne średnią

W tym celu przygotowano funkcję *filnna_by_city* widoczną na listingu 6.10, przyjmującą jako argument wiersz struktury *DataFrame*. Funkcja sprawdza, jaka jest wartość pola *city* i przypisuje ją do zmiennej. Następnie w pętli *for* sprawdza, czy któryś z parametrów *floor, floorCount* lub *buildYear* ma wartość *NaN*. Jeśli tak, przypisuje do tego pola wartość znalezioną w strukturze *mean_values_by_city*, utworzonej na listingu 6.9, odpowiadającą wartości średniej dla konkretnego pola. Tak przygotowaną funkcję można zaimplementować dla bazy danych.

```
data_replace_nan = data_replace_nan.apply(fillna_by_city, axis=1)
```

Listing 6.11. Wypełnienie pustych wartości numerycznych

Zbiór po wykonaniu kodu z listingu 6.11 nie ma żadnych wartości *NaN*, można zatem przejść do kolejnego etapu.

6.2 Zamiana danych tekstowych na liczby

Aby dane mogły być przetworzone przez funkcje szkolące, ich typ musi być co najmniej liczbowy (w niektórych przypadkach nawet wyłącznie zmiennoprzecinkowy). Wobec tego możliwe są dwa podejścia do kodowania zmiennych tekstowych:

1) *LabelEncoder* – przypisuje każdej unikalnej wartości w danej kolumnie wartość liczbową całkowitą, po czym zamienia wartości tekstowe na liczbowe; niewątpliwymi zaletami tej metody są jej prosta implementacja i nie powiększanie zbioru danych, jednak ma ona poważną wadę – wrażliwe algorytmy (jak na przykład regresja liniowa) mogą błędnie interpretować porządek wśród stworzonego zbioru liczb, a nie zawsze są one powiązane, co może prowadzić do błędów podczas szkolenia modelu;

2) *OneHotEncoder* – dla każdej unikalnej wartości tworzy nową kolumnę (a zatem dla kolumny *cities* stworzyłby dodatkowe 15 kolumn), następnie przypisuje wartości 1 lub 0, w zależności od tego, czy dany rekord należy do danej kategorii; zdecydowaną zaletą tego modelu jest eliminacja potencjalnych błędów wynikających z próby interpretacji porządku przez algorytm, natomiast w ten sposób szybko powiększa wymiarowość danych, co może być problematyczne dla dużych zbiorów danych.

W przypadku zbioru danych rozpatrywanego w tym projekcie logicznym podejściem byłoby zastosowanie wyłącznie metody *OneHotEncoder* z uwagi na fakt, że w większości pomiędzy danymi tekstowymi, a ceną nie ma prostej zależności, a funkcja szkoląca model może doszukiwać się zależności liniowej pomiędzy etykietami w kolumnach tekstowych, a wartością. Dodanie nowych kolumn może rozwiązać ten problem, jednak z uwagi na charakter działania metody *OneHot* wymagałoby to dodania ponad 20 nowych kolumn, co znacząco zwiększyłoby objętość zbioru, a w efekcie może wydłużyć proces uczenia.

Wobec tego, że zarówno pierwsze, jak i drugie podejście ma istotne zalety i wady w kontekście tego projektu, przetestowane zostaną obie te strategie.

6.2.1 Label Encoder

LabelEncoder to klasa modułu *sklearn.preprocessing* umożliwiająca normalizację etykiet oraz transformację etykiet tekstowych na numeryczne. W tym projekcie będzie używana jedynie do zamiany wartości tekstowych na liczby za pomocą funkcji *fit_transform*, która jako argument przyjmuje dane tabelaryczne (eng. *array-like of shape*), zwraca zaś taki sam typ danych ze zmienionymi etykietami. Wobec tego należy wybrać kolumny z uprzednio stworzonych zbiorów, które wymagają transformacji.

```
print(data_replace_nan.dtypes)
```

```
city                    object
type                    object
squareMeters           float64
rooms                  float64
floor                  float64
floorCount             float64
buildYear              float64
latitude               float64
longitude              float64
centreDistance         float64
poiCount               float64
schoolDistance         float64
clinicDistance         float64
postOfficeDistance     float64
kindergartenDistance   float64
restaurantDistance     float64
collegeDistance        float64
pharmacyDistance       float64
ownership               object
buildingMaterial        object
condition               object
hasParkingSpace           bool
hasBalcony                bool
hasElevator             object
hasSecurity               bool
hasStorageRoom            bool
price                    int64
month                    int64
dtype: object
```

Listing 6.12. Wizualizacja typów danych w zbiorze

Typy danych w zbiorze można sprawdzić za pomocą własności *dtypes* obiektu *DataFrame*. Z listingu 6.12 jasno wynika, że transformacji należy poddać kolumny *city*, *type*, *ownership*, *buildingMaterial*, *condition* i *hasElevator*. Aby to zrobić należy stworzyć obiekt klasy *LabelEncoder* i użyć funkcji *fit_transform*. Analogiczne akcje zostaną przeprowadzone dla drugiego zbioru danych, różnić będzie się tylko liczba kolumn.

```
data_replaced_labeled['city'] = enc.fit_transform(data_replaced_labeled['city'])
data_replaced_labeled['type'] = enc.fit_transform(data_replaced_labeled['type'])
data_replaced_labeled['ownership'] = enc.fit_transform(data_replaced_labeled['ownership'])
data_replaced_labeled['buildingMaterial'] = enc.fit_transform(data_replaced_labeled['buildingMaterial'])
data_replaced_labeled['condition'] = enc.fit_transform(data_replaced_labeled['condition'])
data_replaced_labeled['hasElevator'] = enc.fit_transform(data_replaced_labeled['hasElevator'])
```

Listing 6.13. Transformacja etykiet w poszczególnych kolumnach

Po utworzeniu obiektu *enc* klasy *LabelEncoder* i kopii obiektu *data_replace_nan* o nazwie *data_replaced_labeled* można użyć funkcji *fit_transform* na poszczególnych kolumnach, jak zaprezentowano na listingu 6.13. Po tej operacji wszystkie typy w

strukturze są numeryczne, co czyni je gotowymi do przetwarzania przez funkcje szkolące modele predykcyjne.

6.2.2. OneHotEncoder

OneHotEncoder również jest klasą modułu *sklearn.preprocessing*. Do jej użycia można zastosować klasę *ColumnTransformer* z biblioteki *sklearn.compose*. Jej wykorzystanie pozwala na wygodne zarządzanie enkoderami kolumn, może się okazać przydatna także w dalszej części projektu, przy próbie testowania modelu – dane wtedy trzeba będzie dostosować w ten sam sposób, jak dane treningowe, wobec czego zapisany sposób ich transformacji okaże się niezbędny. Istnieje również alternatywna wersja kodowania danych metodą *OneHot*, za pomocą funkcji *get_dummies* z biblioteki *Pandas*, jednak dla utrzymania spójności wykorzystywanych metod zostanie tu wykorzystana klasa *OneHotEncoder*.

```
ct = make_column_transformer(
  (OneHotEncoder(), ['city', 'type', 'ownership', 'buildingMaterial', 'condition']))
```

```
ct.fit(data_replaced_onehot)
```

```
ColumnTransformer(transformers=[('onehotencoder', OneHotEncoder(),
                                 ['city', 'type', 'ownership',
                                  'buildingMaterial', 'condition'])])
onehotencoder
['city', 'type', 'ownership', 'buildingMaterial', 'condition']
OneHotEncoder
OneHotEncoder()
```

Listing 6.14. Stworzenie obiektu *ColumnTransformer* i wstępna obróbka danych

Obiekt *ColumnTransformer* można stworzyć za pomocą funkcji *make_column_transformer*, jako argument podając rodzaj enkodera i nazwy kolumn, do których ma być użyty (można podać także większą liczbę enkoderów). Następnie należy użyć funkcji *fit*, która wywołuje funkcję *fit* dla każdego enkodera, w tym przypadku wstępnie przetwarza dane, zamieniając zmienne tekstowe na wartości liczbowe – funkcja *fit_transform* obiektu *OneHotEncoder* nie przyjmuje wartości tekstowych, więc należy się ich wcześniej pozbyć. Na listingu 6.14 można zobaczyć także strukturę transformera – jest on swego rodzaju kontenerem na enkodery, pomocny także w przypadku użycia większej ich liczby do jednego zbioru danych.

Mając tak przygotowane dane można wywołać funkcję *fit_transform*, która w tym przypadku dla każdej unikalnej wartości we wszystkich kolumnach wymagających transformacji stworzy nową kolumnę, a tak powstałą tabelę wypełni zerami i jedynkami w zależności od przynależności rekordów do konkretnych kategorii.

```
enc_data = pd.DataFrame(ct.fit_transform(data_replaced_onehot).toarray())
```

Listing 6.15. Wywołanie funkcji *fit_transform*

Należy jednak pamiętać, że funkcja *fit_transform* zwraca tu obiekt klasy *scipy.sparse._csr.csr_matrix*, a używanym dotychczas typem jest *DataFrame*, wobec czego konieczna jest konwersja – najpierw do tablicy za pomocą funkcji *toarray*, następnie do *DataFrame* umieszczając całość w konstruktorze tego typu, jak zaprezentowano na listingu 6.15. W ten sposób otrzymano obiekt z zakodowanymi danymi.

```
enc_data.head()
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	1.0
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0

Listing 6.16. Struktura zakodowanych kolumn

Jak wynika z listingu 6.16 otrzymano 27 kolumn zapełnionych wartościami zero-jedynkowymi. Można zatem pozbyć się oryginalnych kolumn ze zbioru i dołączyć kolumny zakodowane, co przedstawiono na listingu 6.17.

```
data_replaced_onehot = data_replaced_onehot.drop(columns = ['city', 'type', 'ownership', 'buildingMaterial', 'condition'])
```

```
data_replaced_onehot = data_replaced_onehot.join(enc_data)
```

Listing 6.17. Zamiana oryginalnych kolumn na zakodowane

Te same kroki podjęto dla drugiego zbioru danych, w wyniku czego otrzymano dwie struktury zakodowane metodą *OneHot*. Pierwsza ma 46, a druga 43 kolumny, co jest dużą liczbą w porównaniu do początkowych 29 kolumn, jednak może mieć znaczący wpływ na wydajność wyszkolonego modelu.

Łącznie stworzono 4 zbiory:

1) *data_deleted_labeled* – dane z usuniętymi pustymi rekordami zakodowane metodą *LabelEncoder;*
2) *data_replaced_labeled* – dane z uzupełnionymi brakującymi danymi zakodowane metodą *LabelEncoder;*
3) *data_deleted_onehot* – dane z usuniętymi pustymi rekordami zakodowane metodą *OneHot;*
4) *data_replaced_onehot* – dane z uzupełnionymi brakującymi danymi zakodowane metodą *OneHot.*

Wszystkie te zbiory zostaną dostarczone jako argumenty do uczenia modeli predykcyjnych w celu wyłonienia najbardziej efektywnej metody obróbki danych.

6.3 Przygotowanie modelu predykcyjnego z użyciem biblioteki scikit-learn

Pierwszym krokiem w celu stworzenia modelu jest podzielenie danych wejściowych na zbiór treningowy i testowy, a pośród tych zbiorów wyłonienie parametrów i wartości. Można to zrobić za pomocą funkcji *train_test_split* z modułu *sklearn.model_selection*, co zaprezentowano na listingu 6.18.

```
x_train , x_test , y_train , y_test = train_test_split(data_deleted_labeled.drop('price', axis=1),
                                                       data_deleted_labeled['price'],
                                                       test_size = 0.10,random_state = 29)
```

Listing 6.18. Podział danych na treningowe i testowe

Funkcja *train_test_split* jako pierwsze argumenty przyjmuje struktury tabelaryczne o tej samej długości z danymi treningowymi i odpowiadającymi im wartościami. Ponadto można ustawić argumenty opcjonalne, w tym przypadku użyto *test_size* celem zdefiniowania rozmiaru danych testowych jako 10% zbioru i *random_state* do kontroli generatora liczb losowych. Funkcja ta dodatkowo miesza dane wejściowe, co jest niezbędne w przypadku zbioru rozpatrywanego w tym projekcie z uwagi na fakt, że dane są posortowane według parametru *city*. Gdyby nie zostały posortowane wydajność modelu mogłaby zostać zaniżona, ponieważ dane testowe nie byłyby reprezentacyjne. Jako wynik otrzymano zbiór testowy i treningowy podzielony na dane i wartości.

Tak przygotowane zbiory można używać do szkolenia modeli predykcyjnych. Z uwagi na wcześniejsze obserwacje jako pierwszy algorytm wybrana została regresja liniowa, zaimplementowana w *scikit-learn* jako *LinearRegression* w module *linear_model*. Algorytm ten działa za pomocą metody najmniejszych kwadratów – próbuje zminimalizować sumę kwadratów różnic między obserwowanymi wartościami docelowymi w zestawie danych, a wartościami docelowymi przewidywanymi przez liniową aproksymację.

Aby stworzyć taki model należy utworzyć obiekt klasy *LinearRegression*. Przyjmuje ona 4 argumenty wejściowe w konstruktorze:

1) *fit_intercept* o typie *Boolean* – jeśli ustawione na *True*, model będzie uwzględniał wyraz wolny podczas dopasowywania modelu, domyślnie ustawione na *True*;
2) *copy_x* o typie *Boolean* – jeśli ustawione na *True* zbiór danych wejściowych zostanie skopiowany, w przeciwnym wypadku może być nadpisany, domyślnie ustawione na *True*;
3) *n_jobs* o typie *Integer* – liczba procesów do użycia podczas szkolenia modelu, domyślnie ustawiona na *None*, co oznacza najczęściej 1 proces, ustawienie to pomaga w szczególności przy dużych zbiorach danych;
4) *positive* o typie *Boolean* – gdy ustawione na *True* wymusza, by współczynniki były dodatnie, opcja ta jest obsługiwana tylko dla macierzy gęstych, domyślnie ustawiona na *False*.

W tym przypadku odpowiedni będzie model ze wszystkimi argumentami ustawionymi na domyślne. Następnie można użyć funkcji *fit*, służącej do wyszkolenia modelu. Przyjmuje ona dane oraz wartości docelowe w postaci tablic o identycznej długości, jako

opcjonalny argument można również dodać wagi dla poszczególnych rekordów, jednak w tym przypadku wszystkie próbki mają identyczną wagę. Proces ten ukazano na listingu 6.19.

```
reg = LinearRegression()
```

```
reg.fit(x_train, y_train)
```

```
▾ LinearRegression
LinearRegression()
```

Listing 6.19. Stworzenie i wyszkolenie modelu *LinearRegression*

Stworzenie takiego modelu zajęło niecałą sekundę, ze względu na stosunkowo niewielką liczbę danych. Aby sprawdzić wydajność takiego modelu należy użyć funkcji *score*, która jako parametry przyjmuje tabelę z danymi i ich wartościami docelowymi, ważne aby nie były to jednak dane używane podczas trenowania, gdyż te są już „znane" dla modelu, co dałoby niemiarodajny wynik funkcji.

$$R^2 = \frac{\sum_{t=1}^{n}(\widehat{y_t} - \bar{y})^2}{\sum_{t=1}^{n}(y_t - \bar{y})^2} \quad (6.1)$$

n – łączna liczba obserwacji
y_t – rzeczywista wartość dla obserwacji t
$\widehat{y_t}$ – wartość dla obserwacji t wyznaczona przez model
$\bar{y}$ – średnia wartość zmiennych zależnych (wartości)

Jako wynik metoda *score* zwraca współczynnik determinacji (6.1), który pozwala ocenić, jak dobrze model wyjaśnia zmienność danych – jest to wartość zmiennoprzecinkowa z przedziału <0,1>, czym bardziej zbliżona do 1 tym lepsza wydajność modelu.

```
reg.score(x_test,y_test)
```

```
0.6066081013835932
```

Listing 6.20. Sprawdzenie wydajności modelu metodą *score*

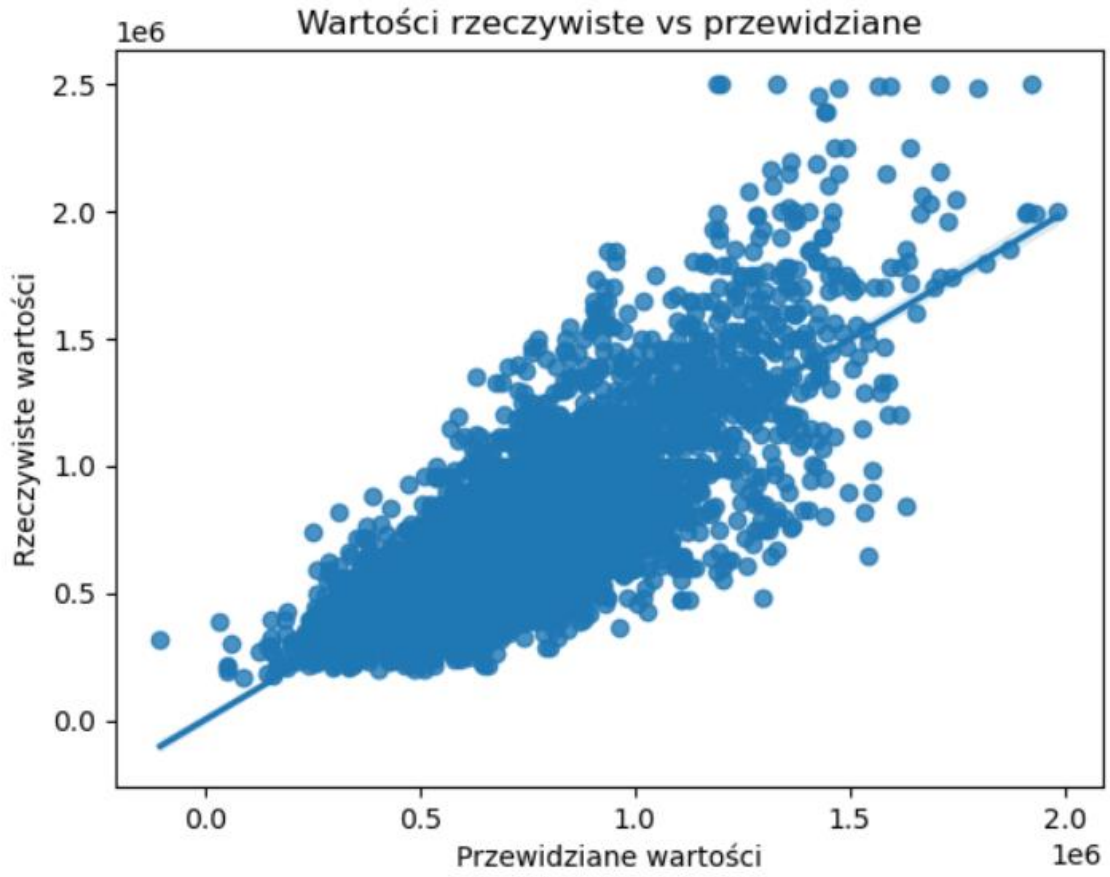
Jak można zauważyć na listingu 6.20 wydajność wyszkolonego modelu wynosi ≈ 0.6. Można to uznać za wynik niezadowalający, aby to lepiej zwizualizować należy porównać wartości przewidziane przez model z wartościami rzeczywistymi. W tym celu trzeba użyć funkcji *predict* dla zbioru testowego, metoda ta przewiduje wartości dla przekazanych danych, jako wynik zwraca tablicę z wartościami przewidzianymi. Otrzymaną tablicę należy porównać ze zbiorem wartości rzeczywistych dla przekazanego zbioru.

```
y_pred = reg.predict(x_test)
```

```
sns.regplot(x=y_pred, y=y_test)
plt.xlabel('Przewidziane wartości')
plt.ylabel('Rzeczywiste wartości')
plt.title('Wartości rzeczywiste vs przewidziane')
plt.show()
```

Listing 6.21. Użycie metody *predict* i stworzenie wykresu regresji liniowej

Wykres regresji liniowej można utworzyć funkcją *regplot* z biblioteki *Seaborn*, jak pokazano na listingu 6.21.



Rys. 6.2. Wykres regresji liniowej

Wykres na rysunku 6.2 przedstawia linię regresji i dopasowanie danych przewidzianych do rzeczywistych wartości. Widoczne są duże rozrzuty, w szczególności przy najwyższych wartościach – stąd płynie wniosek, że skrajne wartości zaburzają stabilność modelu. Jednak linia regresji wskazuje, że dane są pozytywnie skorelowane. W przypadku tego modelu nie ma parametrów szkolenia które można zmienić, zatem przeprowadzone zostały te same kroki dla pozostałych trzech zbiorów.

Tabela 6.1. Wyniki szkolenia dla badanych zbiorów danych

Zbiór danych	Wynik szkolenia (*score*)
data_deleted_labeled	0.6066081013835932
data_replaced_labeled	0.6040711105089784
data_deleted_onehot	0.7470282162223789
data_replaced_onehot	0.7555898977136916

Z tabeli 6.1 wynika, że modele tworzone w oparciu o dane zakodowane metodą *OneHot* cechują się znacznie wyższym parametrem *score*, z niewielką przewagą dla danych z uzupełnionymi wartościami *NaN*. Po sprawdzeniu wykresu regresji widać jednak dalej duże odstępstwa danych przewidzianych przez model od danych rzeczywistych, sięgające nawet 500 000 zł w skrajnych przypadkach, najczęściej wynoszące jednak około 100 000 zł, wobec czego wynik ten dalej nie może zostać uznany za satysfakcjonujący.

6.4 Gradient Boosting

Po niepowodzeniu w wyszkoleniu algorytmu klasyczną metodą regresji liniowej można spróbować stworzyć model z wykorzystaniem *Gradient Boosting*. Jest to rodzaj algorytmu uczenia maszynowego, który buduje model predykcyjny za pomocą kombinacji słabych modeli w celu uzyskania większej dokładności. Każdy kolejny model jest dostosowywany w celu poprawy błędów popełnionych przez modele już istniejące. Metoda ta cechuje się zdolnością modelowania nieliniowych zależności, gdyż drzewa decyzyjne używane w tym algorytmie są bardziej elastyczne, a co za tym idzie – metoda ta lepiej sobie radzi z obserwacjami odstającymi (ang. *outliers*). Rozwiązanie to wymaga dużo więcej zasobów do wyszkolenia modelu niż klasyczne algorytmy, jednak jak podaje Chaphalkar N. B. algorytmu tego typu mogą sobie radzić dużo lepiej, niż klasyczna regresja liniowa[13].

W celu stworzenia modelu metodą *Gradient Boosting* użyta zostanie klasa *GradientBoostingRegressor* z modułu *sklearn.ensemble*. W konstruktorze przyjmuje parametry definiujące charakter modelu:

1) *n_estimators* typu *int* – liczba drzew w modelu, domyślnie 100;
2) *learning_rate* typu *float* – współczynnik uczenia, regulujący wkład każdego drzewa, domyślnie 0.1;
3) *max_depth* typu *int* lub *none* – maksymalna głębokość każdego drzewa, domyślnie 3;
4) *min_samples_split* typu *int*/*float*– minimalna liczba próbek wymagana do podziału węzła w drzewie, domyślnie 2;
5) *loss* typu *string* – funkcja straty używana podczas budowy modelu, domyślnie *squared_error*, dostępne także *absolute_error, huber* i *quantile*;
6) *verbose* typu *int* – parametr umożliwiający wyświetlanie informacji o progresie szkolenia, domyślnie 0, czym większa liczba tym częściej wypisywane informacje.

Klasa ta przyjmuje dużo więcej parametrów, jednak jedynie te wymienione będą modyfikowane podczas badań, są one najbardziej determinujące dla jakości wyszkolonego modelu i zasobów używanych podczas szkolenia.

```
gbr = ensemble.GradientBoostingRegressor(n_estimators = 200, max_depth = 5, min_samples_split = 2,
          learning_rate = 0.1, loss = 'squared_error', verbose = 1)
```

Listing 6.22. Utworzenie obiektu *GradientBoostingRegressor*

Po utworzeniu obiektu klasy *GradientBoostingRegressor* jak na listingu 6.22, można przystąpić do szkolenia i sprawdzania wydajności modelu. Klasa ta dostarcza funkcji *fit*, *score* i *predict* podobnie jak w przypadku *LinearRegression*. W tym przypadku zostanie zastosowany zbiór *data_replaced_onehot* z uwagi na wysoki wynik w poprzednich testach. Konieczne będzie także przetestowanie różnych parametrów szkolenia.

```
gbr.fit(x_train, y_train)
```

```
      Iter       Train Loss   Remaining Time
         1 107142781814.7420            1.85m
         2 93337294476.5867            1.91m
         3 81750571420.4122            1.90m
         4 72148599708.9937            1.90m
         5 64237252949.3434            1.84m
         6 57608402844.0834            1.83m
         7 52196594904.2795            1.80m
         8 47564095022.0413            1.78m
         9 43643074944.8264            1.77m
        10 40354230443.1472            1.75m
        20 23955394101.4631            1.63m
        30 19089274326.7867            1.53m
        40 16846981828.0225            1.43m
        50 15722070982.0239            1.34m
        60 14893434415.1648            1.25m
        70 14332835241.7498            1.16m
        80 13837251911.5424            1.07m
        90 13341690887.4141           58.47s
       100 12890633616.6828           53.07s
       200 10592779086.6510            0.00s
```

```
GradientBoostingRegressor(max_depth=5, n_estimators=200, verbose=1)
```

Listing 6.23. Szkolenie modelu za pomocą *GradientBoostingRegressor*

Parametr *verbose* ustawiony w konstruktorze na wartość 1 pozwala na obserwację procesu szkolenia na bieżąco, co widać na listingu 6.23.

```
gbr.score(x_test, y_test)
```

```
0.9015484197040572
```

Listing 6.24. Wynik szkolenia metodą *Gradient Boosting*

Jak wynika z listingu 6.24, za pierwszą próbą otrzymano dużo lepsze wyniki niż za pomocą algorytmu *LinearRegression*, jednak należy pamiętać, że szkolenie trwa dużo dłużej i zużywa więcej zasobów – w tym przypadku trwało to około 2 minut, w przeciwieństwie do około sekundy w przypadku regresji liniowej. Model można jednak dostrajać za pomocą podawania różnych argumentów w konstruktorze klasy. W procesie tym zazwyczaj należy zwracać szczególną uwagę na podane parametry, aby nie doprowadzić do przetrenowania modelu (ang. *overfitting*), w tym przypadku jednak model jest wysoce odporny na to zjawisko (zgodnie z dokumentacją[6]), co znacząco ułatwia zadanie.

Proces dostrajania modelu obejmuje zwiększanie parametrów *n_estimators*, *max_depth* i *min_samples_split*, modyfikowanie *learning_rate* jak również wybór różnych funkcji straty. Każde zwiększenie parametrów *n_estimators* i *max_depth* znacząco wydłuża czas trenowania modelu.

Tabela 6.2. Wpływ parametru *n_estimators* na wydajność modelu

Parametr *n_estimators*	Wynik *score*
200	0.9015484197040572
300	0.9117666557346096
400	0.9188979648374778
500	0.9243770276596536
600	0.9291525463699922

Tabela 6.3. Wpływ parametru *max_depth* na wydajność modelu

Parametr *max_depth*	Wynik *score*
5	0.9015484197040572
10	0.9639867519922809
15	0.9676332056656941
20	0.9580243135568205

Tabela 6.4. Wpływ parametru *min_samples_split* na wydajność modelu

Parametr *min_samples_split*	Wynik *score*
2	0.9015484197040572
4	0.9146122868195299
6	0.9148675678725119
8	0.9160763518674946

Tabela 6.5. Wpływ parametru *learning_rate* na wydajność modelu

Parametr *learning_rate*	Wynik *score*
0.05	0.8854011094114969
0.1	0.9015484197040572
0.2	0.9169190801709567
0.3	0.9264044835805282

W procesie dostrajania przyjęto wartości bazowe parametrów modelu zaprezentowane na listingu 6.22 i modyfikowano po jednym parametrze, jak w tabelach 6.2-6.6, co pozwoliło na obserwację zachowania modelu pod wpływem różnych modyfikacji. Jako funkcję straty ostatecznie wybrano *huber*, gdyż łączy ona cechy *squared_error* i *absolute_error*, wartości wszystkich parametrów przedstawiono na listingu 6.25.

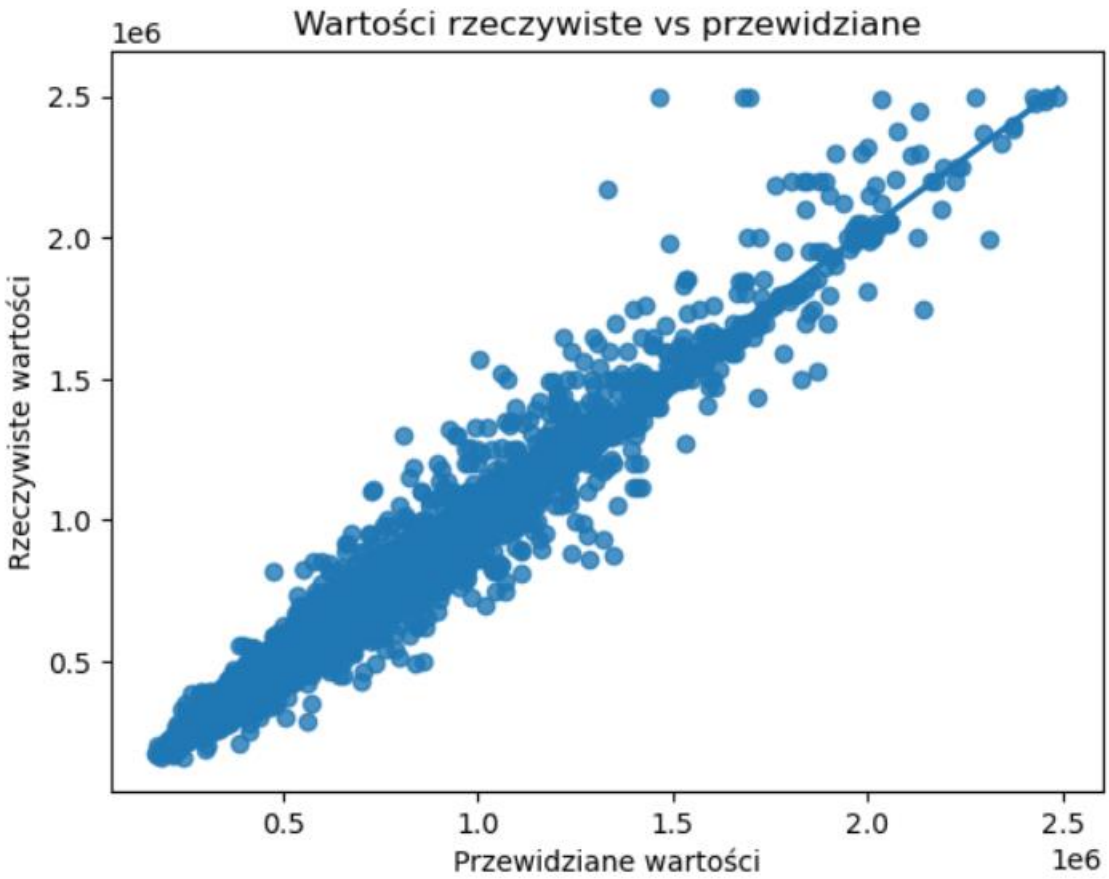
```
gbr = ensemble.GradientBoostingRegressor(n_estimators = 400, max_depth = 10, min_samples_split = 4,
          learning_rate = 0.1, loss = 'huber', verbose = 1)
```

Listing 6.25. Ostateczna struktura przygotowanego modelu

Można zauważyć, że nie są to wartości, które osiągały najlepsze wyniki parametru *score*, lecz trzeba mieć na uwadze, że równie ważna jest liczba zasobów potrzebna do wyszkolenia takiego modelu – należy dążyć do tego, aby była ona jak najmniejsza. Czas oczekiwania na wyszkolenie modelu o najlepszych parametrach zgodnie z pomiarami w tabelach 6.2-6.5 na urządzeniu wykorzystywanym w projekcie wynosił około godziny, podczas gdy wyszkolenie modelu zaproponowanego na listingu 6.25 trwa około 10 minut. Różnica w wydajności tych modeli to około 1,5%, co czyni zaproponowane rozwiązanie odpowiednim kompromisem pomiędzy wydajnością, a kosztem stworzenia modelu. Parametr *score* dla tego modelu wynosi ≈ 0.96, a jego wykres regresji przedstawiony jest na rysunku 6.3. Wynika z niego, że zdecydowana większość danych znajduje się na linii regresji, co jest wysoce pożądane, dalej jednak znajdują się pewne wartości odstające, do ich eliminacji może być konieczna dalsza edycja zbioru danych. Można sprawdzić także średni błąd bezwzględny (6.2) – jak wynika z listingu 6.26 wynosi on ≈ 36,5 tysiąca złotych, co jest wynikiem akceptowalnym z uwagi na fakt, że średnia cena nieruchomości w rozpatrywanym zbiorze wynosi ≈ 700 tysięcy złotych, jak wynika z rysunku 5.5. Stworzony model można zatem uznać za satysfakcjonujące rozwiązanie.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_t - \hat{y_t}| \quad (6.2)$$

n – łączna liczba obserwacji
y_t – rzeczywista wartość dla obserwacji t
$\hat{y_t}$ – wartość dla obserwacji t wyznaczona przez model



Rys. 6.3. Wykres regresji ostatecznego modelu

```
from sklearn.metrics import mean_absolute_error

mae = mean_absolute_error(y_test, y_pred)

print(f"Średni błąd bezwzględny: {mae}")
```

```
Średni błąd bezwzględny: 36685.09280430686
```

Listing 6.26. Sprawdzenie średniego błędu bezwzględnego

PODSUMOWANIE

W projekcie znaleziono bazę danych, którą poddano dokładnej analizie, następnie modyfikacji w celu jej najlepszego przystosowania do wyszkolenia modelu predykcyjnego. Rozpatrzono cztery różne podejścia do edycji danych, z których wyłoniono najlepsze, badając wydajność każdego z tych zbiorów w modelu regresji liniowej. Najlepszą metodą okazało się uzupełnienie brakujących danych w zbiorze predykcjami i zakodowanie kolumn tekstowych. Stwierdzono, że modele otrzymane klasyczną metodą regresji liniowej nie mają satysfakcjonującej wydajności, wobec czego zastosowano algorytm *Gradient Boosting*, który przyniósł znacznie lepsze rezultaty z uwagi na jego zaawansowanie. Algorytm ten wymagał jednak dostrojenia, czego dokonano poprzez serię pomiarów zachowania modelu predykcyjnego z różnymi parametrami.

Projekt ten pokazał jak ważna w procesie tworzenia modelu predykcyjnego jest sama analiza i odpowiednia modyfikacja danych, już ten aspekt pozwala na osiągnięcie znacznie lepszych wyników szkolenia modeli predykcyjnych. Ponadto w przypadku zaawansowanych algorytmów takich jak *Gradient Boosting* kluczowe jest zapoznanie się z dokumentacją, co pozwala na zrozumienie poszczególnych parametrów i odpowiednie ich modyfikowanie celem osiągnięcia wyższej wydajności.

Model przygotowany w projekcie można stosować do predykcji cen nieruchomości, przykładowo w serwisach internetowych czy w biurach nieruchomości. Nie jest on jednak prosty w użyciu, gdyż jest wytrenowany na zakodowanych danych. W kolejnych etapach projektu można zatem przygotować interfejs użytkownika pozwalający na wygodne korzystanie ze stworzonego modelu. Ponadto można rozpatrzyć wykorzystanie innych, bardziej zaawansowanych algorytmów, jak na przykład sieci neuronowe – mogą one pomóc osiągać jeszcze lepsze rezultaty.

BIBLIOGRAFIA

1. Źródło bazy danych {HYPERLINK „https://www.kaggle.com/datasets/krzysztofjamroz/apartment-prices-in-poland” data dostępu 30.11.2023 r.}
2. Licencja bazy danych {HYPERLINK „https://www.apache.org/licenses/LICENSE-2.0” data dostępu 30.11.2023 r.}
3. Open Street Map {HYPERLINK “https://www.openstreetmap.org/” data dostępu 30.11.2023 r.}
4. Python {HYPERLINK “https://www.python.org/” data dostępu 6.01.2024 r.}
5. Jupyter Notebook {HYPERLINK “https://jupyter.org/” data dostępu 6.01.2024 r.}
6. scikit-learn {HYPERLINK “https://scikit-learn.org/” data dostępu 6.01.2024 r.}
7. Pandas {HYPERLINK “https://pandas.pydata.org/” data dostępu 6.01.2024 r.}
8. Matplotlib {HYPERLINK “https://matplotlib.org/” data dostępu 6.01.2024 r.}
9. Seaborn {HYPERLINK “https://seaborn.pydata.org/” data dostępu 6.01.2024 r.}
10. Pyplot {HYPERLINK “https://docs.scipy.org/doc/scipy/index.html” data dostępu 17.01.2024 r.}
11. Kevin Jolly - Machine Learning with Scikit-learn Quick Start Guide, Packt Publishing, 2018 s.6-62
12. Baldominos, A.; Blanco, I.; Moreno, A.J.; Iturrarte, R.; Bernárdez, Ó.; Afonso, C. Identifying Real Estate Opportunities Using Machine Learning. Appl. Sci. 2018, 8, 2321. {HYPERLINK “https://doi.org/10.3390/app8112321” data dostępu 17.01.2024 r.}
13. Chaphalkar, N. & Sandbhor, Sayali. (2013). Use of Artificial Intelligence in Real Property Valuation. International Journal of Engineering and Technology. 5. 2334-2337. {HYPERLINK “https://www.researchgate.net/publication/286560163_Use_of_Artificial_Intelligence_in_Real_Property_Valuation” data dostępu 17.01.2024 r.}

SPIS LISTINGÓW

SPIS RYSUNKÓW

SPIS TABEL