

Sprawozdanie na konwersatorium

Z przedmiotu „Zastosowanie Metod Uczenia Maszynowego”

WSZiB Kraków, 2025

Temat projektu:

**Uczenie nadzorowane modelu w celu predykcji cen nieruchomości w
Krakowie i okolicach**

wykonał: Bartłomiej Potoniec

data wykonania: 15.01.2025

Wstęp

W niniejszym sprawozdaniu przedstawiłem realizację projektu związanego z wykorzystaniem metod uczenia maszynowego służącego do predykcji cen nieruchomości w Krakowie i okolicach. Temat taki wybrałem, ponieważ jestem w trakcie pisania aplikacji typu Otodom i chciałbym w niej wykorzystać mój model predykcyjny w taki sposób, aby podczas dodawania ogłoszenia po wypełnieniu wymaganych pól formularza aplikacja (a za kulisami model w Azure ML) sama podała sugerowaną cenę wystawianego mieszkania. Jestem zdania, że człowiek najlepiej uczy się przy praktycznych projektach, a najlepiej takich, które są później przydatne. W tym przypadku spełniłem obie przesłanki.

Jeśli mowa o przewidywaniu ceny, to ciężko wyobrazić sobie lepszą metodę uczenia niż regresja, która oprócz klasteryzacji najbardziej przypadła mi do gustu podczas zajęć. Początkowo chciałem też zaimplementować osobny model klasteryzacji, który grupowałby predykowane ceny w grupy (kawalerki, mieszkanie dla par bezdzietnych, mieszkanie dla rodziny, itd.). Niestety nie osiągnąłem wyników wystarczająco zadowalających jednak na końcu sprawozdania postaram się opisać kroki, które podjąłem w tym celu.

Update: Początkowo sprawozdanie miało mieć formę „tutoriala”, w którym chciałem pokazać krok po kroku jak sam uczyłem się różnych metod i funkcji Azure oraz na bieżąco w kolejnych punktach ulepszać model, natomiast ostatecznie takie sprawozdanie byłoby zbyt długie. Sam jestem zwolennikiem materiałów rzeczowych, a nie treściwych, więc właśnie w taki sposób postaram opisać mój projekt.

W kolejnych rozdziałach postaram się więc poruszyć najbardziej esencjonalne kwestie projektu takie jak:

- Krótki opis modelu regresji
- Wstępny przegląd i przygotowanie danych
- Utworzenie projektu w Azure ML
- Najprostszy model – największy błąd predykcji
- Zastosowanie zaawansowanych metod uczenia maszynowego
- Przedstawienie wyników i konkluzje
- Podsumowanie

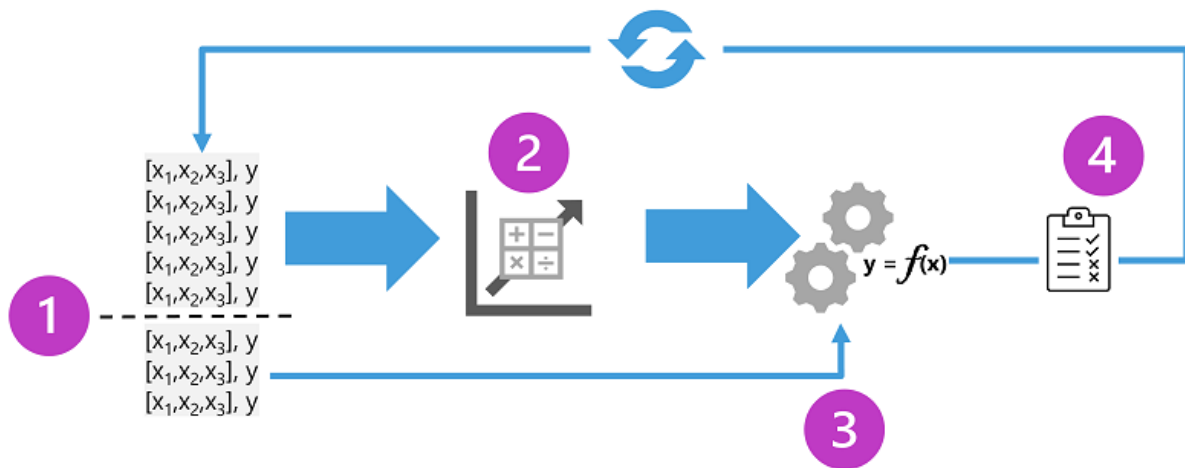
Krótki opis modelu regresji

Jeśli mowa o projekcie opartym o model regresji, to pasowałoby najpierw podać definicję tego terminu. Model regresji zgodnie z definicją (wziętą oczywiście z Wikipedii) jest techniką stosowaną w statystyce i uczeniu maszynowym, która służy do modelowania zależności między zmienną objaśnianą (zwaną też zmienną zależną lub celem) a jedną lub większą liczbą zmiennych objaśniających (zwanych też cechami lub predyktorami). W przypadku regresji **wynikiem jest ciągła wartość liczbowa**. Wyróżniamy wiele rodzajów regresji wśród których są m.in.:

- Regresja liniowa, zakładająca liniową zależność między zmiennymi objaśniającymi a zmienną objaśnianą,
- Regresja wielomianowa, rozszerzająca regresję liniową uwzględniając wyższe stopnie zmiennych, co pozwala na modelowanie bardziej złożonych zależności nieliniowych,
- Regresja logistyczna, która stosowana jest tak naprawdę do klasyfikacji,
- Regresja ridge, lasso, elastic net, itd...

Jednoznaczny(?) wybór modelu regresji w moim projekcie zostanie pokazany dopiero na końcu sprawozdania, kiedy dane zostaną odpowiednio przygotowane. Jednakże już na wstępie chciałbym zaznaczyć, że fraza „jednoznaczny wybór” nie jest zbyt odpowiednia, ponieważ uczenie maszynowe jest procesem ciągłym, a sama poprawa czy pogorszenie danych może tak naprawdę wpływać na werdykt a propos wyboru odpowiedniego algorytmu. W Azure ML istnieją dedykowane funkcje które dla przygotowanych danych potrafią dobrać najbardziej optymalny algorytm, ale o tym na końcu.

Choć istnieje wiele modeli regresji, to wszystkie mają pewne cechy wspólne. Po pierwsze zawsze są podzbiorem uczenia nadzorowanego, co oznacza że w samych danych używanych do treningu istnieje wartość wynikowa nazywana *labelem*. Po drugie wartość wynikowa jest wartością liczbową, co jest dużym ułatwieniem przy analizie wyników (np. jeśli wartością wynikową jest cena mieszkania, a analizy mówią o błędzie statystycznym na poziomie 50 000, to ta wartość jest tak naprawdę błędem w predykcji ceny). Najprostszy model regresji można zobrazować następująco:



1. Dzielimy dane losowo tak aby otrzymać zbiór danych treningowych oraz danych, które posłużą do walidacji trenowanego modelu.
2. Używamy algorytmu regresji tak aby otrzymać pewną predykcję
3. Wykorzystujemy dane walidacyjne z drugiego zbioru do przetestowania modelu
4. Porównujemy znane rzeczywiste rezultaty (*labele*) w zestawie danych walidacyjnych z rezultatami (*labelami*), które przewidział model

Jest to najprostszy model regresji, który zakłada minimalną ingerencję w dane. W tym modelu dane są doskonałe (brak wartości odstających, wszystkie pola uzupełnione, kolumny idealnie dobrane dla predykcji, brak niepotrzebnych kolumn, itd..), co jak wiemy w praktyce się nie zdarza. To właśnie taki model zastosuję w pierwszej fazie, aby zobaczyć jak duży błąd statystyczny w predykcji ceny nieruchomości przewidzi model i mieć później (po znacznej poprawie danych) punkt odniesienia (jak bardzo poprawa danych treningowych wpłynęła na zmniejszenie błędu predykcji).

Co bardzo ważne – na obrazku przy cyferce „1” mamy oznaczenia takie jak x_n oraz y . wartości x_n są wartościami na podstawie których budowany jest model predykcji, natomiast y jest labeliem czyli wartością przewidywaną. Nie można w przypadku regresji używać, czy nawet w jakiś sposób (np. poprzez Feature Engineering) operować na labelu przy budowie modelu! Wstyd się przyznać, ale sam zrobiłem ten błąd, gdy pierwszy raz po zajęciach bawiłem się z modelem. Mianowicie wykorzystałem label w ten sposób, że podzieliłem cenę mieszkania (label) przez metraż mieszkania, co miało pomóc modelowi w zauważeniu korelacji między ceną za metr i metrażem (w przypadku regresji modelowi o wiele łatwiej modeluje się na „mniejszych” liczbach takich jak cena za metr niż

na ogromnych liczbach rzędu miliona, jak w przypadku ceny nieruchomości). Wyniki oczywiście były doskonałe, błąd statystyczny bardzo niski, predykcja i dokładność na poziomie 99,7%. Było tak dobrze, że sam się zdziwiłem, bo wtenczas jeszcze niewiele zrobiłem na samych danych. Oczywisty błąd w tym podejściu zauważyłem przy walidacji, gdy w API walidacyjnym Azure musiałem podać... cenę mieszkania, aby z niej można było określić cenę za metr. Czyli musiałem podać dokładnie to, co chcę przewidzieć. Błąd w tym podejściu był oczywisty, ale przynajmniej wyniosłem z niego lekcję na przyszłość – w przypadku regresji nie można operować na labelu.

Wstępne przygotowanie danych

Aby zbudować jakiegokolwiek model potrzebujemy danych. W przypadku regresji ilość danych ma ogromne znaczenie – im więcej danych (aż do pewnej liczby) tym tak naprawdę lepiej, bo model może uczyć się lepiej i z mniejszym ryzykiem przeuczenia i można też wykorzystać więcej danych jako dane walidacyjne. W momencie dopierania tematu projektu wiedziałem że będę potrzebował danych, które zawierają ceny nieruchomości na terenie Krakowa. Na szczęście Kraków jest dużym miastem i ktoś już takie dane przygotował, a następnie opublikował je na serwisie Kaggle. Można je znaleźć pod nazwą „House Prices in Poland” lub pod poniższym linkiem:

<https://www.kaggle.com/datasets/dawidcegielski/house-prices-in-poland/data>

Rzućmy też szybko okiem na opis tego datasetu:

House Prices in Poland

Do EDA and Create Model that Predict the price of apartments.

Data Card Code (5) Discussion (0) Suggestions (0)

About Dataset

Context

The data has been pre-cleaned but still needs to be take care of. The data come from one of the websites, where we can find advertisements of the sale of apartments. The data is from February 2021.

Content

Description:

- address - Full address
- city - Warszawa (Warsaw), Kraków (Cracow), Poznań (Poznan).
- floor - The number of the floor where the apartment is located
- id - id
- latitude - latitude
- longitude - longitude
- price - Price of apartment in PLN [TARGET]
- rooms - Number of rooms in the apartment
- sq - Number of square meters of the apartment
- year - Year of the building / apartment















Już na tym etapie widzimy że mamy do czynienia z danymi cen nieruchomości dla trzech miast: Warszawy, Krakowa i Poznania, co od razu wskazuje pierwszą i najważniejszą operację do wykonania – przefiltrowanie cen wyłącznie dla

Krakowa. Kolumny jakie autor zamieścił w tym dataset na pewno nie są wyczerpujące, ale sensowne i szczerze mówiąc jestem z nich zadowolony. Spodziewałem się dużo gorszego datasetu i dużo gorzej przygotowanych danych. To bardzo dużo daje ponieważ najgorsze dane to te nieuporządkowane i nieuzupełnione, bo bardzo ciężko jest potem sensownie zastąpić dany wiersz, a w dużej ilości przypadków (kiedy dana kolumna jest predykcyjnie istotna) to taki wiersz jest tak naprawdę do wyrzucenia.

Przed rozpoczęciem tworzenia modelu predykcyjnego warto w ogóle spojrzeć na dataset, z jakimi zmiennymi mamy do czynienia, czy dane kolumny będą predykcyjnie istotne, zobaczyć rozkład konkretnych wartości czy nawet wykonać pewne operacje na danych poza Azure ML (w moim przypadku wszystkie operacje na danych będą wykonywane wyłącznie w Designerze Azure ML). Spójrzmy jak prezentuje się część danych pobranego datasetu (kolejna strona).

Jak widać na zrzucie ekranu już kilka pierwszych wierszy sporo mówi nam o możliwych lub nawet koniecznych operacjach do wykonania. Przede wszystkim trzeba będzie

- przefiltrować dane tak, aby widniały wyłącznie dla Krakowa,
- „Poobcinać” dane tak, aby nie zawierały wartości odstających – jak widać na obrazku na kolejnej stronie dane zostały przykładowo posortowane malejąco wg kolumny „sq” (metrażu lokalu). Zauważyć można wartości nie tylko duże (np. rzędu 300, 400, 500...), które oznaczają mogą lokale handlowe do wynajęcia, ale również wartości bardzo duże (rzędu paru tysięcy), które oznaczają mogą całe pawilony, oraz wartości wręcz nierealne (rzędu miliona) które prawdopodobnie zostały wprowadzone przypadkowo, lub w ogóle są błędem. Tego typu wartości odstające istnieją oczywiście nie tylko pod kolumną „sq”, ale pod każdą kolumną typu numerycznego
- Pogrupować i uporządkować dane – istnieją w datasecie dane, które bez poprawnego pogrupowania/uporządkowania nie tylko nie wniosą nic sensownego do modelu, ale mogą nawet zaszkodzić w poprawnej predykcji...

 Cracow Houses	 Houses	 Houses_After_Cleaning								
 house_id	 address	 city	 floor	 id	 latitude	 longitude	 price	 rooms	 sq	 year
<div><div>5198</div><div>Prądnik Biały Tonie</div></div>	10798 Wawer ul. Przyłuszczkowa	Kraków	2	22229 50.1121816	19.8994401	1 007 185,00 zł	1 007 185,00 zł	5	1007185	2020
	18637 Stare Miasto Naramowice	Warszawa	1	12896 52.2319581	21.0067249	389 880,00 zł	389 880,00 zł	2	9000	2022
	5734 Krowodrza Armii Krajowej	Poznań	1	1517 52.4006632	16.9197325917	544 169,00 zł	544 169,00 zł	4	8065	2021
	13662 Krowodrza Jana Buszka	Kraków	4	19707 50.0700650999	19.8979885774	6 299 000,00 zł	6 299 000,00 zł	10	442,2	2017
	13652 Bronowice ul. Stanisława Przybyszewskiego	Kraków	4	30031 50.067447	19.9029341	6 299 000,00 zł	6 299 000,00 zł	10	442,2	2017
	18636 Nowe Miasto Rataje	Kraków	3	28533 50.0469432	19.9971534358	6 400 000,00 zł	6 400 000,00 zł	8	441	2020
	21860 Żoliborz Stary Żoliborz Bitwy pod Rokitną	Poznań	1	4736 52.4006632	16.9197325917	347 776,00 zł	347 776,00 zł	2	379,52	2021
	11229 Rembertów	Warszawa	1	6550 52.2319581	21.0067249	9 000 000,00 zł	9 000 000,00 zł	6	368	2001
	18488 Mokotów Fort Piłsudskiego	Warszawa	0	8043 52.2614149	21.1628191	1 390 000,00 zł	1 390 000,00 zł	6	360	2004
	9464 Mokotów Ksawerów Ludwika Idzikowskiego	Warszawa	0	11202 52.2319581	21.0067249	7 414 000,00 zł	7 414 000,00 zł	5	337	2019
	22976 Mokotów Ksawerów	Warszawa	2	10821 52.2319581	21.0067249	6 820 000,00 zł	6 820 000,00 zł	5	336	2018
	19202 Podgórze Płaszowska	Warszawa	3	18203 52.181083	21.0069622	8 996 130,00 zł	8 996 130,00 zł	7	333,19	2020
	7012 Mokotów Karola Chodkiewicza	Kraków	0	22696 50.0452951	19.9748057	2 400 000,00 zł	2 400 000,00 zł	10	325	1940
	16331 Mokotów	Warszawa	6	13878 52.2319581	21.0067249	9 000 000,00 zł	9 000 000,00 zł	5	315,06	2010
	22754 Śródmieście Śródmieście Południowe Aleja Jana Chrystiana Szucha	Warszawa	7	10912 52.1939874	21.0457809	8 800 000,00 zł	8 800 000,00 zł	5	315	2010
	7092 Śródmieście Powiśle	Warszawa	6	15897 52.2173857	21.0238869	5 550 000,00 zł	5 550 000,00 zł	7	313,18	1910
	3582 Mokotów ul. Adama Naruszewicza	Warszawa	3	7475 52.2427517	21.0240188	7 000 000,00 zł	7 000 000,00 zł	6	304	2014
	9515 Stare Miasto Długa	Warszawa	8	13792 52.19103945	21.0174490384	4 500 000,00 zł	4 500 000,00 zł	4	303	2012
	9362 Ursynów Wyzółki	Kraków	3	26340 50.0469432	19.9971534358	1 990 000,00 zł	1 990 000,00 zł	4	301	1918
	18906 Mokotów	Warszawa	0	14563 52.1599734	20.9909161852	1 899 000,00 zł	1 899 000,00 zł	5	300	1996
	6761 Wilanów	Warszawa	6	11179 52.1939874	21.0457809	5 350 000,00 zł	5 350 000,00 zł	7	293,71	2016
	13117 Mokotów Cypryjska	Warszawa	0	11195 52.1530829	21.1104411	3 250 000,00 zł	3 250 000,00 zł	5	292,19	2008
	7 Mokotów Pory	Warszawa	5	14014 52.2319581	21.0067249	2 400 000,00 zł	2 400 000,00 zł	5	281,3	2004
	1268 Mokotów Stegny Pory	Warszawa	10	13308 52.1840585	21.044302	2 890 000,00 zł	2 890 000,00 zł	6	280	2003
	17766 Stare Miasto Kazimierz Skawińska	Warszawa	10	15088 52.2319581	21.0067249	2 900 000,00 zł	2 900 000,00 zł	5	280	2003
	129 Śródmieście	Kraków	3	18356 50.047494	19.9419376	4 300 000,00 zł	4 300 000,00 zł	6	275	2011
	5820 Łągowniki-Borek Fałęcki Borek Fałęcki Ogrodniki	Warszawa	10	13009 52.2328098	21.019067	9 506 700,00 zł	9 506 700,00 zł	5	271,6	2021
	12968 Śródmieście Złota	Kraków	1	19267 50.0469432	19.9971534358	550 000,00 zł	550 000,00 zł	2	271	1960
	11683 Żoliborz Stary Żoliborz park Żołnierzy Żywiela	Warszawa	9	6506 52.2309554499	21.0011759913	15 000 000,00 zł	15 000 000,00 zł	5	262	2017
	5779 Zwierzyniec Józefa Korzeniowskiego	Warszawa	4	16021 52.2319581	21.0067249	6 500 000,00 zł	6 500 000,00 zł	5	262	2005
	13775 Zwierzyniec Wola Justowska Józefa Korzeniowskiego	Kraków	0	29236 50.0621921	19.8898476	2 800 000,00 zł	2 800 000,00 zł	6	260	1980
	17282 Wilanów Piechoty Łanowej	Kraków	0	24166 50.0621921	19.8898476	2 800 000,00 zł	2 800 000,00 zł	10	260	2020
	117 Wilanów Piechoty Łanowej	Warszawa	2	9760 52.1728323	21.0851058	4 350 000,00 zł	4 350 000,00 zł	6	259,21	2020
	7856 Wola Mirów	Warszawa	1	9761 52.1728323	21.0851058	4 150 000,00 zł	4 150 000,00 zł	6	257,92	2020
	6680 Mokotów	Warszawa	10	11360 52.2393149	20.9858382	2 599 000,00 zł	2 599 000,00 zł	6	250,3	1998
	3543 Śródmieście Powiśle Drewniana	Warszawa	3	11962 52.1939874	21.0457809	2 950 000,00 zł	2 950 000,00 zł	5	250	2010
	4671 Grunwald Łazarz	Warszawa	5	9096 52.2391222	21.0245637	10 950 000,00 zł	10 950 000,00 zł	5	249	2011
	21193 Wilanów Franciszka Klimczaka	Poznań	5	4352 52.40537	16.9023298890	536 800,00 zł	536 800,00 zł	10	244	1908
		Warszawa	3	10673 52.16364095	21.0819235538	3 960 000,00 zł	3 960 000,00 zł	5	240,28	2007
		Warszawa								

...mowa tutaj oczywiście o kolumnie „address”, w której występują zmienne typu string. Jest to ciekawy przypadek i warto przyjrzeć mu się bliżej. Jak widać na poprzednim obrazku praktycznie każdy wiersz ma swoją unikalną wartość w kolumnie „address”, ponieważ przechowywana jest tam nie tylko dzielnica ale też nazwa osiedla lub ulicy. Jako że mieszkania zazwyczaj będą występować na różnych osiedlach i dzielnicach to można byłoby przyznać, że w kolumnie tej występują wyłącznie wartości unikalne. Jednak są też przypadki, kiedy dwa lokale faktycznie znajdują się w tym samym miejscu, lub podana jest jedynie nazwa dzielnicy (przykładowo – Mokotów – na powyższym obrazku).

Pogrupujmy sobie adresy za pomocą poniższej kwerendy SQL

```
SELECT address
FROM Houses
WHERE city LIKE "Kraków"
GROUP BY address
ORDER BY address
```

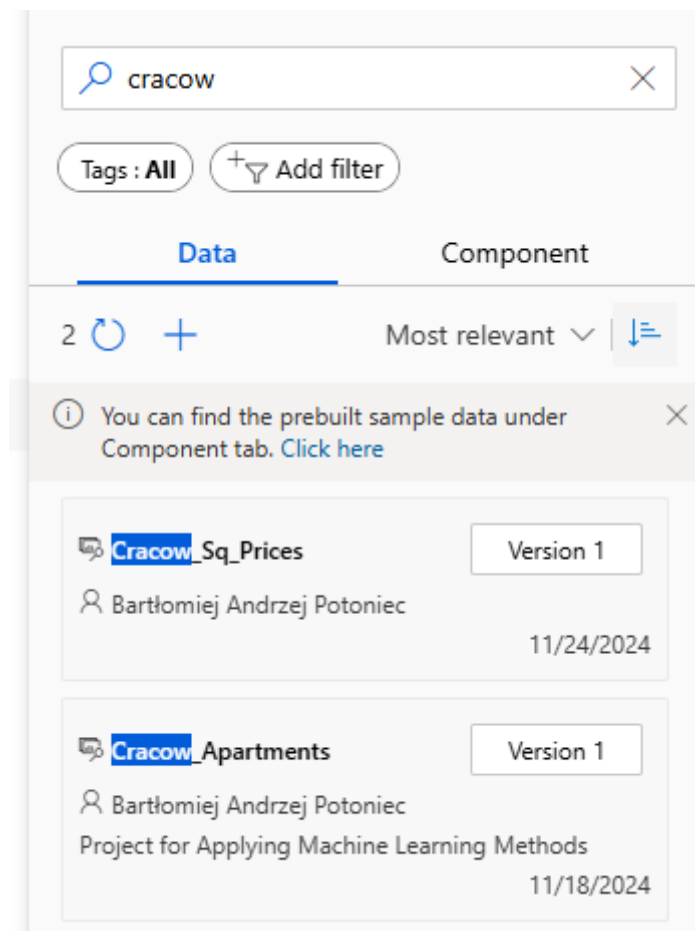
Na kolejnej stronie pokazano rezultat wykonania powyższej kwerendy wyłącznie dla miasta Kraków. Od razu rzuca się w oczy jak bardzo rozbieżne są wartości w tej kolumnie. Jednakże nie są to wartości w pełni unikalne (tzn. istnieją takie co najmniej dwa wiersze, które mają tę samą wartość w kolumnie „address”). W takim przypadku możemy określić wartości w tej kolumnie **zmiennymi kategorycznymi o wysokiej kardynalności**. Oznacza to, że liczba unikalnych wartości jest bardzo duża, potencjalnie bliska liczbie wierszy w danych (czyli każda wartość jest praktycznie unikalna). Natomiast to co jest ciekawsze to fakt, że jeśli dany wiersz zawiera jakąkolwiek wartość w tej kolumnie (brak pustej wartości w danym wierszu), to zawsze podana jest dzielnica niezależnie od tego czy występuje też nazwa ulicy lub osiedla. Ta obserwacja bardzo nam się przyda przy budowie modelu, a sposób wykorzystania tej kolumny (a raczej jej wartości) zostanie pokazany w kolejnych rozdziałach.

Jak pokazała praktyka tego typu analizy danych jeszcze przed budową modelu są bardzo przydatne i dzięki nim już na wstępie można określić kierunek i sposób budowy dobrego modelu predykcyjnego, co oczywiście przekłada się na wydajność i oszczędność czasu.

address
Bieńczyce
Bieńczyce Cienista
Bieńczyce Elżbiety Łokietkówny
Bieńczyce gen. Leopolda Okulickiego
Bieńczyce Kalinowe
Bieńczyce Kombatantów
Bieńczyce Na Lotnisku
Bieńczyce Okulickiego
Bieńczyce os. Kazimierzowskie
Bieńczyce os. Albertyńskie
Bieńczyce os. Dywizjonu 303
Bieńczyce os. Jagiellońskie
Bieńczyce os. Kalinowe
Bieńczyce os. Kazimierzowskie
Bieńczyce os. Kombatantów
Bieńczyce os. Na Lotnisku
Bieńczyce os. Niepodległości
Bieńczyce os. Przy Arce
Bieńczyce os. Słoneczne
Bieńczyce os. Strusia
Bieńczyce os. Teatralne
Bieńczyce os. Wysokie
Bieńczyce os. Złotej Jesieni
Bieńczyce Osiedle Górali
Bieńczyce Osiedle Jagiellońskie
Bieńczyce Osiedle Kalinowe
Bieńczyce Osiedle Kalinowe 20
Bieńczyce Osiedle Kazimierzowskie
Bieńczyce Osiedle Krakowiaków
Bieńczyce Osiedle Przy Arce
Bieńczyce osiedle Wysokie
Bieńczyce Planty Bieńczyckie
Bieńczyce Przy Arce
Bieńczyce ul. Fatimska
Bieżanów-Prokocim
Bieżanów-Prokocim Agatowa
Bieżanów-Prokocim Aleksandry
Bieżanów-Prokocim Barbary
Bieżanów-Prokocim Bieżanowska

Utworzenie projektu w Azure ML

Ten rozdział będzie bardzo krótki, ponieważ na samym wstępie pisałem, że nie będę rozpisywał się w jaki sposób obsługiwać oprogramowanie Azure ML. Jednakże warto zamieścić informacje o utworzeniu samego projektu. Dane zostały załadowane wraz ze wszystkimi kolumnami, które oryginalnie istnieją w bazie danych „House Prices in Poland” oraz określone jako tabelaryczne. Zostały również sprawdzone typy zmiennych na wypadek gdyby Azure nieprawidłowo (lub nie po naszej myśli) określił je za nas. Zrzut ekranu z overview załadowanych do Azure danych znajduje się na kolejnej stronie. Utworzyłem również Compute Instance, bez którego model nie zostanie uruchomiony. Na samym końcu stworzyłem w Designerze nowy Pipeline gdzie będzie projektowany cały model. Załadowane dane dostępne są od tej pory w przyborniku Designera w zakładce „Data” i można ich używać jak wszystkich dostępnych w Designerze bloków. Opisywane tutaj dane noszą nazwę „Cracow_Apartments” w Azure na zrzucie ekranu poniżej (te drugie też się przydadzą, opiszę je w późniejszym rozdziale).



Cracow_Apartments

Version: 1 (latest) ☆

Details Consume Explore Models Jobs

Refresh Generate profile

Preview Profile

Number of columns: 11 Number of rows: First 1000

Column1	address	city	floor	id	latitude	longitude	price	rooms	sq	year
0	Podgórze Zabłocie ...	Kraków	2.0	23918.0	50.0492242	19.9703793	749000.0	3.0	74.05	2021.0
1	Praga-Południe Gro...	Warszawa	3.0	17828.0	52.2497745	21.1068857	240548.0	1.0	24.38	2021.0
2	Krowodrza Czarno...	Kraków	2.0	22784.0	50.0669642	19.9200249	427000.0	2.0	37.0	1970.0
3	Grunwald	Poznań	2.0	4315.0	52.404212	16.882542	1290000.0	5.0	166.0	1935.0
4	Ochota Gotowy bu...	Warszawa	1.0	11770.0	52.212225	20.9726299	996000.0	5.0	105.0	2020.0
5	Nowa Huta Czyżyn...	Kraków	2.0	26071.0	50.0469432	19.997153435836697	414600.0	1.0	34.55	2022.0
6	Podgórze Płaszów ...	Kraków	0.0	22569.0	50.0498929	19.9906026	750000.0	4.0	81.4	2021.0
7	Mokotów Pory	Warszawa	10.0	13308.0	52.1840585	21.044302	2890000.0	6.0	280.0	2003.0
8	Ursynów Wyzyny	Warszawa	3.0	11387.0	52.1402821	21.0563452	615000.0	4.0	63.4	1982.0
9	Bemowo	Warszawa	1.0	10904.0	52.2389738	20.9132881	429000.0	1.0	40.0	1999.0
10	Śródmieście	Warszawa	0.0	16251.0	52.2328098	21.019067	375000.0	1.0	29.0	1968.0
11	Praga-Południe Go...	Warszawa	2.0	13355.0	52.2286331	21.1065722	520000.0	3.0	67.0	1989.0
12	Białoleka	Warszawa	0.0	15740.0	52.3196649	21.0211773	400000.0	3.0	57.0	2020.0
13	Nowe Miasto Malta...	Poznań	0.0	6081.0	52.3916079	16.99406261278815	421427.0	3.0	60.29	2019.0
14	Wola	Warszawa	0.0	12035.0	52.2362379	20.9547815	591771.83	2.0	52.27	2021.0
15	Grunwald Świerzaw...	Poznań	8.0	1085.0	52.4006632	16.91973259178088	547000.0	4.0	77.39	2020.0

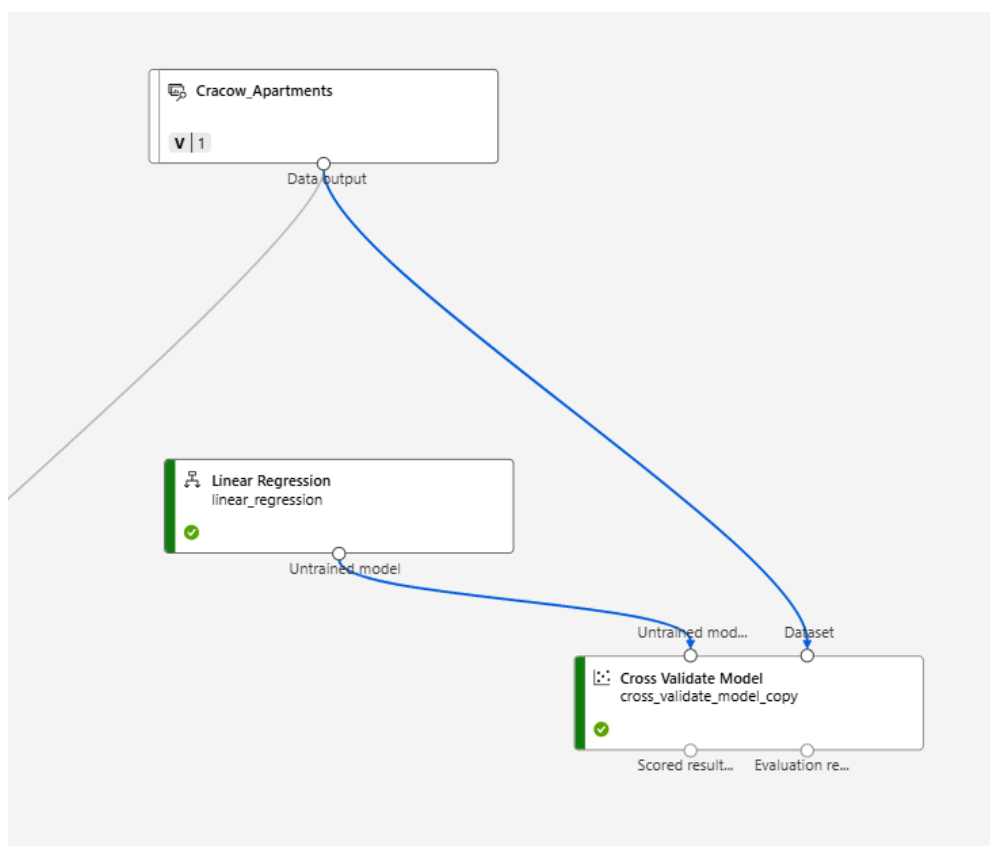
Najprostszy model – największy błąd predykcji

Skoro są już załadowane i dostępne w Designerze dane z naszego datasetu to można bezproblemowo zacząć tworzyć pipeline modelu predykcji. Pipeline to nic innego jak zbiór połączonych ze sobą blozków Azure ML wykonujących konkretne zadanie bazując na podanych parametrach (jeśli takowe są wymagane). Jednak na samym wstępie chciałbym pokazać jaki błąd predykcji przewidzi model jeśli na danych nie zostaną wykonane żadne operacje, o których pisałem wcześniej (filtrowanie, obcinanie wartości odstających, grupowanie, itd...).

Sam proces trenowania modelu nie jest trudny, zwłaszcza w Azure ML. Polega głównie na odpowiednim doborze algorytmu, który byłby najbardziej optymalny dla naszego modelu i zastosowanych danych (a i tutaj istnieją pewne bloki i techniki w Azure, które pomagają nam w tym doborze, ale o tym później).

Najbardziej pracochłonną i najtrudniejszą częścią tworzenia modelu ML jest część związana z pracą na samych danych – należy odpowiednio przygotować dane, o czym była już mowa wcześniej, i będzie w kolejnych rozdziałach.

Skoro proces trenowania to tak naprawdę ostatni i potencjalnie prosty etap, to spróbujmy zobaczyć jak duży błąd predykcji przewidzi model dla naszego datasetu. Najprostszy pipeline dla tego założenia pokazano na obrazku poniżej



Składa się on z datasetu, który jest rootem zawierającym dane wejściowe. Widać na obrazku, że od tego datasetu istnieje połączenie do bloku „Cross Validate Model”. Blok ten służy do oceny wydajności modelu poprzez przeprowadzanie walidacji krzyżowej. Wykonuje automatycznie trening modelu na różnych podzbiorach danych i testuje jego wydajność na pozostałych, pomagając zidentyfikować potencjalne problemy z przeuczeniem lub niedouczeniem. Wyniki są prezentowane w postaci metryk, które pomagają w wyborze najlepszego modelu. Lekką wadą jest, że ten blok z automatu trenuje model na dziesięciu foldach co może powodować problemy z uczeniem na małych zbiorach, ale podanie losowej wartości dla „Random Seed” powinno pomóc w doborze losowych danych (a tym samym w zapobieganiu przeuczenia). Jak widać na obrazku powyżej blok „Cross Validate Model” oprócz połączenia z datasetem oczekuje jeszcze jednego połączenia – „Untrained model” – tu oczywiście musimy załączyć odpowiedni algorytm. W moim przypadku oczywiście chodzi o „jakiś” algorytm regresyjny. W tym momencie (celowo) załączę najbardziej oczywisty i kojarzący się z regresją – algorytm regresji liniowej (Linear Regression), aby zobaczyć jakie wyniki otrzymamy przy najprostszych założeniach. Po wykonaniu obliczeń w programie Azure i przejściu do odpowiedniego taska klikamy prawym klawiszem myszy na blok, a następnie wybieramy Evaluation Results. Zobaczmy więc na poniższym obrazku jakie wyniki otrzymaliśmy.

Cracow_Apartments Completed

Evaluation_results_by_f

Rows 12 Columns 7

Fold Number	Number of examples in fold	Mean_Absolute_Error	Root_Mean_Squared_Error	Relative_Squared_Error	Relative_Absolute_Error	Coefficient_of_Determination
0	2377	526318.399231	15244584.033684	805.128515	1.868614	-804.128515
1	2377	197628.659106	386299.131534	0.637981	0.746364	0.362019
2	2376	199000.607917	419382.069328	0.602548	0.728819	0.397452
3	2376	194753.507844	352949.53297	0.59913	0.754814	0.40087
4	2376	195541.474915	425173.863106	0.49054	0.704444	0.50946
5	2377	212387.013395	422762.080175	0.672113	0.815888	0.327887
6	2376	197805.731766	389240.266766	0.550157	0.72158	0.449843
7	2376	214565.169449	443232.090252	0.660909	0.733389	0.339091
8	2377	212542.823329	529010.618719	0.728862	0.719516	0.271138
9	2376	185657.002711	339105.627243	0.515113	0.696321	0.484887
Mean	23764	233620.038966	1895173.931378	81.058587	0.848975	-80.058587
Standard Deviation	23764	103262.392158	4690800.570061	254.412251	0.359804	254.412251

Wyniki te oczywiście pochodzą z walidacji krzyżowej modelu. Zinterpretujemy je pokrótce:

- **Fold Number** oznacza numer folda danych użytych w walidacji krzyżowej. Dane są podzielone na kilka części (foldów), a każda część jest używana jako zestaw testowy, podczas gdy pozostałe foldy są używane do trenowania modelu.
- **Number of examples in fold** oznacza liczbę przykładów (rekordów) w danym foldzie. Jest to liczba próbek danych, które zostały użyte jako zestaw testowy w tej konkretnej iteracji.
- **Mean Absolute Error (MAE)** to średni błąd bezwzględny. Określa średnią różnicę między przewidywanymi wartościami modelu a rzeczywistymi wartościami. Mniejsza wartość oznacza lepsze dopasowanie modelu.
- **Root Mean Squared Error (RMSE)** jest to pierwiastek z średniego błędu kwadratowego. Oznacza miarę różnicy między wartościami przewidywanymi a rzeczywistymi, ale bardziej wrażliwa na większe błędy niż MAE, ponieważ błędy są kwadratowane przed ich uśrednieniem.
- **Relative Squared Error** to stosunek błędu kwadratowego modelu do błędu kwadratowego modelu bazowego (np. modelu przewidującego średnią wartość). Wartość bliższa 0 oznacza lepszą wydajność modelu w porównaniu z modelem bazowym.
- **Relative Absolute Error** jest stosunkiem błędu bezwzględnego modelu do błędu bezwzględnego modelu bazowego. Jak w przypadku poprzedniej kolumny, mniejsze wartości wskazują na lepszą wydajność modelu.
- **Coefficient of Determination (R^2)** oznacza współczynnik determinacji. Mierzy, jak dobrze model wyjaśnia zmienność danych. Wartości bliskie 1 wskazują na dobry model, natomiast wartości bliskie 0 oznaczają, że model słabo wyjaśnia zmienność. Ujemne wartości mogą wskazywać, że model jest gorszy od najprostszego modelu bazowego, który przewiduje średnią.

Jak widać predykcja modelu jest wręcz naganna. W przypadku walidacji crossowej odnosimy się do wartości „Mean” natomiast kolejne foldy i wartości ich kolumn nie są podane przypadkowo. Ogromną wskazówką przy optymalizacji i ulepszaniu modelu jest właśnie spojrzenie jak prezentują się kolejne foldy i jak bardzo różnią się ich wartości. Tutaj na pierwszy rzut oka widać, że różnice są bardzo duże. Najważniejszym składnikiem jest dla nas kolumna „Mean_Absolute_Error” ponieważ to ona wskazuje wprost na różnicę między realną wartością, a tą przewidzianą przez model – u nas różnice między

realną (podaną w dataset) wartością ceny mieszkania (labela) a tą predykcijną, przewidzianą przez model. Biorąc pod uwagę jedynie wartości Mean w przypadku predykcji tego modelu różnica między realną ceną mieszkania, a tą przewidzianą przez model (MAE) **to aż 233620** (ponad 233 tysiące złotych!), a w przypadku RMSE jest **to aż 1895173** (prawie 2 miliony złotych!). Przedstawmy sobie w tabeli wartości uśrednione dla predykcji tego niezoptymalizowanego i najprostszego modelu, tak aby można je później było porównać z modelem zoptymalizowanym.

Numer folda	Liczba rekordów w foldzie	MAE	RMSE	RSE	RAE	R ²
Mean	23764	~233620	~1895173	81,05	0,85	-80,05

Zastosowanie zaawansowanych metod uczenia maszynowego

W poprzednim rozdziale udowodniliśmy złudne przekonanie jakoby ML opierał się głównie na pisaniu algorytmów. Mit ten obala sam Azure ML, który algorytmy ma już zaimplementowane i wystarczy ich użyć jako bloczków (i ew. podać wymagane argumenty). Skoro już wiadomo, że najprostszy model nie jest tym najlepszym, i że praca w ML to głównie praca związana z ulepszaniem danych, to należy zbudować lepszy, który przewidzi jak najlepsze wyniki. Oczywiście nie oznacza to, że sam proces użycia algorytmów nie wymaga żadnej pracy, bo należy **wybrać ten odpowiedni dla naszego modelu** i potrafić go dostosować (np. za pomocą parametrów), aby jak najlepiej i najwydajniej współpracował z naszym modelem i również przedstawiał jak najlepsze wyniki. Jednakże proces ten wykonamy **na końcu**, gdy już będziemy mieć pewność, że nasze dane są odpowiednio zoptymalizowane.

W tym rozdziale przedstawię po kolei sposób, w jaki zadbałem aby predykcja cen nieruchomości (a raczej, jak już wiemy z poprzedniego rozdziału, różnica między cenami realnymi a predykcyjnymi) była jak najlepsza. Główną pracę jaką wykonam będzie polegała na **Feature Engineeringu** i różnym sposobom modyfikacji (lub dodania) danych, tak aby ostatecznie przedstawić wynik predykcji.

Pomimo, że Azure ML zawiera cały zestaw predefiniowanych bloków, to używać będę głównie bloków przeznaczonych do wykonania kodu Python (zwłaszcza przy Feature Engineering) oraz SQL, ponieważ jestem w trakcie nauki języka Python, a nie oszukujmy się, w ML znajomość Pythona i SQLa jest wręcz konieczna.

Początkowo rozdział ten chciałem potraktować jak „tutorial”, w którym krok po kroku chciałem wykonywać kolejne operacje na danych i regularnie przedstawiać wyniki. Postanowiłem jednak porzucić tę myśl, gdyż wtedy sprawozdanie byłoby zbyt obszerne. Wyniki więc przedstawię na końcu tego rozdziału, a kolejne operacje na danych będę po prostu szczegółowo opisywał.

1. Wybór rekordów zawierających dane dla miasta Kraków

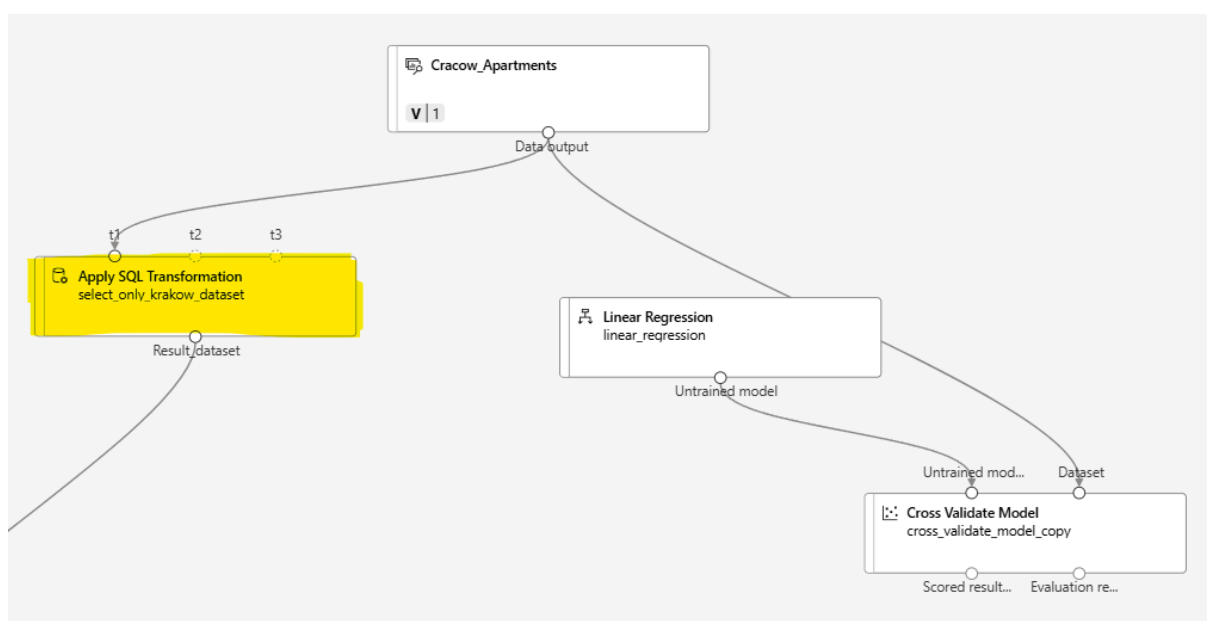
Jak wspomniałem na początku mój model dotyczył będzie wyłącznie cen mieszkań w Krakowie. Jak też wiemy z poprzednich rozdziałów dataset

„House Prices in Poland” zawiera dane dla miast: Warszawa, Kraków i Poznań. Należy więc oczywiście wybrać jedynie te dane, które są zdefiniowane dla miasta Kraków. Można oczywiście było (i to by było nawet lepsze) przefiltrować te dane nawet w programie Access jeszcze przed załadowaniem datasetu do Azure, jednak jak deklarowałem wcześniej, wszelkie operacje na danych w przypadku tego projektu chcę wykonać w Designerze Azure’a.

Aby wybrać dane wyłącznie dla miasta Kraków posłużę się blokiem „Apply SQL Transformation” wypisując następujące zapytanie SQL:

```
SELECT * FROM t1
WHERE t1.city LIKE “%Kraków%”
AND t1.year >= 1950
```

Natomiast tak w tej chwili wygląda Designer:



Na żółto zaznaczony jest omawiany blok. Blok „Cracow_Apartments” jest oczywiście naszym datasetem i pełni rolę roota. Po prawej stronie (pod rootem) widnieją bloki stworzone dla ukazania najprostszego modelu w poprzednim rozdziale. Jest to oczywiste – i w kolejnych przykładach nie będę już tego przypominał – że w dowolnym bloku kodu (np. Python lub SQL) „nieznane” nazwy kolumn lub zmiennych (np. tutaj „t1”) odnoszą się do węzłów wejściowych lub wyjściowych danego bloku w którym kod jest zdefiniowany.

Zobaczmy jakie wyniki zostaną przedstawione po wykonaniu tego bloku

Result_dataset										
Rows ⓘ		Columns ⓘ								
9066		11								
Column1	address	city	floor	id	latitude	longitude	price	rooms	sq	year
0	Podgórze Zabłocie Stanisława Klimickiego	Kraków	2	23918	50.049224	19.970379	749000	3	74.05	2021
2	Krowodrza Czarnowiejska	Kraków	2	22784	50.066964	19.920025	427000	2	37	1970
5	Nowa Huta Czyżyny ul. Woźniców	Kraków	2	26071	50.046943	19.997153	414600	1	34.55	2022
6	Podgórze Płaszów Koszykarska	Kraków	0	22569	50.049893	19.990603	750000	4	81.4	2021
26	Prądnik Czerwony ul. Śliczna	Kraków	1	20324	50.046943	19.997153	756707.4	4	82.43	2021
29	Prądnik Biały Henryka Pachońskiego	Kraków	6	22385	50.094977	19.920405	405900	2	41	2022
30	Dębniki Ruczaj prof. Michała Bobrzyńskiego	Kraków	4	30144	50.020627	19.89819	699000	2	60	2010
33	Podgórze Zabłocie	Kraków	3	27693	50.047877	19.961369	530000	1	31	2016
34	Podgórze Rzemieślnicza	Kraków	2	28526	50.046943	19.997153	529000	3	53.84	2021
38	Stare Miasto Piasek Łobzowska	Kraków	1	20681	50.071374	19.92809	620000	2	43	1990
41	Podgórze Duchackie Kurdwanów por. Halszki	Kraków	0	18497	50.005813	19.947709	159000	2	49	2000
42	Zwierzyniec Wola Justowska Jesionowa	Kraków	0	29381	50.067974	19.874854	899000	3	74.44	2020
43	Stare Miasto pl. Na Groblach	Kraków	0	26869	50.0561	19.932005	595000	1	31.24	2019
46	Podgórze Duchackie ul. Walerego Sławka	Kraków	6	24758	50.024231	19.959569	539000	3	57.81	2021
50	Grzegórzki ul. Fabryczna	Kraków	8	28842	50.058444	19.970235	971904	3	75.93	2021
51	Wzgórze Krzesławickie ul. Gustawa Morcinka	Kraków	0	26701	50.104781	20.036274	270648	2	37.59	2022
Dębniki Kliny-Żaciszka hna										

Jak widać otrzymaliśmy 9066 rekordów. To niestety może być zbyt mało aby naprawdę dobrze wytrenować model. Jednak temat ten poruszę w końcowym rozdziale, gdy będę przedstawiać konkluzje. Na tym etapie jednak wystarczy nam wiedza, że nie pracujemy już na głównym datasetcie (chyba że celowo podłączymy do niego jakiś blok), ale tym przefiltrowanym przez zapytanie SQL dla miasta Kraków.

2. Wstępne czyszczenie danych - obcinanie wartości odstających

Kolejnym etapem w naszym pipeline będzie wykonanie wstępnego czyszczenia danych, a mianowicie obcięcie wartości odstających. Robię to na tym etapie, ponieważ jest to konieczne dla kolejnych bloków – wartości odstające mogą wpłynąć na ich wyniki. Wartości odstające (outliers) z definicji to dane lub obserwacje, które znacznie odbiegają od pozostałych danych w zbiorze. Mogą one wynikać z błędów pomiarowych, błędów wprowadzania danych, wyjątkowych zdarzeń lub po prostu być rzadkimi przypadkami w danych, które z kolei mogą negatywnie wpłynąć na predykcję.

Konkretną kolumnę jaką miałem na myśli mówiąc o ewentualnym wpływie na kolejne bloki jest kolumna „sq” oznaczająca metraż lokali. Zobaczmy (w programie Access) jak prezentują się wartości odstające w tej kolumnie filtrując po mieście Kraków i sortując rosnąco a następnie malejąco po kolumnie „sq”

house_id	address	city	floor	id	latitude	longitude	price	rooms	sq	year	Kliknij, aby dodać
2155	Krowodrza ul. Juliusza Lea	Kraków	0	26812	50.0703596	19.9188768	229 000,00 zł	1	12	2018	
23628	Bieżanów-Prokocim Mała Góra	Kraków	0	25437	50.0114948	20.0245485	135 000,00 zł	1	13	2020	
8187	Prądnik Czerwony ul. Zenona Kł Kraków	Kraków	0	29290	50.0814925	19.9500226	210 994,00 zł	1	13,3	2021	
23094	Prądnik Czerwony ul. Zenona Kł Kraków	Kraków	1	26937	50.0814925	19.9500226	210 000,00 zł	1	13,3	2021	
12482	Śródmieście Zenona Klemensiev Kraków	Kraków	0	24664	50.0814925	19.9500226	208 899,00 zł	1	13,3	2021	
11972	Prądnik Czerwony ul. Zenona Kł Kraków	Kraków	0	29291	50.0814925	19.9500226	210 994,00 zł	1	13,3	2021	
195	Prądnik Czerwony ul. Zenona Kł Kraków	Kraków	1	26936	50.0814925	19.9500226	210 000,00 zł	1	13,3	2021	
2833	Śródmieście Aleja 29 Listopada	Kraków	2	21950	51.728259	18.5076219	206 111,00 zł	1	13,44	2021	
17077	Śródmieście	Kraków	2	26251	50.055	19.9430556	169 410,00 zł	1	13,44	2021	
23402	Stare Miasto św. Gertrudy	Kraków	2	19048	50.0594355	19.9419231	280 000,00 zł	1	14,03	1886	
6951	Stare Miasto	Kraków	1	28046	50.0590398	19.9371682979	184 470,00 zł	1	14,19	2021	
15363	Stare Miasto	Kraków	4	28129	50.0590398	19.9371682979	217 050,00 zł	1	14,47	2021	
1282	Stare Miasto	Kraków	4	28133	50.0590398	19.9371682979	217 050,00 zł	1	14,47	2021	
5737	Stare Miasto Żelazna	Kraków	4	28643	50.0745345	19.9464297	266 971,50 zł	1	14,47	2021	
23012	Stare Miasto	Kraków	4	25628	50.0590398	19.9371682979	217 050,00 zł	1	14,47	2021	
3913	Stare Miasto św. Gertrudy	Kraków	2	19054	50.0594355	19.9419231	290 000,00 zł	1	14,81	1886	
5142	Dębniki	Kraków	1	29859	50.0518611	19.9270367	239 000,00 zł	1	15	1927	
16771	Dębniki	Kraków	2	30036	50.0518611	19.9270367	239 000,00 zł	1	15,21	1927	
3956	Stare Miasto Żelazna	Kraków	4	28074	50.0745345	19.9464297	225 000,00 zł	1	15,25	2021	
12500	Prądnik Czerwony	Kraków	1	18295	50.0859418999	19.9660606081	232 543,00 zł	1	15,34	2021	
842	Prądnik Czerwony ul. Zenona Kł Kraków	Kraków	1	26868	50.0814925	19.9500226	227 430,00 zł	1	15,34	2021	
1302	Śródmieście Zenona Klemensiev Kraków	Kraków	1	21953	50.0814925	19.9500226	229 802,00 zł	1	15,34	2021	
426	Prądnik Czerwony ul. Zenona Kł Kraków	Kraków	0	29282	50.0814925	19.9500226	225 803,00 zł	1	15,38	2021	
3139	Stare Miasto Żelazna	Kraków	4	25707	50.0745345	19.9464297	296 460,75 zł	1	15,55	2021	
7114	Stare Miasto Żelazna	Kraków	4	23119	50.0745345	19.9464297	241 025,00 zł	1	15,55	2021	
17205	Śródmieście Zenona Klemensiev Kraków	Kraków	2	21952	50.0814925	19.9500226	228 224,00 zł	1	15,68	2021	
8339	Prądnik Czerwony ul. Zenona Kł Kraków	Kraków	1	26377	50.0814925	19.9500226	227 350,00 zł	1	15,69	2021	
17997	Śródmieście Zenona Klemensiev Kraków	Kraków	0	24663	50.0814925	19.9500226	218 564,00 zł	1	15,69	2021	
3945	Prądnik Czerwony ul. Zenona Kł Kraków	Kraków	0	22630	50.0814925	19.9500226	221 935,00 zł	1	15,69	2021	
326	Prądnik Czerwony ul. Zenona Kł Kraków	Kraków	0	26289	50.0814925	19.9500226	226 100,00 zł	1	15,74	2021	
20798	Prądnik Czerwony ul. Zenona Kł Kraków	Kraków	4	26953	50.0814925	19.9500226	242 500,00 zł	1	15,76	2021	
10758	Stare Miasto	Kraków	1	25630	50.0590398	19.9371682979	219 312,00 zł	1	15,95	2021	
13882	Stare Miasto	Kraków	3	28134	50.0590398	19.9371682979	239 250,00 zł	1	15,95	2021	
18007	Prądnik Czerwony ul. Zenona Kł Kraków	Kraków	5	22633	50.0814925	19.9500226	227 000,00 zł	1	16,04	2021	
14394	Stare Miasto Żelazna	Kraków	4	18706	50.0745345	19.9464297	240 600,00 zł	1	16,04	2021	
1444	Stare Miasto	Kraków	4	27958	50.0590398	19.9371682979	240 600,00 zł	1	16,04	2021	
3667	Stare Miasto	Kraków	4	28132	50.0590398	19.9371682979	240 600,00 zł	1	16,04	2021	
2343	Krowodrza Stanisława Konarskie Kraków	Kraków	0	27707	50.0840413	19.9781604675	200 000,00 zł	1	16,2	1930	
18	Krowodrza Stanisława Konarskie Kraków	Kraków	2	27706	50.0840413	19.9781604675	280 000,00 zł	1	16,2	1930	

W przypadku sortowania rosnąco można stwierdzić, że istnieje sporo wartości odstających. 12, 13, czy 14 m² to przecież zbyt mało na mieszkanie.. Jednak istnieją mini-kawalerki z takim metrażem i mało tego, obecnie są coraz bardziej popularne. **Dlatego ja nie potraktuję tych rekordów jako mających wartości odstające dla tej kolumny** – chcę aby mój model uwzględniał taki metraż zgodnie z trendem rynkowym.

house_id	address	city	floor	id	latitude	longitude	price	rooms	sq	year	Kliknij, aby dodać
6198	Prądnik Biały Tonie	Kraków	2	22229	50.1121816	19.8994401	1 007 185,00 zł	5	1007,185	2020	
5734	Krowodrza Armii Krajowej	Kraków	4	19707	50.0700650999	19.8979885774	6 299 000,00 zł	10	442,2	2017	
13662	Krowodrza Jana Buszka	Kraków	4	30031	50.067447	19.9029341	6 299 000,00 zł	10	442,2	2017	
13652	Bronowice ul. Stanisława Przyby	Kraków	3	28533	50.0469432	19.9971534358	6 400 000,00 zł	8	441	2020	
19202	Podgórze Płaszowska	Kraków	0	22696	50.0452951	19.9748057	2 400 000,00 zł	10	325	1940	
9515	Stare Miasto Długa	Kraków	3	26340	50.0469432	19.9971534358	1 990 000,00 zł	4	301	1918	
17766	Stare Miasto Kazimierz Skawinski	Kraków	3	18356	50.047494	19.9419376	4 300 000,00 zł	6	275	2011	
5820	Łagiewniki-Borek Fałęcki Borek f	Kraków	1	19267	50.0469432	19.9971534358	550 000,00 zł	2	271	1960	
13775	Zwierzyniec Wola Justowska Józ	Kraków	0	24166	50.0621921	19.8898476	2 800 000,00 zł	10	260	2020	
5779	Zwierzyniec Józefa Korzeniowski	Kraków	0	29236	50.0621921	19.8898476	2 800 000,00 zł	6	260	1980	
13462	Zwierzyniec Morelowa	Kraków	0	28572	50.0697051	19.8691042	3 500 000,00 zł	7	234	2008	
7139	Zwierzyniec Wola Justowska Mo	Kraków	0	29622	50.0469432	19.9971534358	3 500 000,00 zł	4	234	2016	
7366	Stare Miasto marsz. Józefa Piłsud	Kraków	4	30237	50.0600489	19.8471648	3 500 000,00 zł	5	232	1906	
10665	Stare Miasto Krupnicza	Kraków	3	29297	50.0469432	19.9971534358	5 750 000,00 zł	4	230	2008	
16002	Grzegórzki ul. Grzegórzecka	Kraków	9	28703	50.0587885	19.9500605	5 653 500,00 zł	6	226,14	2020	
6018	Grzegórzki Grzegórzecka	Kraków	9	25476	50.0587885	19.9500605	5 653 500,00 zł	6	226,14	2022	
6505	Grzegórzki	Kraków	10	29200	50.06497215	19.9688255164	5 653 500,00 zł	6	226,14	2021	
16953	Grzegórzki ul. Grzegórzecka	Kraków	7	28713	50.0587885	19.9500605	5 633 750,00 zł	5	225,35	2020	
2444	Grzegórzki	Kraków	7	29431	50.06497215	19.9688255164	5 633 750,00 zł	6	225,35	2020	
16920	Grzegórzki Olsza	Kraków	3	27975	50.0789773	19.9605879	1 050 000,00 zł	10	220	1980	
20601	Grzegórzki	Kraków	10	29208	50.06497215	19.9688255164	4 340 600,00 zł	5	217,03	2020	
3906	Krowodrza Oboźna(rejon)	Kraków	2	28581	50.45334045	30.3194026976	429 000,00 zł	7	207	1936	
5721	Dębniki Kliny-Zacisze Spacerowa	Kraków	1	23201	50.0050868	19.8770531	1 247 755,00 zł	7	204,55	2021	
16669	Stare Miasto ul. Wielopole	Kraków	5	30067	50.0583387	19.9464791	2 990 000,00 zł	7	201,58	1938	
179	Prądnik Czerwony Marka Eminow	Kraków	1	25162	50.0868616	19.9696987	1 045 000,00 zł	4	200	1984	
12335	Krowodrza	Kraków	0	29312	50.07087985	19.9165631181	1 650 000,00 zł	6	200	1919	
11265	Grzegórzki	Kraków	6	22457	50.06497215	19.9688255164	2 350 000,00 zł	7	195	2000	
18784	Grzegórzki	Kraków	6	24136	50.06497215	19.9688255164	2 350 000,00 zł	7	195	2000	
13280	Prądnik Czerwony	Kraków	4	23172	50.0859418999	19.9666006081	2 400 000,00 zł	5	194,1	2017	
9898	Prądnik Czerwony Reduta	Kraków	5	26231	50.1609054	19.7430891272	2 500 000,00 zł	5	194,1	2015	
10906	Prądnik Czerwony Reduta	Kraków	4	25900	50.1609054	19.7430891272	2 500 000,00 zł	5	194,1	2017	
20809	Prądnik Czerwony	Kraków	4	25586	50.0859418999	19.9666006081	2 500 000,00 zł	5	194,1	2017	
5254	Prądnik Czerwony Reduta	Kraków	4	26256	50.1609054	19.7430891272	2 500 000,00 zł	5	194,1	2017	
2736	Prądnik Czerwony Reduta	Kraków	3	26393	50.1609054	19.7430891272	2 400 000,00 zł	5	194	2017	
12059	Prądnik Czerwony Reduta	Kraków	4	26589	50.1609054	19.7430891272	2 500 000,00 zł	5	194	2017	
13472	Grzegórzki	Kraków	2	26219	50.06497215	19.9688255164	908 000,00 zł	1	190	1940	
4064	Grzegórzki	Kraków	2	24200	50.06497215	19.9688255164	908 000,00 zł	1	189	1950	
12750	Grzegórzki Kielecka	Kraków	2	26006	50.06497215	19.9688255164	909 000,00 zł	4	189	1940	
9381	Zwierzyniec ul. Wiosenna	Kraków	0	19335	50.0683302	19.8500249	1 500 000,00 zł	4	188,81	2020	

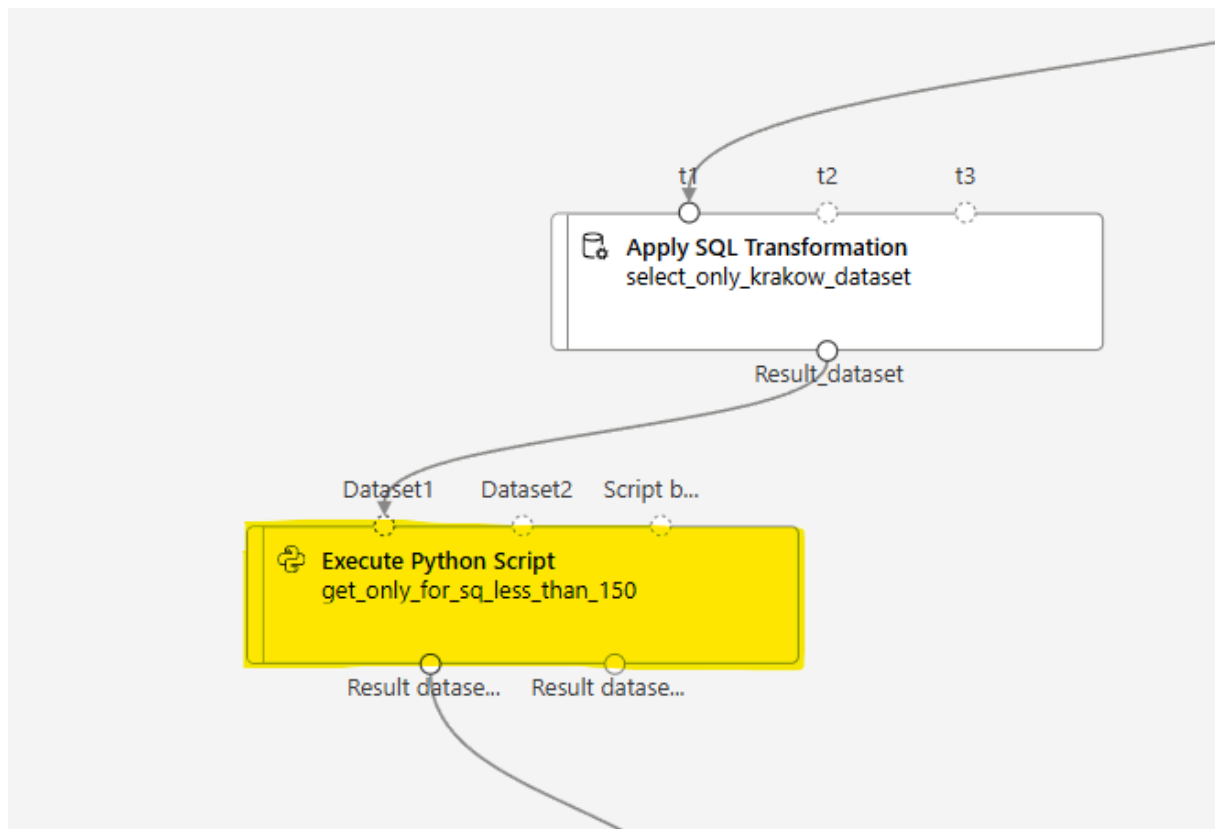
Natomiast w przypadku sortowania malejącego na pierwszy rzut oka widać jasne odstępstwa. Szczególnie pierwszy wiersz – ponad milion m² metrażu lokalu? Nawet jeśli założyć, że to prawda to cena za taki metraż jest kpina. Wychodzi na to, że za metr kwadratowy takiej nieruchomości zapłacimy około złotówkę. Ten rekord to książkowy przykład wartości odstającej i musi zostać (w moim projekcie) usunięty. Pozostałe (widoczne) rekordy są już bardziej wiarygodne patrząc na cenę, ale ich wysoki metraż trochę dziwi. Mieszkanie na 200, 300 czy 400 m² już ciężko nazwać mieszkaniem, a bardziej jakimś pałacem albo luksusową willą z basenem. Jednak istnieje jeszcze jedna opcja – ten dataset uwzględnia również lokale gospodarcze przeznaczone na prowadzenie np. działalności gospodarczej – i w ten sposób będę traktował lokale z tak wysokim metrażem w tym dataset. Jako że mój model ma działać prawidłowo dla cen **lokalu mieszkaniowych** to postanowiłem, że nie będę uwzględniał lokali gospodarczych. Poza tym umówmy się, rekordów dla metrażu 200, 300, czy 400 jest zbyt mało w dataset, aby model mógł na ich podstawie prawidłowo się wyuczyć. Po przejrzeniu danych i analizie postanowiłem, że obetnę dane do wartości „sq” mniejszej lub równej 150 m² – wartość ta zapewnia, że dane są spójne, a rekordów jest wystarczająco do prawidłowego wyuczenia modelu.

Aby obciąć dane wyłącznie dla wartości „sq” mniejszej lub równej 150 m² posłużę się blokiem „Execute Python Script” wypisując następujący kod Python:

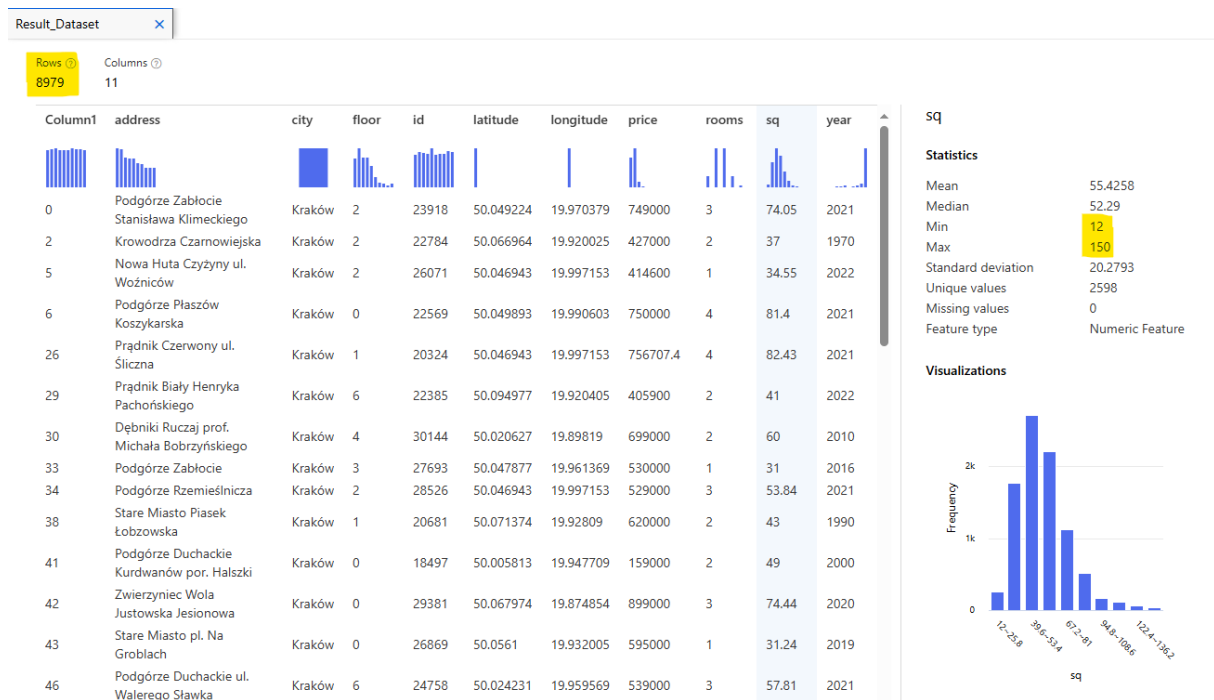
```
def azureml_main(dataframe1 = None, dataframe2 = None):
    df_filtered = dataframe1[dataframe1['sq'] <= 150]
    return df_filtered
```

Kod ten zawiera domyślną dla bloków w Azure funkcję *azureml_main*, która przyjmuje jako parametry *dataframe1* oraz *dataframe2* – te parametry odzwierciedlają dane, które łączymy do węzłów wejściowych bloku blokiem „Execute Python Script”. W następnej linijce wykonywana jest operacja filtrowania *dataframe1* – czyli danych z węzła, do którego dołączyliśmy wynik poprzedniego bloku – w taki sposób, że wybierane są tylko te wartości, których „sq” jest mniejsze lub równe 150 – czyli dokładnie tak jak założyliśmy. Na koniec wynik filtrowania przypisywany jest do tablicy *df_filtered* i cała przefiltrowana tablica jest zwracana na wyjście bloku jako rezultat.

W tej chwili Designer programu Azure ML wygląda następująco:



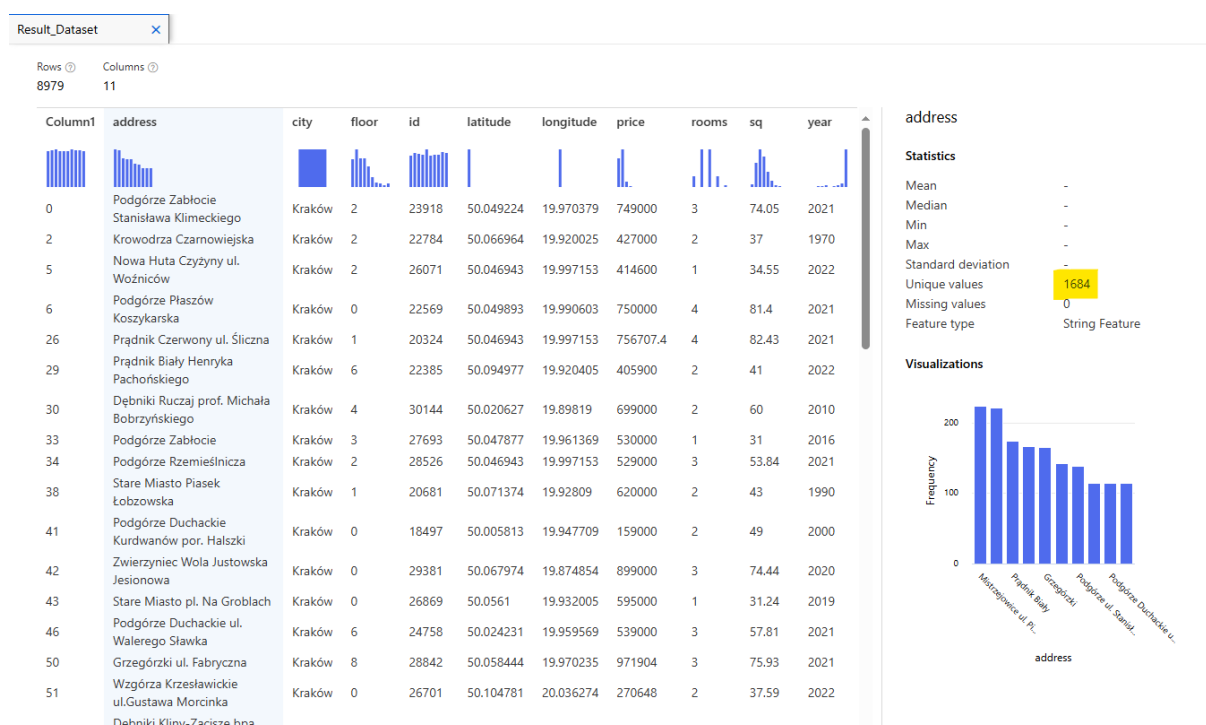
Na obrazku widać poprzednio przedstawiony i obecnie wykonany (na żółto) blok. Zobaczmy jakie wyniki zostaną przedstawione po wykonaniu tego bloku



Na obrazku widać, że blok kodu prawidłowo obciął nam wartości. Wartość „Min” dla kolumny „sq” wynosi 12, a „Max” dokładnie 150. Niestety liczba wierszy znów spadła tym razem do 8979, co może wpłynąć (i wpłynie) na uczenie modelu. Jednakże bardziej wyniki mogłyby „popsuć” wartości odstające, których właśnie się pozbyliśmy

3. Unifikacja dzielnic – wykorzystanie kolumny potencjalnie nieprzydatnej dla modelu

Jak opisywałem w rozdziale „Wstępne przygotowanie danych” istnieją kolumny w naszym dataset, których wartości są bardzo rozbieżne, a ich unikalność jest wysoka lub bliska liczbie wszystkich wierszy. Taką kolumną jest oczywiście „address”. Spójrzmy sobie jeszcze raz na wartości tej kolumny, ale tym razem już w programie Azure po wykonaniu poprzedniego pipeline’u.



W poprzednim rozdziale pisałem, że „[...] jeśli dany wiersz zawiera jakąkolwiek wartość w tej kolumnie (brak pustej wartości w danym wierszu), to zawsze podana jest dzielnica niezależnie od tego czy występuje też nazwa ulicy lub osiedla” – obserwację tę wykorzystał Azure grupując wartości kolumny „address”. W tym momencie możemy więc zobaczyć jaka jest liczba unikatów w tej kolumnie po wykonaniu poprzedniego pipeline’u – obecnie ta liczba wynosi 1684, a więc mniej niż badaliśmy przy okazji wstępnego przygotowania danych ale wciąż bardzo, bardzo dużo. Niestety to zbyt wiele aby model mógł w sensowny sposób zauważyć jakąś korelację między labellem (ceną), a tą kolumną – bo dokładnie o to nam chodzi.

Przypomnę jeszcze jedną, już ostatnią, obserwację którą omawialiśmy przy wstępnym przygotowaniu danych.

Mianowicie w tamtym rozdziale pisałem, że „[...] to co jest ciekawsze to fakt, że jeśli dany wiersz zawiera jakąkolwiek wartość w tej kolumnie (brak pustej wartości w danym wierszu), to zawsze podana jest dzielnica niezależnie od tego czy występuje też nazwa ulicy lub osiedla. Ta obserwacja bardzo nam się przyda przy budowie modelu, a sposób wykorzystania tej kolumny (a raczej jej wartości) zostanie pokazany w kolejnych rozdziałach.”

Obserwacja ta była kluczowa, ponieważ na tym etapie jest już oczywiste co trzeba zrobić aby:

1. Zmniejszyć liczbę unikatów w kolumnie „address”
2. Pomimo zmniejszenia liczby unikatów postarać się nauczyć model kojarzyć dane osiedle z ceną – bo o to nam chodzi

Jednak zanim to zrobimy jeszcze krótka dygresja. Czy dzielnica danego miasta ma jakiś wpływ na cenę lokalu? Oczywiście tak – w pobliżu centrum (Stare miasto, Piaski, itd.) ceny są zazwyczaj wyższe, natomiast osiedla bardziej oddalone (Nowa Huta, Prądnik, Bieżanów, itd.) charakteryzują się nieco mniejszą ceną. Ważniejszym pytaniem jest czy dzielnica danego miasta ma **kluczowy** wpływ na cenę? Oczywiście nie – na cenę mają wpływ indywidualne własności danego lokalu takie jak: typ budynku (apartament, kamienica, blok) metraż, wykończenie, wyposażenie, itd...

Tak więc już na tym etapie możemy się domyślić, że przyporządkowanie wierszy do kilkunastu dzielnic nie da nam jakichś wielkich korzyści, ale raz, może być przydatne, a dwa, dlaczego mielibyśmy marnować kolumnę „address” skoro można ją wykorzystać (a przynajmniej spróbować i zobaczyć jaki wpływ będzie miała na model, ale o tym później)?

Na podstawie analizy danych wyodrębniłem kilkanaście nazw dzielnic, które w Azure zostaną przydzielone do nowej kolumny (nie będziemy zastępować wartości oryginalnych):

1. Bieńczyce
2. Bieżanów
3. Bronowice
4. Czyżyny
5. Dębniki
6. Grzegórzki
7. Krowodrza
8. Łagiewniki
9. Mistrzejowice

10. Nowa Huta
11. Podgórze Duchackie
12. Podgórze
13. Prądnik Biały
14. Prądnik Czerwony
15. Stare Miasto
16. Swoszowice
17. Śródmieście
18. Wzgórza Krzesławickie
19. Zwierzyniec

Aby przyporządkować rekordom wartości dla nowej kolumny „district” posłużę się blokiem „Execute Python Script” wypisując następujący kod Python:

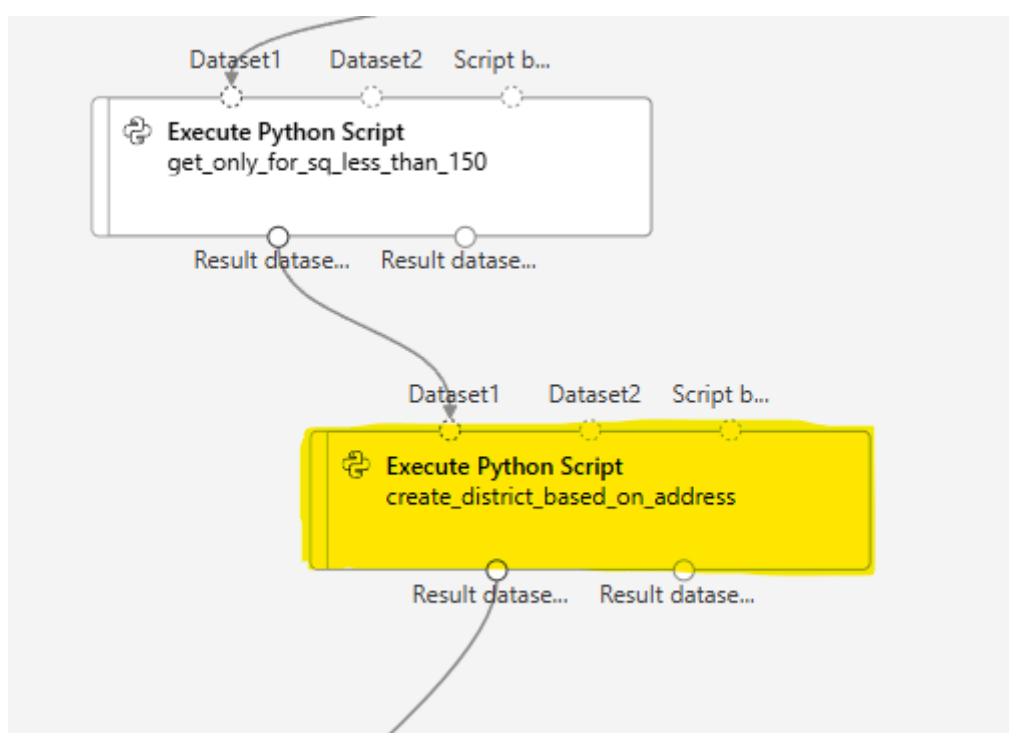
```
# Tablica dzielnic Krakowa
districts = [
    "Bieńczyce",
    "Bieżanów",
    "Bronowice",
    "Czyżyny",
    "Dębniki",
    "Grzegórzki",
    "Krowodrza",
    "Łagiewniki",
    "Mistrzejowice",
    "Nowa Huta",
    "Podgórze Duchackie",
    "Podgórze",
    "Prądnik Biały",
    "Prądnik Czerwony",
    "Stare Miasto",
    "Szoszowice",
    "Śródmieście",
    "Wzgórza Krzesławickie",
    "Zwierzyniec",
]

# Znajdź nazwę dzielnicy wewnątrz adresu
# - Jeśli nie można dopasować żadnej: przypisz "Unknown"
def extract_district(address, districts):
    for district in districts:
        if district in address:
            return district
    return "Unknown"

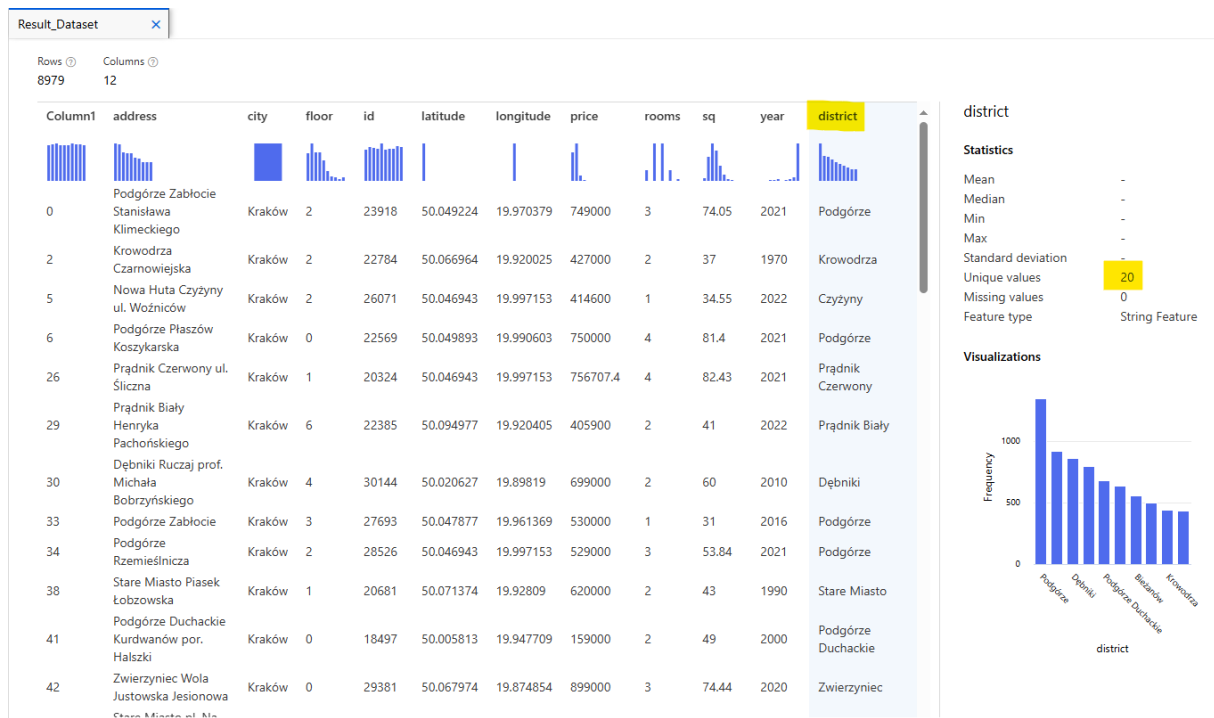
def azureml_main(dataframe1 = None, dataframe2 = None):
    dataframe1['district'] = dataframe1['address'].apply(lambda x:
        extract_district(x, districts))
    return dataframe1
```

Oczywiście wypisanie dzielnic w tablicy wewnątrz kodu to podejście celowo uproszczone ponieważ zakładam, że dataset nie będzie się rozszerzał i nie dojdą żadne nowe adresy. Kod zawiera funkcję *extract_district*, która analizuje adres i sprawdza, czy znajduje się w nim nazwa którejs z dzielnic. Jeśli tak, zwraca nazwę dzielnicy z tablicy, a jeśli nie, zwraca „Unknown”. Funkcja *azureml_main* modyfikuje przekazany do niej *dataframe1*, dodając nową kolumnę ***district***, która zawiera nazwę dzielnicy wyodrębnioną z kolumny *address* (a raczej z tablicy na podstawie analizy zawartości dzielnicy wewnątrz adresu) za pomocą funkcji *extract_district*. Jeśli nie można znaleźć pasującej dzielnicy, przypisuje wartość „Unknown”. Funkcja na koniec zwraca zaktualizowany DataFrame, czyli w praktyce nową kolumnę ***district***.

Zobaczmy jak obecnie wygląda nasz pipeline, bo dodaniu nowego bloku kodu:



Zobaczmy jak wygląda rezultat wykonania obecnego pipeline’u na obrazku na kolejnej stronie. Widzimy nowo utworzoną kolumnę ***district*** oraz wartości przypisane do niej czyli nazwy dzielnic, które zdefiniowaliśmy w tablicy w bloku kodu Python. Widać również informacje o tej kolumnie oraz jej unikatowych wartościach, których jest 20. Przy wypisywaniu dzielnic widzieliśmy, że wartości dzielnic jest 19, a nie 20. Cóż, należy pamiętać, że w kodzie uwzględniliśmy również przypadek kiedy żadna dzielnica nie odpowiada tej zdefiniowanej w kolumnie „address”. Wtedy do nowo utworzonej kolumny „district” miała być przypisana wartość „Unknown”



Taka wartość nic nie wniesie do naszego modelu. Zresztą nie chcemy, żeby model w jakikolwiek sposób kojarzył wartość „Unknown” z jakąkolwiek ceną. Jest to tylko informacja dla nas, że w jakimś konkretnym wierszu dzielnica nie została zdefiniowana (wartość pusta) lub została zdefiniowana nieprawidłowo (dziwny adres lub brak nazwy dzielnicy). Dlatego w późniejszym etapie przy końcowym oczyszczaniu wierszy postaramy się, aby wiersze zawierające wartość „Unknown” dla kolumny „district” zostały usunięte, bo mogą spowodować nieprawidłowe uczenie modelu. Taki zabieg niestety znów zmniejszy liczebność naszego datasetu, co niestety również będzie miało pewien wpływ na uczenie modelu - chcielibyśmy aby nasz dataset liczył jak najwięcej wierszy.

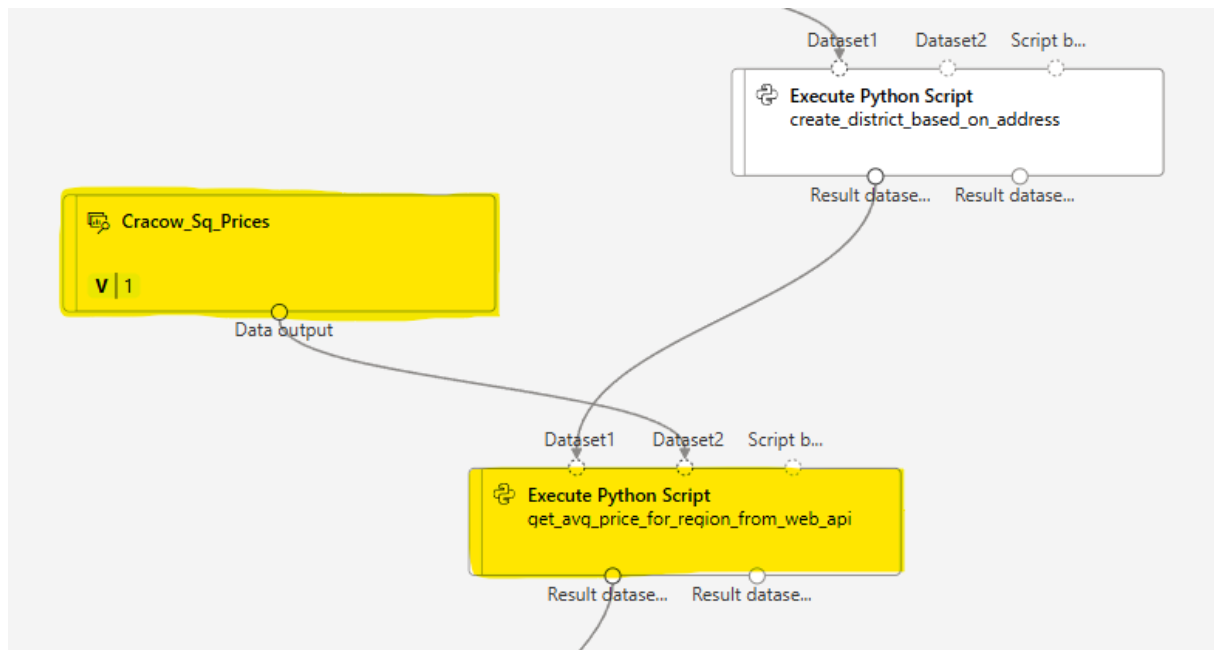
4. Podłączenie „Web API” w celu skojarzenia średniej ceny w danym regionie

Wiemy już na tym etapie, że w kolejnych blokach pipeline’u nasz dataset zostaje pomniejszany. Wiemy też, że nie mamy zbyt obszernego (po filtracji tylko dla miasta Kraków) datasetu. Wiemy także, że dataset (jego kolumny) są bardzo podstawowe. Jak jeszcze można wykorzystać Feature Engineering aby jak najlepiej wytrenować model? możemy w miarę możliwości maksymalnie wykorzystać istniejące kolumny, aby na ich podstawie zbudować nowe – na tym polega Feature Engineering i dokładnie to zrobiliśmy w poprzednim rozdziale – zbudowaliśmy nową kolumnę „district” na podstawie nie nadających się do uczenia adresów w kolumnie „address”. Skoro kolumny są raczej ubogie (powiedzmy podstawowe) i nie mamy zbyt dużego pola do popisu „wewnątrz” datasetu, to... może by spróbować pobrać dane z „zewnątrz”? Wiemy, że istnieją w naszym dataset kolumny „latitude” oraz „longitude” – wartości typu double, które kompletnie nic nie mówią modelowi oraz (jak sprawdziłem podczas budowania modelu) nic nie wnoszą do uczenia – ich wskaźnik przydatności wynosił zaledwie 0,8, co powoduje, że w praktyce musiałyby zostać wyrzucone przed rozpoczęciem trenowania. Jednakże, są to dane bardzo cenne – określają **lokalizację** nieruchomości w dataset. A co jeśli spróbowałibyśmy na podstawie lokalizacji wiersza pobrać ceny orientacyjne z regionu tej lokalizacji i wyliczyć ich średnią? Taka operacja na pewno sprawiłaby, że model lepiej kojarzyłby średnią cenę w danych regionie niż z kolumny „district” (tej z poprzedniego rozdziału).

Pytanie tylko skąd pobrać takie dane? Moim głównym zamiarem było pobranie danych cenowych na podstawie lokalizacji z jakiegoś zewnętrznego API – np. OtoDom. Niestety, tego typu rozwiązania są komercyjne i płatne, a samo korzystanie z dedykowanych połączeń sieciowych w Azure również kosztuje. Dlatego na potrzeby tego projektu przyjąłem, że moim „Web API” będzie plik tekstowy, a dokładnie zrzut z datasetu zawierający komplet (niefiltrowanych) wierszy z kolumnami „price_per_sq” – oznaczającą cenę na metr kwadratowy lokalu oraz współrzędne lokalizacyjne – latitude i longitude. Na kolejnej stronie przedstawiono zrzut ekranu paru pierwszych rekordów omawianego „Web API” w programie Access

Houses	×	Cracow_Houses	×	
price_per_sq	↕	latitude	↕	longitude
40119,05		50.0512149		19.9380389
34385,77		50.0469432		19.9971534358
31508,47		50.1035684		19.9536031063
29411,76		50.0561004		19.9320048
28022,47		50.0561004		19.9320048
28000		50.0561004		19.9320048
28000		50.0590398		19.9371682979
28000		50.0561004		19.9320048
26620		50.0679951		19.9383129908
26620		50.0590398		19.9371682979
26558,89		50.0469432		19.9971534358
26000		50.0588125		19.9333281
25925,93		50.0512149		19.9380389
25443,42		50.0528639		19.9282295
25377,36		50.0512149		19.9380389
25225,23		50.0518611		19.9270367
25200		50.0561004		19.9320048
25044,72		50.0561004		19.9320048
25024,85		50.0590398		19.9371682979
25000		50.0469432		19.9971534358
25000		50.0527679		19.9282413
25000		50.0587885		19.9500605
25000		50.0561004		19.9320048
25000		50.0587885		19.9500605
25000		50.06497215		19.9688255164
25000		50.06497215		19.9688255164
25000		50.0587885		19.9500605
24999		50.0518611		19.9270367
24999		50.0518611		19.9270367
24999		50.0527679		19.9282413
24999		50.0527679		19.9282413
24999		50.0518611		19.9270367
24999		50.0527679		19.9282413
24999		50.0527679		19.9282413
24999		50.0527679		19.9282413
24999		50.0527679		19.9282413
24992,43		50.0561004		19.9320048
24666,67		50.063859		19.8884689
23620		50.0590398		19.9371682979
23620		50.0679951		19.9383129908
23620		50.0679951		19.9383129908

Jest tylko jeden problem – jak zastosować tego typu „Web API” w naszym pipeline? Odpowiedź jest bardzo prosta – wystarczy dodać go jako nowe źródło danych, a następnie za pomocą bloku „Execute Python Script” wykonać odpowiednie operacje. Jak już wiemy blok „Execute Python Script” posiada dwa wejścia na dane (dataframe1 i dataframe2). Możemy więc do jednego wejścia podłączyć poprzednio wykonany blok w naszym pipeline, a do drugiego nasze nowe źródło danych. W takiej konfiguracji nasz pipeline będzie wyglądał następująco:



Po lewej stronie widnieje „Web API” (nowe źródło danych), w którym zawarte są informacje o współrzędnych geolokalizacyjnych danego lokalu oraz jego cena za metr kwadratowy. Poniżej natomiast widnieje blok „Execute Python Script” z następującym kodem:

```
import pandas as pd
import numpy as np
from numpy import radians, sin, cos, sqrt, arctan2

def haversine_vectorized(lat1, lon1, lats2, lons2):
    """
    Wektorowe obliczenie odległości Haversine między jednym punktem a
    wieloma innymi.
    """
    R = 6371 # Promień Ziemi w kilometrach
    lat1, lon1, lats2, lons2 = map(radians, [lat1, lon1, lats2, lons2])

    dlat = lats2 - lat1
    dlon = lons2 - lon1

    a = sin(dlat / 2.0)**2 + cos(lat1) * cos(lats2) * sin(dlon /
2.0)**2
```

```

    c = 2 * arctan2(sqrt(a), sqrt(1 - a))
    return R * c

def calculate_average_price(long, lat, reference_data,
    initial_radius=1, step=1, max_radius=100):
    """
    Oblicza średnią cenę w okręgu wokół podanych współrzędnych,
    iterując do skutku, aż znajdzie dane.

    Args:
    - long: długość geograficzna punktu (longitude).
    - lat: szerokość geograficzna punktu (latitude).
    - reference_data: DataFrame zawierający dane referencyjne z
    kolumnami 'longitude', 'latitude', 'price'.
    - initial_radius: początkowy promień okręgu (w km).
    - step: wartość o jaką zwiększamy promień, jeśli brak danych.
    - max_radius: maksymalny promień, do którego zwiększamy zakres (w
    km).

    Returns:
    - Średnia cena w okręgu.
    """
    current_radius = initial_radius

    while current_radius <= max_radius:
        # Oblicz odległości do wszystkich punktów w referencyjnych
        danych
        distances = haversine_vectorized(lat, long,
        reference_data['latitude'], reference_data['longitude'])

        # Filtruj punkty w promieniu
        filtered_data = reference_data[distances <= current_radius]

        # Jeśli znaleziono dane, oblicz średnią cenę
        if not filtered_data.empty:
            return round(filtered_data['price_per_sq'].mean(), 2)

        # Zwiększ promień, jeśli brak danych
        current_radius += step
    # Jeśli nie znaleziono danych nawet w maksymalnym promieniu
    return np.nan # Opcjonalnie można zwrócić wartość domyślną, np. 0
lub -1

def azureml_main(dataframe1, dataframe2):
    """
    Args:
    - dataframe1: Dane wejściowe (dane do analizy)
    - dataframe2: Dane referencyjne (zestaw cen mieszkań)

    Returns:
    - DataFrame z nową kolumną 'average_price_in_radius'
    """
    # Iteracyjnie oblicz średnią cenę dla każdej lokalizacji
    dataframe1['avg_price_per_sq'] = dataframe1.apply(
        lambda row: calculate_average_price(
            row['longitude'], row['latitude'], dataframe2
        ),
        axis=1
    )

```



```

    )

    dataframe1['avg_price'] = round((dataframe1['avg_price_per_sq'] *
dataframe1['sq']), 2)

    return dataframe1

```

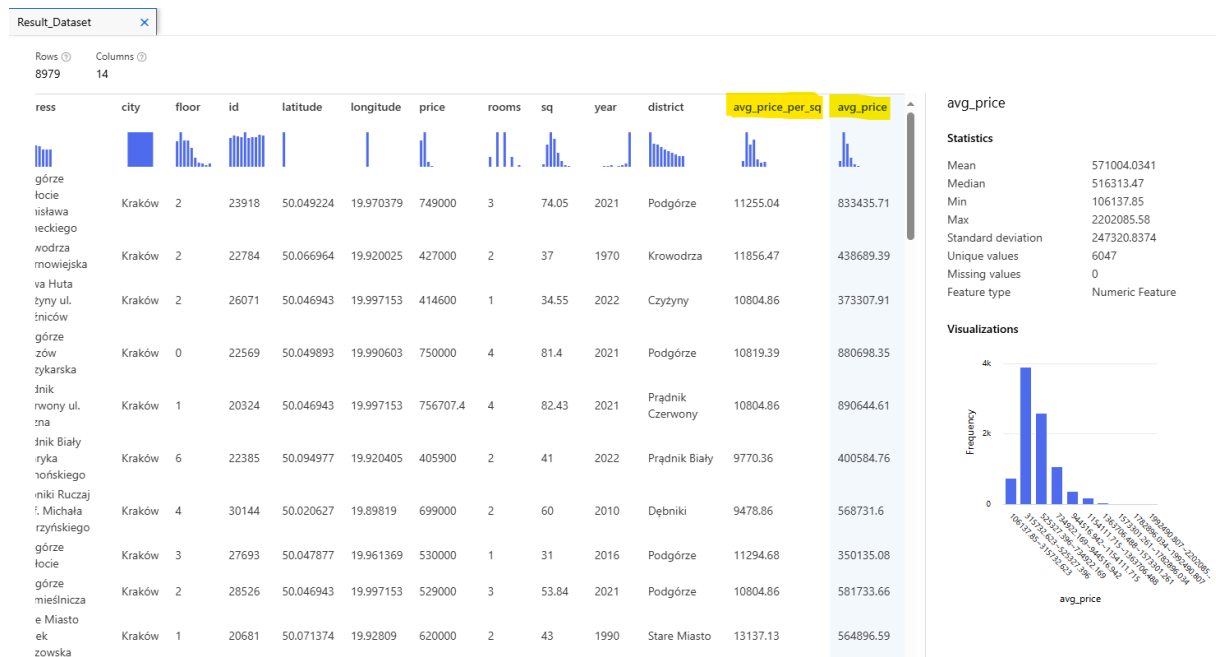
Kod ten importuje biblioteki pandas i numpy, które są niezbędne do pracy z danymi w postaci tabelarycznej oraz do wykonywania operacji matematycznych i obliczeń wektorowych. Następnie definiuje funkcję *haversine_vectorized*, która oblicza odległość w linii prostej (w kilometrach) między jednym punktem a wieloma innymi za pomocą wzoru Haversine. Funkcja ta przyjmuje jako argumenty współrzędne geograficzne jednego punktu (lat1, lon1) oraz wielu innych punktów (lats2, lons2), konwertuje te współrzędne na radiany i stosuje wzór Haversine do obliczenia odległości.

Kolejna funkcja, *calculate_average_price*, ma na celu obliczenie średniej ceny w określonym promieniu od podanych współrzędnych geograficznych. Funkcja ta przyjmuje długość i szerokość geograficzną punktu (long, lat), zbiór danych referencyjnych (reference_data), który zawiera informacje o lokalizacjach i cenach, oraz parametry określające początkowy promień (u nas 0,5 km), krok zwiększenia promienia (również 0,5 km) i maksymalny promień poszukiwań. Funkcja zaczyna od obliczenia odległości od danego punktu do wszystkich punktów w zbiorze danych referencyjnych za pomocą funkcji *haversine_vectorized*. Następnie filtruje dane, aby pozostawić tylko te punkty, które znajdują się w obrębie bieżącego promienia. Jeśli znajdzie jakiegokolwiek dane w tym promieniu, oblicza i zwraca średnią cenę za metr kwadratowy. Jeśli nie znajdzie danych, zwiększa promień i powtarza proces, aż do osiągnięcia maksymalnego promienia. W przypadku braku danych nawet po maksymalnym rozszerzeniu promienia funkcja zwraca wartość np. NaN (Not a Number).

Funkcja *azureml_main* jest punktem wejścia skryptu. Przyjmuje ona dwa DataFrame'y jako argumenty: *dataframe1*, który zawiera dane do analizy z naszego datasetu, oraz *dataframe2*, który zawiera dane referencyjne z cenami mieszkań – nasze „Web API”. W tej funkcji dla każdej lokalizacji w *dataframe1* wywoływana jest funkcja *calculate_average_price*, która oblicza średnią cenę na podstawie danych referencyjnych i zapisuje ją w nowej kolumnie *avg_price_per_sq*. Następnie obliczana jest całkowita średnia cena mieszkania, mnożąc średnią cenę za metr kwadratowy przez powierzchnię mieszkania (sq), i wynik zapisywany jest w kolumnie *avg_price*. Ostatecznie zaktualizowany

DataFrame, zawierający nowe kolumny ze średnimi cenami, jest zwracany jako wynik funkcji.

Nasuwa się pytanie dlaczego pisałem własną skomplikowaną logikę dla funkcji *haversine_vectorized* zamiast zaimplementować jakieś gotowe rozwiązanie, których w Pythonie nie brakuje. Otóż początkowo miałem zamiar zaimportować paczkę geopy ale niestety Azure nie posiadał jej domyślnie w swoim środowisku uruchomieniowym. Trzeba byłoby zainicjować nowy obraz dockera ze środowiskiem, które zapewnia geopy, co również wiąże się z kosztami. A oto jak prezentują się wyniki aktualnego pipeline'u:



Jak widać utworzone zostały 2 nowe kolumny: *avg_price* oraz *avg_price_per_sq* a to dlatego że chciałem wiedzieć czy model lepiej uczy się na danych jednostkowych (cena za metr) czy spójnych (cena za metr pomnożona przez metraż lokalu).

W taki sposób podłączyliśmy „Web API” do naszego pipeline’u symulując korzystanie z danych zewnętrznych do nauki modelu.

5. Konwersja roku budowy budynku na wiek budynku

Ostatnim z zagadnień Feature Engineering, które wykonamy to lekka modyfikacja kolumny „year” na „building_age”. Konwersja kolumny zawierającej rok budowy budynku na wiek budynku (liczbę lat od daty budowy do chwili obecnej) może poprawić wyniki modelu uczenia maszynowego ze względu na:

- **Znaczenie** - wiek budynku jest bardziej intuicyjną i bezpośrednią cechą, która może lepiej oddawać wpływ stanu budynku na jego wartość, koszt utrzymania, czy poziom zużycia. W przeciwieństwie do surowego roku budowy, wiek budynku bezpośrednio informuje, jak długo budynek był użytkowany, co może mieć większy wpływ na jego obecny stan i cenę.
- **Lepsza interpretacja dla modelu** - modele uczenia maszynowego mogą lepiej złapać związki między wiekiem a innymi cechami. Rok budowy jest liczbą absolutną, ale dla modelu predykcyjnego może być trudne do zinterpretowania w kontekście innych danych. Natomiast wiek jako wartość względna ułatwia zrozumienie relacji (np. starsze budynki mogą mieć inny profil cenowy niż nowe).
- **Unikanie problemów z czasem** - rok budowy jest zmienną, której znaczenie zmienia się w czasie. Np. „rok 2000” staje się mniej aktualny w miarę upływu czasu. Konwersja na wiek budynku pozwala uniknąć tego problemu i czyni ją bardziej uniwersalną i niezależną od bieżącego roku.
- **Redukcja wariancji** - przy odpowiednim przygotowaniu danych, wiek może zmniejszyć wariancję w danych, co może pomóc modelowi lepiej generalizować i poprawić jego wydajność.

Taką konwersję ponownie przeprowadzę poprzez blok „Execute Python Script” w naszym pipeline. Kod, który będzie potrzebny wygląda następująco:

```
import datetime
```

```
def azureml_main(dataframe1 = None, dataframe2 = None):
```

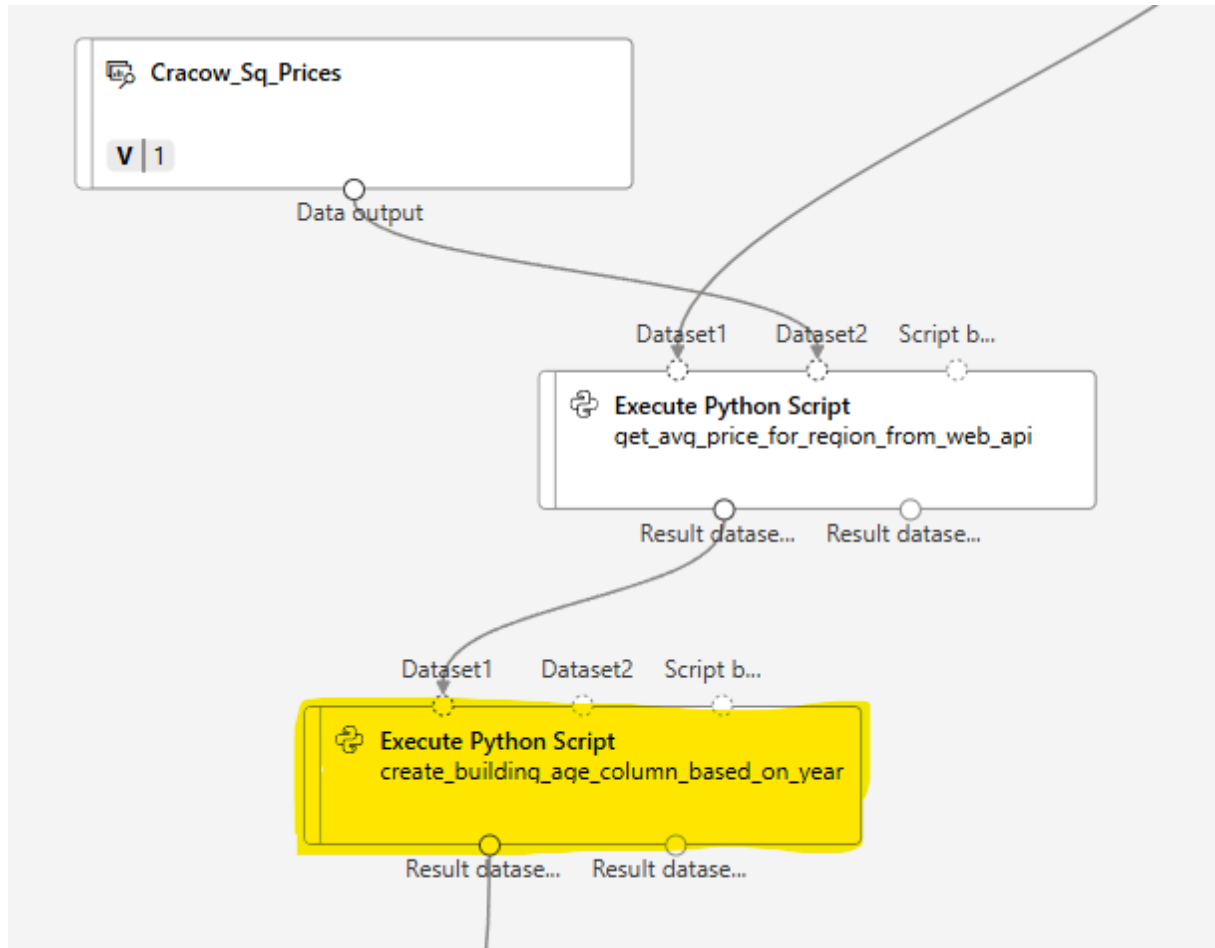
```
    today = datetime.date.today()
```

```
    dataframe1['building_age'] = today.year - dataframe1['year']
```

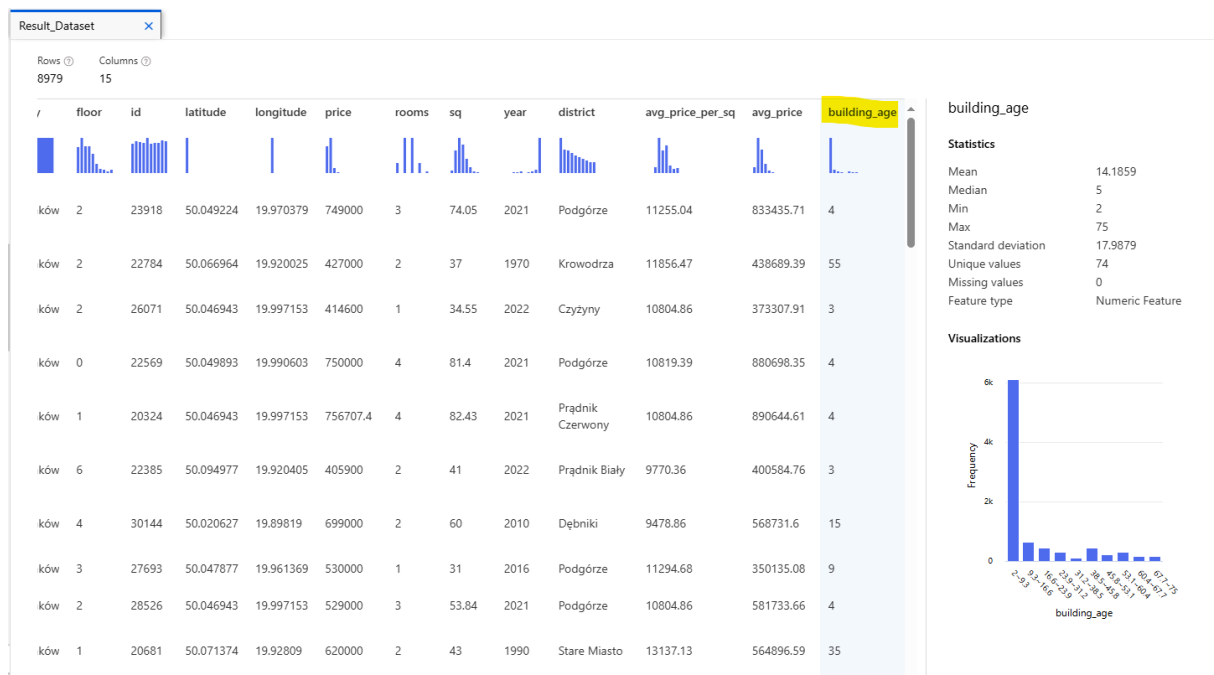
```
    return dataframe1,
```

Widzimy, że konwersja jest prosta, a polega raczej na utworzeniu nowej kolumny na podstawie istniejącej – ponownie robię to w ten sposób, aby nie nadpisywać oryginalnego datasetu.

Nasz pipeline wygląda w tym momencie jak na obrazku poniżej:



Zobaczmy jak prezentują się wyniki wykonania aktualnego pipeline'u na obrazku na kolejnej stronie. Jak widać Azure utworzył nową kolumnę „building_age” do której przypisał wartość różnicy aktualnej daty i roku budowy budynku.



6. Wybór odpowiednich kolumn i końcowe oczyszczanie danych

Podczas pracy wykorzystując Feature Engineering utworzyliśmy wiele nowych kolumn, które będą potencjalnie przydatne do uczenia modelu. Poza tym mamy jeszcze kolumny oryginalne, które były w datasecie od początku. Na obrazkach z wynikami w poprzednich rozdziałach mogliśmy obserwować, że liczba tych kolumn rośnie, a nie wszystkie będą przydatne modelowi (część z nich może nawet zaszkodzić). Wypiszmy sobie jakie aktualnie kolumny zawiera nasz model, i które będą potrzebne, a które nie.

1. **address** – ta kolumna była oryginalnie zawarta w datasecie. Pamiętamy, że na jej podstawie tworzyliśmy kolumnę „district”, ponieważ sama kolumna „address” zawierała wartości zmiennych kategorycznych o wysokiej kardynalności. Tej kolumny **nie użyjemy** do uczenia modelu ze względów, które zostały wyjaśnione w podrozdziale „Unifikacja dzielnic – wykorzystanie kolumny potencjalnie nieprzydatnej dla modelu”
2. **city** – kolumna ta również istnieje oryginalnie w dataset. Jak łatwo można się domyślić **nie użyjemy** jej w procesie uczenia, bo nie ma to najmniejszego sensu. Oryginalny dataset zawierał dane dla miast:

Kraków, Warszawa i Poznań. My rozważamy przypadki użycia jedynie dla miasta Kraków, co czyni tę kolumnę nieprzydatną modelowi.

3. **floor** – chociaż piętro, na którym znajduje się lokal może mieć naprawdę znikomy wpływ na cenę, to jednak zostawimy ją i **użyjemy** jej w procesie uczenia, aby przekonać się za chwilę jak prezentuje się jej wskaźnik przydatności.
4. **id** – kolumna zawierająca identyfikator wiersza w oryginalnym dataset. Nie ma żadnego powiązania z danymi, dla których budujemy model – **nie użyjemy** jej przy uczeniu modelu.
5. **latitude** – czyli kolumna zawierająca szerokości geograficzne danych nieruchomości. Jak pamiętamy na jej podstawie wczytywaliśmy dane z naszego podłączonego „Web API”. **Nie użyjemy** jej w procesie uczenia ze względu na przesłanki zaprezentowane w podrozdziale „Podłączenie „Web API” w celu skojarzenia średniej ceny w danym regionie”.
6. **longitude** – długość geograficzna danych nieruchomości. Ponownie **nie użyjemy** jej w procesie uczenia ze względu na przesłanki zawarte w rozdziale „Podłączenie „Web API” w celu skojarzenia średniej ceny w danym regionie”.
7. **price** – czyli nasz **label**. Z oczywistych względów **nie możemy użyć** kolumny wynikowej do uczenia modelu
8. **rooms** – liczba pokoi jak najbardziej ma wpływ na cenę lokalu, a ponadto jest związana z metrażem (im więcej pokoi tym potencjalnie większy metraż), co sprawia, że model również może zauważyć pewną korelację. **Użyjemy** tej kolumny w procesie uczenia.
9. **sq** – czyli metraż nieruchomości. Ma chyba największy wpływ na cenę lokalu. **Użyjemy** tej kolumny w procesie uczenia.
10. **year** – kolumna oznaczająca rok budowy nieruchomości, z której to przed chwilą budowaliśmy nową kolumnę „building_age”. Z racji budowy tej nowej kolumny **nie użyjemy** kolumny „year” w procesie uczenia.
11. **dictrect** – kolumna, która powstała na podstawie adresów, a która to służy do określenia dzielnicy, w której leży nieruchomość. **Użyjemy** jej w procesie uczenia, bo potencjalnie może mieć wpływ na model.
12. **avg_price_per_sq** i **avg_price** – kolumny oznaczające kolejno średnią cenę na metr w okolicy, w której znajduje się nieruchomość (na podstawie współrzędnych) oraz średnią cenę całkowitą nieruchomości. Zostały zwrócone przez nasze „Web API”. W modelu **użyjemy kolumny**

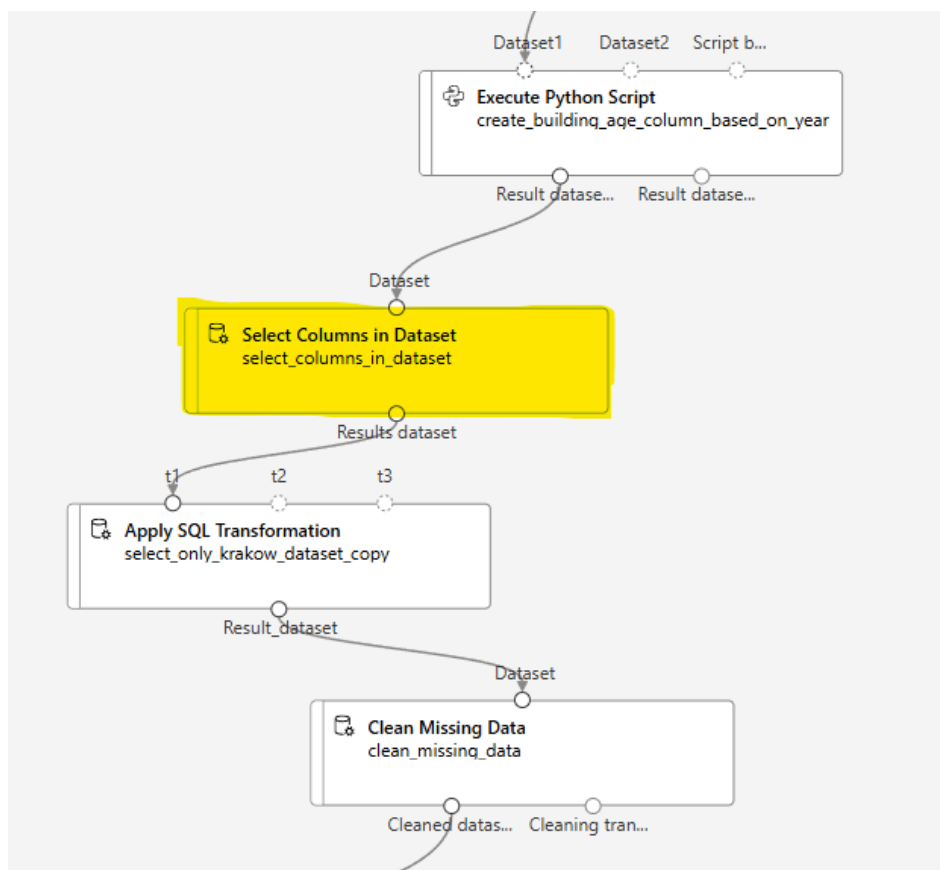
avg_price ponieważ label też jest ceną całkowitą - zachowamy więc spójność.

13.**building_age** – kolumna, o której już pisaliśmy wielokrotnie. Oznacza wiek budynku i **użyjemy** jej zamiast kolumny „year”.








Przedstawmy sobie jeszcze w tabelce jak prezentują się nasze kolumny oraz ich przydatność dla modelu

Kolumna	address	city	floor	id	latitude	longitude	price	rooms	sq	year	district	avg_price	building_age
Użycie	Nie	Nie	Tak	Nie	Nie	Nie	Nie	Tak	Tak	Nie	Tak	Tak	Tak

Kiedy już wiemy, które kolumny będziemy chcieli wykorzystać w modelu (a przynajmniej zbadać ich wskaźnik predykcji), to musimy powiedzieć Azure’owi, że chcemy korzystać wyłącznie z tych kolumn. Możemy to zrobić za pomocą bloku „Select Columns in Dataset”. Tak w tej chwili wygląda nasz pipeline:



W bloku „Select Columns in Dataset” wypisujemy dokładnie te kolumny, które przed chwilą zostały wypisane w tabeli. Bloki pokazane poniżej „Select Columns in Dataset” zostaną omówione za chwilę, natomiast teraz spójrzmy na wynik jaki otrzymujemy po wykonaniu pipeline’u

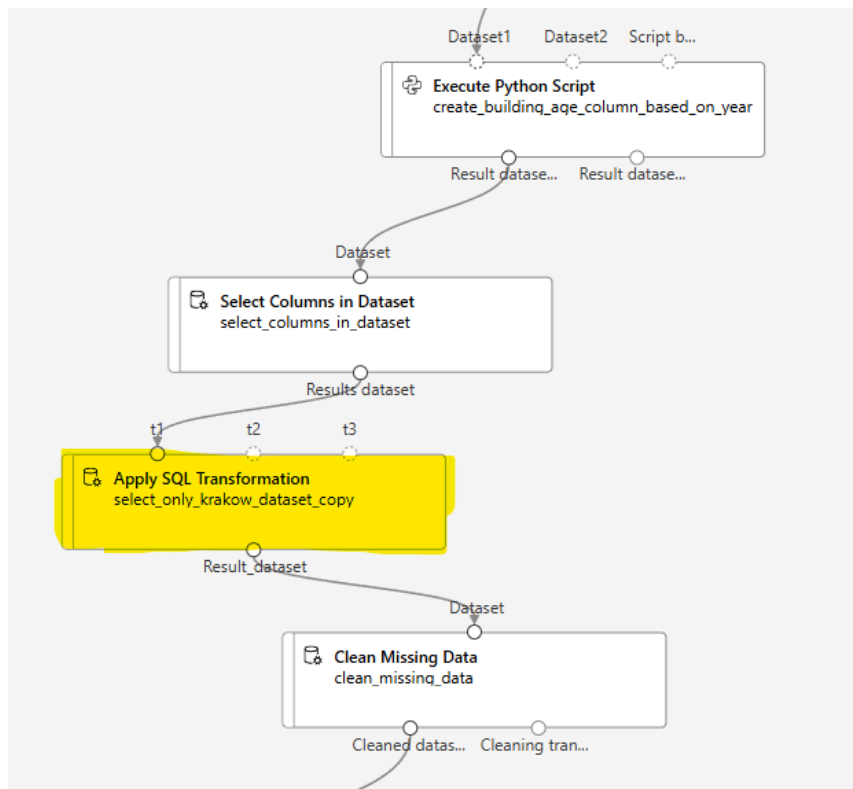
Results_dataset ×						
Rows ?		Columns ?				
8979		7				
floor	price	rooms	sq	district	avg_price	building_age
						
2	749000	3	74.05	Podgórze	833435.71	4
2	427000	2	37	Krowodrza	438689.39	55
2	414600	1	34.55	Czyżyny	373307.91	3
0	750000	4	81.4	Podgórze	880698.35	4
1	756707.4	4	82.43	Prądnik Czerwony	890644.61	4
6	405900	2	41	Prądnik Biały	400584.76	3
4	699000	2	60	Dębniki	568731.6	15
3	530000	1	31	Podgórze	350135.08	9
2	529000	3	53.84	Podgórze	581733.66	4
1	620000	2	43	Stare Miasto	564896.59	35
0	159000	2	49	Podgórze Duchackie	404826.73	25
0	899000	3	74.44	Zwierzyniec	910107.16	5
0	595000	1	31.24	Stare Miasto	447550.49	6
6	539000	3	57.81	Podgórze Duchackie	516717.92	4
8	971904	3	75.93	Grzegórzki	933400.66	4
0	270648	2	37.59	Wzgórze Krzesławickie	254890.27	3
1	266500	1	31.59	Dębniki	242316.78	5
6	498000	2	49.08	Czyżyny	455522.28	6
10	455000	3	45	Krowodrza	530432.55	49
4	1038636	3	88.02	Bronowice	971751.36	4
5	533455	1	36.79	Grzegórzki	452256.16	4
0	382301	2	52.37	Bieżanów	364197.21	5
2	349000	2	37.06	Prądnik Czerwony	385799.42	45
0	730000	3	73.81	Grzegórzki	850474.99	12
4	420000	2	48.2	Podgórze	410231.42	45

Widzimy jednak, że w tabeli wypisanych zostało 6 kolumn do użycia, a na obrazku jest ich 7. Chodzi oczywiście o nasz label – kolumnę „price” – o ile **nie** **użyjemy jej** do uczenia modelu ponieważ jest to nasza zmienna wynikowa, to

jednak musimy ją **wybrać** na tym etapie, aby w późniejszych krokach określić, że faktycznie jest to nasz label. Kolejnym etapem w oczyszczaniu danych będzie usunięcie wierszy, które mają ustawioną wartość „Unknown” dla kolumny „district” – pamiętamy, że mieliśmy to zrobić na końcowym etapie. W tym celu zastosuję blok „Apply SQL Transformation” wpisując następującą kwerendę SQL:

```
SELECT * FROM t1
WHERE t1.district NOT LIKE "%Unknown%"
```

W naszym pipeline blok ten ma odzwierciedlenie zaraz po zastosowaniu bloku „Select Columns in Dataset”





W rezultacie znów utraciliśmy część wierszy.. Ich liczba obecnie wynosi 8902.

Result_dataset

Rows 8902 Columns 7

floor	price	rooms	sq	district	avg_price	building_age
						

To oczywiście znów wpłynie na wynik predykcji, ponieważ utraciliśmy kolejną część obserwacji. Ostatnim etapem oczyszczanie danych będzie oczywiście... oczyszczanie danych – za pomocą bloku „Clean Missing Data” wyczyścimy brakujące dane w naszym datasetcie. Zatrzymajmy się nieco dłużej przy tym bloku ponieważ zrozumienie jego parametrów jest kluczowe do prawidłowego oczyszczenia zbioru. Szczegóły bloku wyglądają następująco:

Clean Missing Data  

Columns to be cleaned ⓘ * Edit column

All columns

Minimum missing value ratio ⓘ * ...

0.0

Maximum missing value ratio ⓘ * ...

1.0

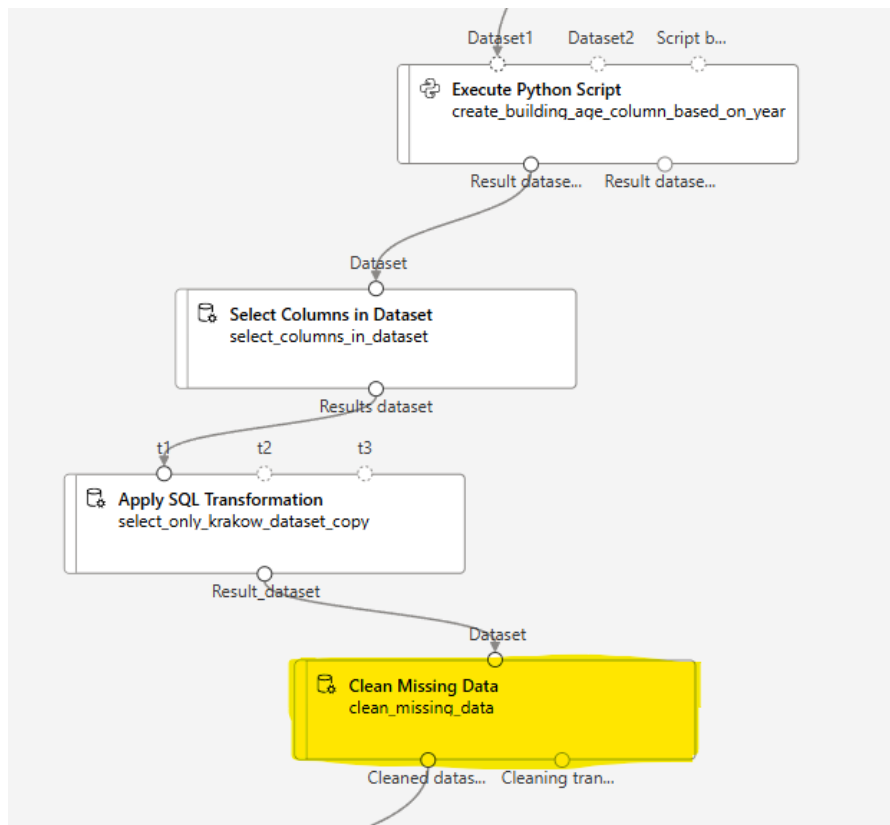
Cleaning mode ⓘ * ...

Remove entire row ▼

Oto co robi każda z tych opcji:

- **Columns to be cleaned** - w tej sekcji wybieramy kolumny, które mają być poddane czyszczeniu. W tym przypadku ustawiono opcję "All columns", co oznacza, że wszystkie kolumny w zbiorze danych będą uwzględnione w procesie czyszczenia.
- **Minimum missing value ratio** - określa minimalny procent brakujących danych, które muszą występować w kolumnie, aby ta kolumna mogła być rozważana do czyszczenia. W tym przypadku ustawione na 0.0, co oznacza, że każda kolumna będzie brana pod uwagę niezależnie od liczby brakujących wartości.
- **Maximum missing value ratio** - określa maksymalny procent brakujących danych, które mogą wystąpić w kolumnie, zanim kolumna zostanie wyczyszczona. Ustawienie 1.0 oznacza, że kolumny z dowolną ilością brakujących danych (do 100%) będą podlegać czyszczeniu.
- **Cleaning mode** – ustawione na "Remove entire row" oznacza, że całe wiersze, które zawierają brakujące dane w którejkolwiek z wybranych kolumn, zostaną usunięte. Jest to strategia czyszczenia danych, która polega na eliminacji niekompletnych rekordów, aby zapewnić integralność danych w dalszych analizach lub modelowaniu.

A oto jak to wygląda w naszym pipeline:



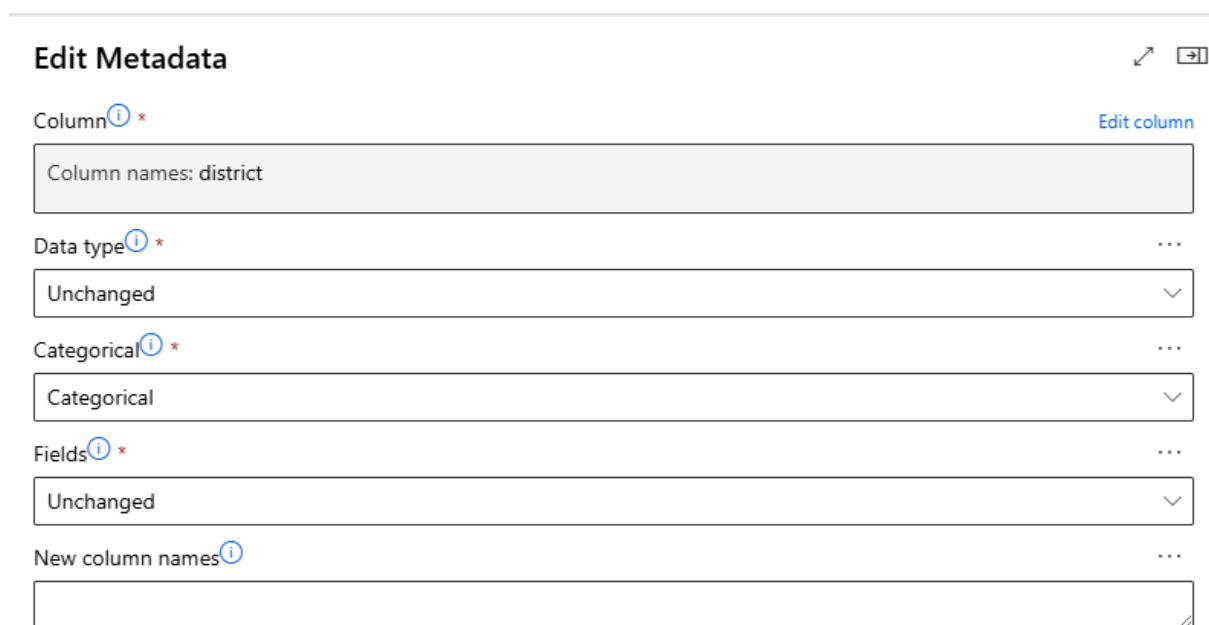
Zobaczmy jeszcze tylko rezultat z wykonania wszystkich pipeline'ów, które zastosowaliśmy w tym rozdziale na poniższym obrazku



Cleaned_dataset						
Rows 8902 Columns 7						
floor	price	rooms	sq	district	avg_price	building_age
2	749000	3	74.05	Podgórze	833435.71	4
2	427000	2	37	Krowodrza	438689.39	55
2	414600	1	34.55	Czyżyny	373307.91	3
0	750000	4	81.4	Podgórze	880698.35	4
1	756707.4	4	82.43	Prądnik Czerwony	890644.61	4
6	405900	2	41	Prądnik Biały	400584.76	3
4	699000	2	60	Dębniki	568731.6	15
2	530000	1	31	Podgórze	350135.08	0


Widzimy, że ostatni cleaning nie wpłynął na liczbę obserwacji – jak wspominałem na początku dane w dataset są wyjątkowo spójne.

7. Edycja metadanych


Jest to krok, który paradoksalnie przysporzył mi najwięcej problemów. Mianowicie do czasu edycji metadanych dla kolumny „district” model wyrzucał wyjątek podczas wykonywaniu treningu (bloku treningowego). Wyjątek jasno mówił, że wartości kolumny „district” nie są kategoryczne. Ostatecznie rozwiązaniem okazała się ręczna konwersja kolumny „district” na kategoryczną. Szczerze mówiąc niezbyt to rozumiem, bo wartości tej kolumny są domyślnie kategoryczne. Wydaje mi się, że jest to typowy issue programu Azure. Jednak natknąłem się na taki problem i chciałbym go również przedstawić. Rozwiązaniem, jak wspomniałem, była edycja kolumny „district” na kategoryczną, a w Azure można to zrobić za pomocą bloku „Edit Metadata”




Edit Metadata  

Column  * [Edit column](#)


Column names: district

Data type  * ...


Unchanged

Categorical  * ...

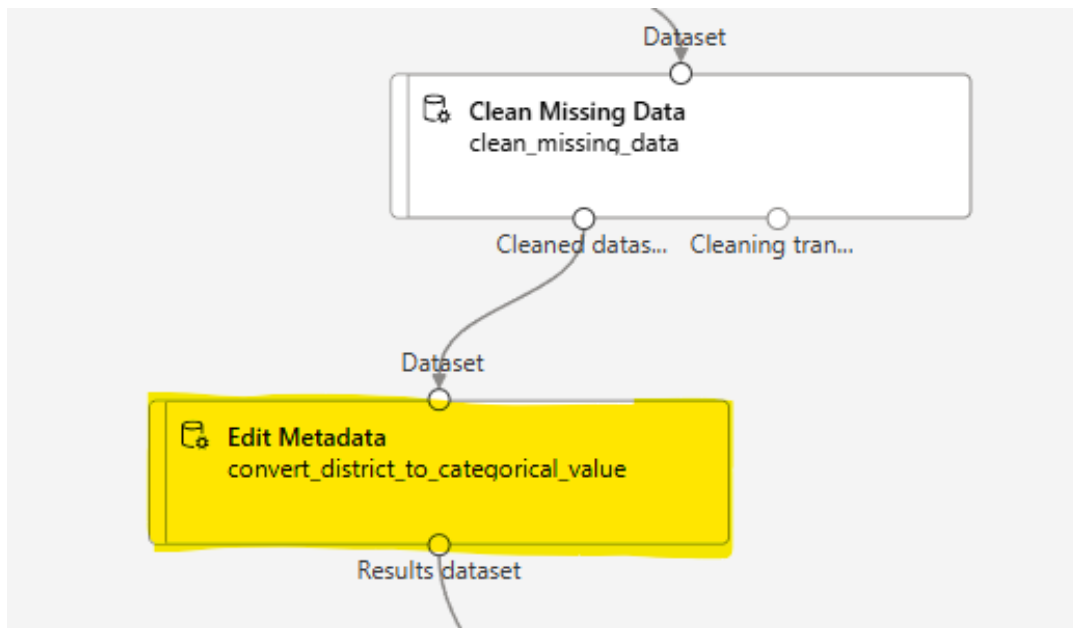
Categorical

Fields  * ...

Unchanged

New column names  * ...

W polu „Column” ustawiona została omawiana kolumna, a w polu „Categorical” wartość „Categorical”, co daje jawnie znać modelowi, że ma traktować tę kolumnę i jej wartości jako kategoryczne. Znajdujemy się w tej chwili na następującym kroku w naszym pipeline (obrazek na kolejnej stronie):



8. Wybór najbardziej istotnych cech

Wspominałem już, że pod koniec pracy z danymi będzie potrzeba wyboru najbardziej istotnych kolumn (cech) w naszym modelu. Póki co nie wiemy jeszcze jaki wpływ będzie miała każda z cech na uczenie modelu. Należałoby określić wskaźnik predykcji – zbyt duża liczba kolumn przy uczeniu może być tak samo szkodliwa jak zbyt mała ich liczba. Na szczęście w Azure istnieje dedykowany blok, który zrobi to za nas – sam określi które kolumny są najbardziej istotne i na tej podstawie wybierze odpowiednią ich (zdefiniowaną przez nas) liczbę. Ten blok w Azure ML nazywa się „Filter Based Feature Selection”. Blok ten jest używany do wybierania najbardziej istotnych cech w zbiorze danych na podstawie określonej metryki. Jego szczegóły zostały pokazane na obrazku na kolejnej stronie. Wyjaśnijmy sobie pokrótce parametry jakie on przyjmuje:

- **Operate on feature columns only** - ustawienie na "True" oznacza, że blok będzie działał tylko na kolumnach cech, a to znaczy że ignoruje kolumny, które nie są cechami (np. etykiety lub inne kolumny, które nie mają być brane pod uwagę przy wyborze cech).

Filter Based Feature Selection



Operate on feature columns only

...

True



Number of desired features *

...

6

Feature scoring method *

...

PearsonCorrelation



Target column *

[Edit column](#)

Column names: price

Output settings



Input settings



Run settings



Node information

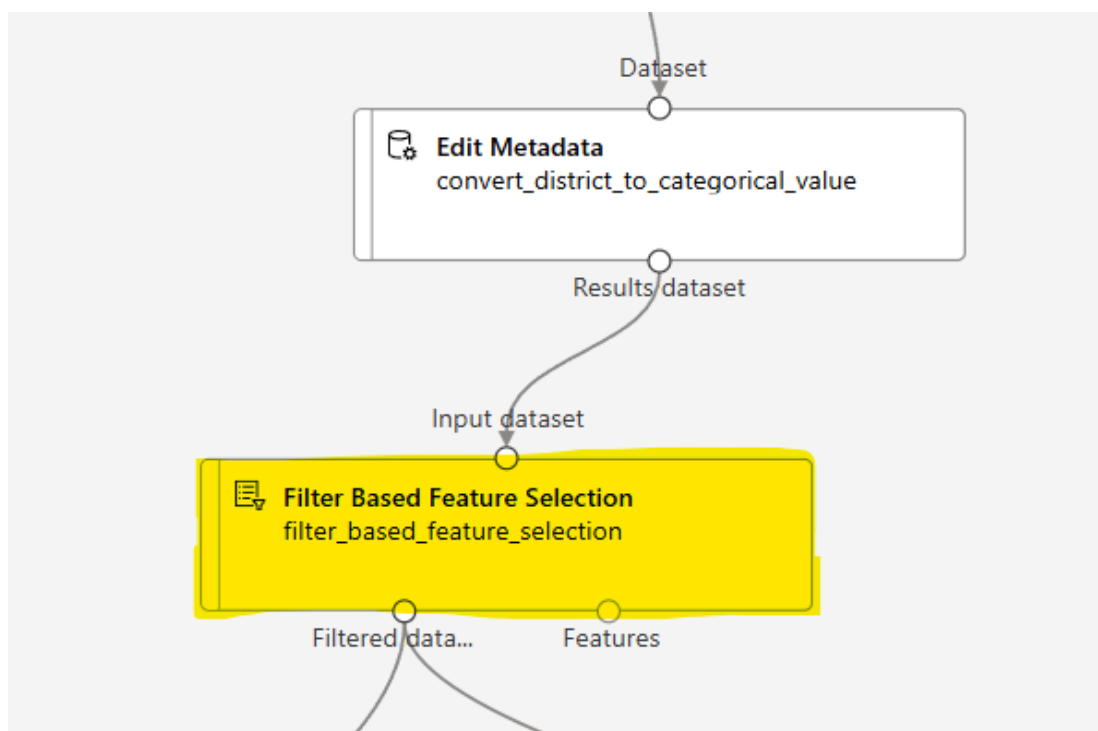


Component information



- **Number of desired features** - liczba "6" oznacza u nas, że blok wybierze sześć najbardziej istotnych cech ze zbioru danych. Po analizie wybranych cech zostaną one przekazane dalej w procesie.
- **Feature scoring method** - ustawienie na "PearsonCorrelation" oznacza, że cechy zostaną ocenione na podstawie współczynnika korelacji Pearsona. Metoda ta mierzy liniową zależność między każdą cechą a kolumną docelową. Cechy o najwyższej wartości korelacji (dodatniej lub ujemnej) są uznawane za najbardziej istotne.
- **Target column** - wskazana kolumna docelowa (label) to "price". Oznacza to, że cechy będą oceniane pod kątem ich korelacji z kolumną "price", która zawiera wartości, które model będzie przewidywał (cena mieszkań).

Osobom początkującym, takim jak ja, może sprawić trudność w doborze odpowiedniej ilości kolumn. W moim przypadku przyznaję, że robiłem to metodą prób i błędów tzn. dobierałem mniejsze i większe ich ilości i sprawdzałem wynik predykcji. Najbardziej optymalne wydaje mi się ustawienie sześciu kolumn, bo powyżej wskaźniki są już tak małe, że praktycznie nie mają żadnego wpływu na wynik, a potrafią go nawet (mniej lub bardziej znacząco) zepsuć. Nasz pipeline wygląda w tej chwili następująco:



Po wykonaniu tego pipeline'u na podstawie parametrów, które przedstawiłem stronę wyżej powinniśmy być w stanie odczytać tzw „Features” w tym bloku, czyli dokładnie wypisane kolejno kolumny (od najbardziej do najmniej istotnej) wraz z podanymi wskaźnikami. Zobaczmy jak to wygląda.

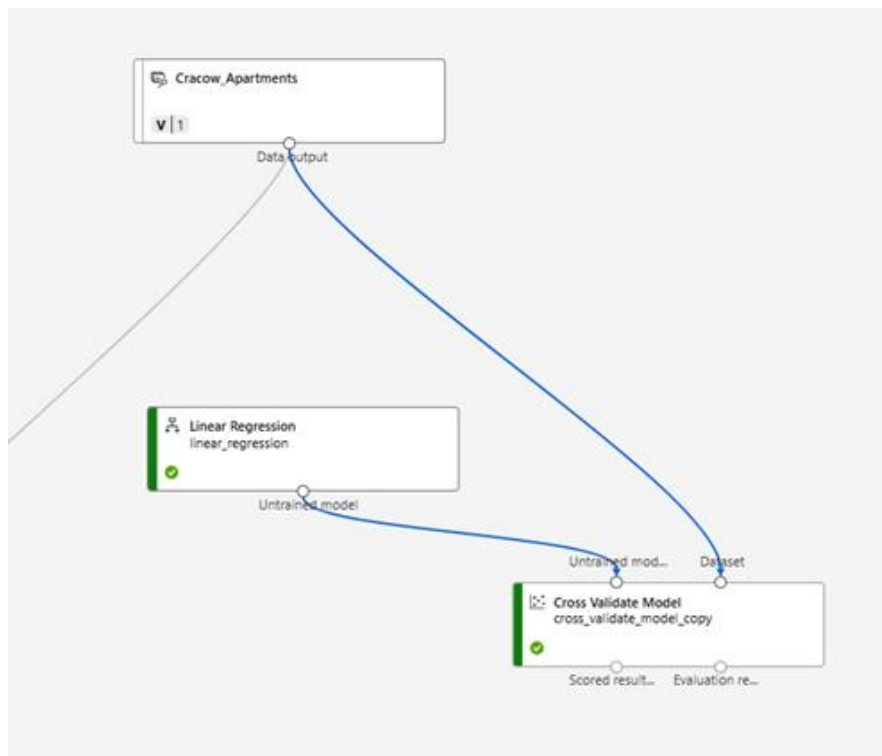
Features		Evaluation_results				
Rows ?	Columns ?					
1	7					
price	avg_price	sq	rooms	district	building_age	floor
1	0.867425	0.742203	0.523116	0.440641	0.12767	0.064258

Pierwsze spostrzeżenie jest takie, że kolumn jest 7, a miało być 6. Kolumna siódma to oczywiście nasz label, który jest najbardziej istotny dla modelu i ma najwyższy wskaźnik – trudno żeby było inaczej, bo dokładnie te wartości przewidujemy. „Price” jako label jednak oczywiście nie bierze udziału w predykcji. Można więc powiedzieć, że najważniejszą dla modelu cechą jest avg_price, który pochodzi z naszego „Web API”! To był strzał w dziesiątkę. Potem kolejno najbardziej istotne są cechy „sq”, „rooms”, „district” (również zasługa Feature Engineeringu i również strzał w dziesiątkę). Najmniej istotne są kolumny „building_age” oraz „floor”. Wszystko wydaje się bardzo logiczne tzn. my sami jako ludzie właśnie w ten sposób myślimy o cechach, które wpływają na cenę: średnia cena w okolicy, metraż, ilość pokoi, potem dzielnica, rok budowy budynku (te nowsze są z reguły droższe) i ewentualnie piętro. Na koniec wypiszmy sobie jeszcze w tabeli nasze cechy i ich wartości.

Cecha	avg_price	sq	rooms	district	building_age	floor
Wartość	0.867425	0.742203	0.523116	0.440641	0.12767	0.064258

9. Trenowanie modelu

Po intensywnym przygotowaniu danych i ich oczyszczeniu przechodzimy do kluczowego kroku jakim jest trenowanie modelu. Trenowanie tego modelu w Azure ML nie będzie się wiele różniło od tego przedstawionego w rozdziale „Najprostszy model – największy błąd predykcji” tzn. sam pipeline i układ bloku pozostaje praktycznie niezmienny. Będziemy mieli blok „Cross Validate Model” przeznaczony do cross-walidacji – i tu żadnych zmian nie będzie, więc nie będę go ponownie opisał. Natomiast **zmieni się algorytm regresji liniowej**. I tu, przy algorytmie chciałbym się przed chwilę zatrzymać. Przypomnijmy sobie jak wyglądał początkowy pipeline najprostszego modelu przedstawionego na obrazku poniżej (to ten sam co w rozdziale „Najprostszy model – największy błąd predykcji”)



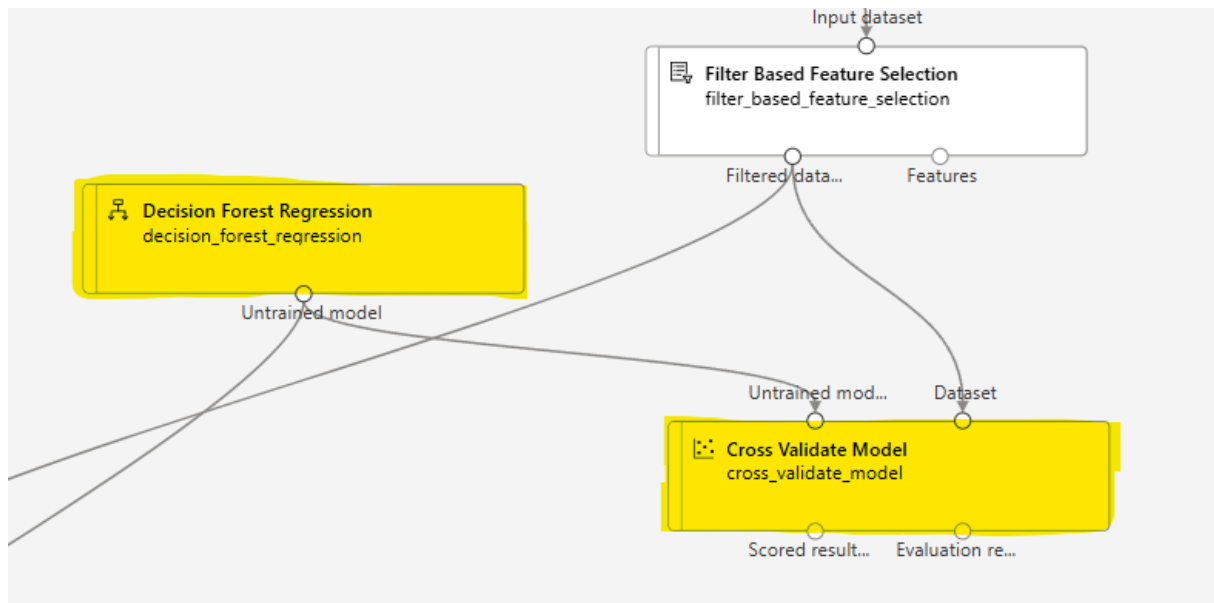
Widzimy, że w początkowym projekcie podłączyliśmy prosty, najbardziej podstawowy algorytm regresji liniowej – w rozdziale „Najprostszy model – największy błąd predykcji” wyjaśniałem dlaczego używam go na samym początku. We wstępie natomiast opisywałem różne rodzaje algorytmów regresji. Ich wybór nie jest prosty i tak było również w moim przypadku – dla tak samo przygotowanych danych testowałem wiele algorytmów, ale ostatecznie wybór padł na algorytm o nazwie „Decision Forest Regression” i chciałbym pokrótce wyjaśnić dlaczego.

Decision Forest Regression jest świetnym wyborem jeśli chodzi o przewidywanie cen nieruchomości ze względu na swoją zdolność do radzenia sobie z wielowymiarowymi i nieliniowymi danymi. Ceny lokali zależą od wielu czynników, takich jak lokalizacja, powierzchnia, wiek budynku, a także odległość od różnych punktów usługowych, co wprowadza złożoność i nieliniowość w danych. Decision Forest Regression radzi sobie z tym problemem, dzięki zastosowaniu wielu drzew decyzyjnych trenowanych na losowych podzbiorach danych. Każde drzewo uczy się na nieco innych danych, co umożliwia modelowi uchwycenie złożonych zależności między zmiennymi. Ponadto agregowanie wyników wielu drzew poprzez uśrednianie wyników pomaga w redukcji wariancji i minimalizuje ryzyko przetrenowania. W praktyce model ten jest w stanie generalizować dobrze nawet na dużych zbiorach danych,

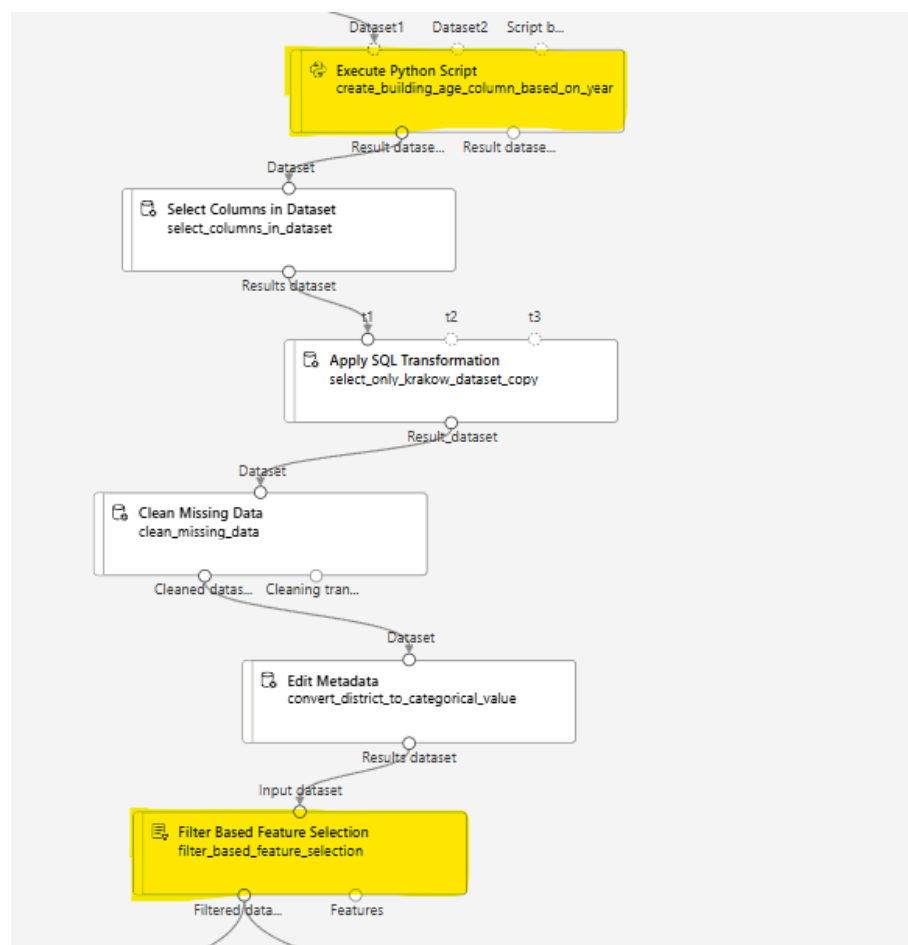
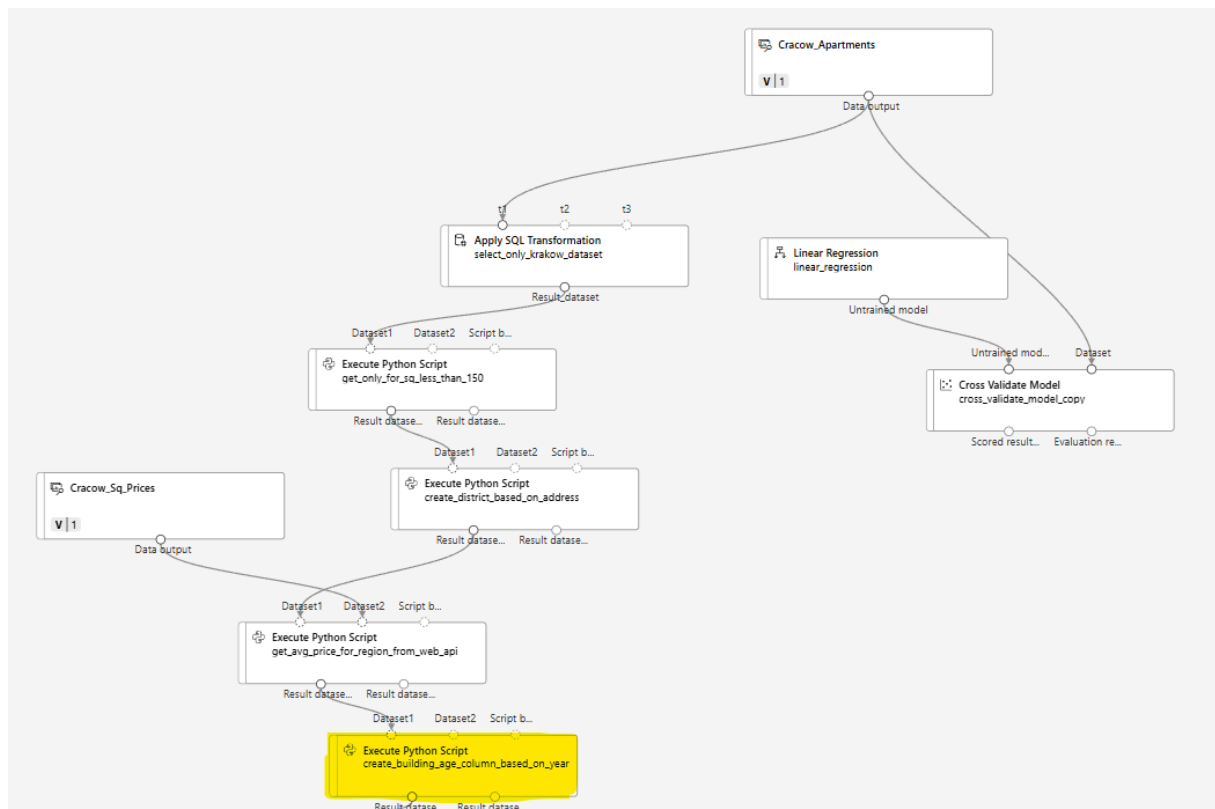
(co u nas raczej nie następuje ale jest typowe przy analizie rynku nieruchomości).

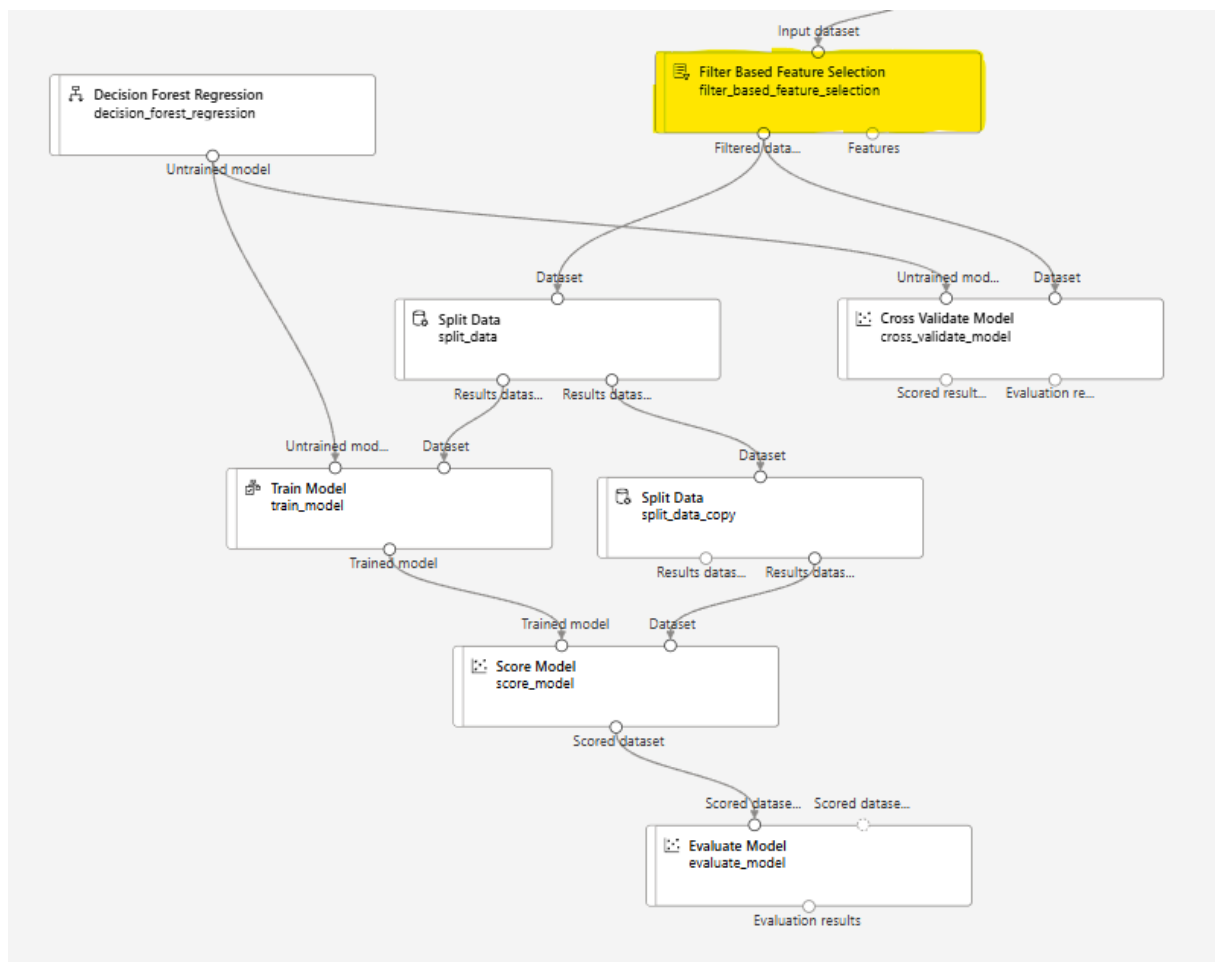
Kolejną zaletą Decision Forest Regression jest odporność na brakujące dane. W sytuacji, gdy nie wszystkie informacje o nieruchomościach są dostępne, model nadal może skutecznie przewidywać ceny, dzięki swojej zdolności do pracy z niepełnymi danymi. Jego zdolność do przetwarzania dużych zbiorów danych również jest istotna, szczególnie gdy analizujemy wiele nieruchomości w różnych lokalizacjach. Trening modelu jest równoległy, co pozwala na efektywne wykorzystanie zasobów obliczeniowych i skrócenie czasu przetwarzania. Algorytm ten oferuje też pewien poziom interpretowalności umożliwiając analizę ważności cech, co jest istotne dla zrozumienia, które zmienne mają największy wpływ na cenę. W porównaniu do innych algorytmów, takich jak Linear Regression, który jest ograniczony do liniowych zależności, czy Support Vector Regression, który jest trudniejszy do skalowania i wymaga starannego tuningu hiperparametrów, Decision Forest Regression oferuje lepszą równowagę między dokładnością a złożonością modelu. Gradient Boosting Machines, choć również skuteczne, są bardziej podatne na przetrenowanie i wymagają więcej czasu na trening, co czyni Decision Forest bardziej praktycznym wyborem. K-Nearest Neighbors natomiast, mimo swojej prostoty, ma problemy ze skalowaniem do dużych zbiorów danych. W rezultacie Decision Forest Regression jest nie tylko skuteczny, ale również skalowalny i odporny na problemy typowe dla danych nieruchomościowych, co czyni go najlepszym wyborem w tego typu zastosowaniach.

W Azure ML oczywiście istnieje dedykowany blok o nazwie „Decision Forest Regression”, którego użyjemy w naszym pipeline to nauki modelu. Na obrazku na kolejnej stronie przedstawiono aktualny pipeline. W tym momencie można już uruchomić model i zobaczyć wyniki predykcji, jednak to zrobimy w kolejnym rozdziale, porównamy sobie do modelu początkowego (z rozdziału „Najprostszy model – największy błąd predykcji”) i wyciągniemy wnioski.



Na koniec tego rozdziału przedstawmy jeszcze całościowe ujęcie pipeline'u jaki został zaprojektowany w Designerze na potrzeby tego projektu. Ponieważ pipeline jest spory, to obrazki będą podzielone, a końcowy pipeline na jednym obrazku będzie początkowym na drugim (i będzie zaznaczony żółtym kolorem). UWAGA: na własne potrzeby walidacyjne mój pipeline (poza walidacją crossową) zawiera również ręcznie wykonany podział danych na dane testowe i walidacyjne oraz scoring i evaluation za pomocą bloków „Score Model” i „Evaluate Model”. Nie omawiam ich jednak tutaj, ponieważ korzystam z walidacji crossowej (ta walidacja o wiele lepiej współdziała z algorytmem Decision Forest Regression, jest prostsza, bardziej intuicyjna i daje lepsze rezultaty).





Przedstawienie wyników i konkluzje

W poprzednim rozdziale przedstawiliśmy i wykonaliśmy ostatecznie zbudowany pipeline. Przyszła pora na przedstawienie wyników lecz najpierw przypomnijmy sobie wyniki predykcji najprościej zbudowanego modelu w rozdziale „Najprostszy model – największy błąd predykcji”.

Numer folda	Liczba rekordów w foldzie	MAE	RMSE	RSE	RAE	R ²
Mean	23764	~233620	~1895173	81,05	0,85	-80,05

Widzimy, że najważniejszy dla nas wskaźnik czyli MAE wyniósł wtedy ok. **233620** (ponad 233 tysiące). Oznacza to, że średni błąd bezwzględny czyli średnia różnica między przewidywanymi wartościami modelu a rzeczywistymi wartościami wyniosła ponad 233 tysiące. Wykonajmy teraz nasz ostateczny pipeline i pokażmy wyniki jakie osiągnęliśmy po odpowiednim przygotowaniu danych i zastosowaniu Feature Engineeringu.

Evaluation_results_by_f						
Rows 12 Columns 7						
Fold Number	Number of examples in fold	Mean_Absolute_Error	Root_Mean_Squared_Error	Relative_Squared_Error	Relative_Absolute_Error	Coefficient_of_Determination
0	890	38974.018902	75411.613394	0.092277	0.228921	0.907723
1	891	52285.991267	140043.841215	0.22948	0.299275	0.77052
2	890	39754.689606	80530.256439	0.103661	0.230617	0.896339
3	890	46844.683438	92939.490079	0.11393	0.267421	0.88607
4	890	39547.45252	74339.722853	0.083796	0.231545	0.916204
5	890	44854.454902	98856.339494	0.14651	0.262419	0.85349
6	890	47385.441947	90854.373625	0.106508	0.255268	0.893492
7	890	45539.525569	93697.411645	0.135474	0.257208	0.864526
8	891	42911.760323	90995.479322	0.105866	0.22678	0.894134
9	890	42482.230435	100144.305637	0.134401	0.23524	0.865599
Mean	8902	44058.024891	93781.28337	0.12519	0.249469	0.87481
Standard Deviation	8902	4188.262316	18602.629521	0.041675	0.023286	0.041675

Na tym etapie widać już sporą różnicę, ale wypiszmy sobie jeszcze wyniki uśrednione w tabeli

Numer folda	Liczba rekordów w foldzie	MAE	RMSE	RSE	RAE	R ²
Mean	8902	~44058	~93781	0,13	0,25	0,87

Znaczenie odpowiednich wskaźników wyjaśniałem już w rozdziale „Najprostszy model – największy błąd predykcji”, więc nie ma sensu opisywać ich raz jeszcze. Widać natomiast jak bardzo wyniki poprawiły się. Zanim opiszę moje spostrzeżenia i konkluzje wypiszmy sobie jeszcze tabelę porównawczą dla najprostszego i zaawansowanego modelu (FE, odpowiednie przygotowanie danych).

Typ modelu	Numer folda	Liczba rekordów w foldzie	MAE	RMSE	RSE	RAE	R ²
Prosty	Mean	23764	~233620	~1895173	81,05	0,85	-80,05
Zaawansowany	Mean	8902	~44058	~93781	0,13	0,25	0,87










Kolorami w tabeli zaznaczyłem korzystnie oraz niekorzystnie (odpowiednio na zielono i czerwono) wypadające współczynniki porównując do siebie modele: prosty i zaawansowany. Widzimy, że jedyną wadą modelu zaawansowanego to sporo zmniejszona liczba foldów – liczba ta jest mała – 9 tys. To naprawdę niewiele przy uczeniu zwłaszcza przy zastosowaniu zaawansowanych algorytmów takich jak Decision Forest Regression. Mimo wszystko najważniejszy współczynnik MAE wynosi teraz ok. 44058 – czyli można powiedzieć, że błąd wyceny danego lokalu jest na poziomie ok. 44 tysięcy na plus lub minus. Czy to sporo? Cóż, zależy jak na to spojrzymy. W moim przypadku, gdzie liczba danych nie była spora, jestem osobą bardzo początkującą, a model ma mi w przyszłości służyć jako tool do aplikacji typu „OtoDom” (gdzie system sam będzie **podpowiadał** proponowaną cenę) wynik naprawdę mnie zadowolił chociaż, nie ukrywam, spodziewałem się lepszej predykcji (próbowałem „wycisnąć” z modelu MAE na poziomie 10 tys.). Ale spójrzmy na to też z nieco innej perspektywy – tzn. rynkowej. Liczba 44 tys. dla

MAE jest wartością bezwzględną i niewiele nam mówi z punktu widzenia rynku nieruchomości. Spróbujmy policzyć sobie stosunek średniej ceny wszystkich mieszkań podzielonej na metraż (cena za metr) do średniej MAE i zobaczmy o ile „metrów kwadratowych lokalu” myli się model. Po zastosowaniu poniższego zapytania w programie Access

```
1 SELECT Round(Avg(price / sq), 2)
2 FROM Houses
3 WHERE city LIKE "Kraków"
4
```

Otrzymamy wartość **10426,37** – jest to nasza średnia cena za metr dla wszystkich lokali w datasetcie. Teraz wyliczając stosunek $\sim 44058 / \sim 10426$ otrzymujemy wartość $\sim 4,2$. Czyli model „myli się” średnio o 4,2 m². I znów na pytanie „czy to sporo” można odpowiedzieć dwojako, tzn: zależy od przypadku użycia. W moim systemie „OtoDom”, który nie będzie aplikacją komercyjną, a raczej edukacyjną, wynik jest naprawdę dobry. Z drugiej strony w realnych zastosowaniach na pewno umieściliśmy informacje dla użytkownika, żeby nie trzymał się sztywno proponowanej ceny, bo model może się mylić o daną wartość – to jest podstawa wszelkich podpowiedzi aplikacji niezależnie czy działają w sposób stricte obliczeniowy, czy na ML. Należy również pamiętać, że cały czas trzymamy się tutaj wartości średnich – oznacza to, że na pewno znajdują się w modelu wartości predykcji bardzo dobre i bardzo złe (np. dla danego wiersza MAE wynosiło procentowo 6%, a dla innego 40%) – dopiero potem wynik jest uśredniany. I tego typu przykłady znajdziemy już na pierwszy rzut oka patrząc na nasze scored result (obrazek na kolejnej stronie). Została tutaj dodana kolumna „Scored Labels”, którą porównać możemy do naszego labela czyli kolumny „price”. Na zielono i czerwono zaznaczyłem po kilka wartości kolejno bardzo ładnych i tych gorszych, a nawet nieakceptowalnych.

Najśmieszniejsze jest to, że tak naprawdę dopiero teraz powinien zacząć się proces ciągłego ulepszania modelu. Nie mogę powiedzieć, że model jest skończony. Należałoby użyć narzędzi do analizy danych i zobaczyć – czy wysokie błędy predykcji są losowe, czy występują pewne korelacje między dobrze i źle przewidzianymi rezultatami? Jeżeli korelacje nie są losowe, to dlaczego tak się dzieje? Jak można poprawić model jeżeli korelacje nie są losowe? A co jeżeli faktycznie są losowe? To tylko parę pytań, które należy zadać na tym etapie.

Scored_results		Evaluation_results_by_f						
Rows ①		Columns ②						
8902		9						
price	avg_price	sq	rooms	district	building_age	floor	Scored Labels	Fold Number
								
749000	833435.71	74.05	3	Podgórze	4	2	748000	1
427000	438689.39	37	2	Krowdrza	55	2	395260	7
414600	373307.91	34.55	1	Czyżyny	3	2	409762.1	8
750000	880698.35	81.4	4	Podgórze	4	0	744171.864	9
756707.4	890644.61	82.43	4	Prądnik Czerwony	4	1	764010.0944	9
405900	400584.76	41	2	Prądnik Biały	3	6	392465.5424	9
699000	568731.6	60	2	Dębники	15	4	636892.9	6
530000	350135.08	31	1	Podgórze	9	3	467423	6
529000	581733.66	53.84	3	Podgórze	4	2	552035.634632	0
620000	564896.59	43	2	Stare Miasto	35	1	520482.76	9
159000	404826.73	49	2	Podgórze Duchackie	25	0	389354.96	8
899000	910107.16	74.44	3	Zwierzyniec	5	0	871171.6	7
595000	447550.49	31.24	1	Stare Miasto	6	0	538531.024	4
539000	516717.92	57.81	3	Podgórze Duchackie	4	6	541874.6	7
971904	933400.66	75.93	3	Grzegórzki	4	8	972405.113	0
270648	254890.27	37.59	2	Wzgórze Krzesławickie	3	0	269674.78	3
266500	242316.78	31.59	1	Dębники	5	1	311187.42	6
498000	455522.28	49.08	2	Czyżyny	6	6	484792.301587	0
455000	530432.55	45	3	Krowdrza	49	10	554452.32	6
1038636	971751.36	88.02	3	Bronowice	4	4	1028147.166	0
533455	452256.16	36.79	1	Grzegórzki	4	5	529064.12	7
382301	364197.21	52.37	2	Bieżanów	5	0	376386.224	8
349000	385799.42	37.06	2	Prądnik Czerwony	45	2	357791.5	6
730000	850474.99	73.81	3	Grzegórzki	12	0	767954	0
439000	410331.42	48.2	3	Podgórze	45	4	435276.64	5

Jedno natomiast jest pewne – model zawsze wymaga ulepszenia, a zakończenie pracy nad modelem następuje wtedy, gdy uznamy (my lub osoby trzecie), że wyniki są zadowalające. Nie oznacza to jednak koniec ciągłego rozwoju modelu. Doskonale wiemy jak ważny jest proces CI/CD (Continuous Integration / Continuous Delivery) w dostarczaniu i rozwoju oprogramowania i tutaj jest tak samo. Nie da się po prostu wykonać modelu i zostawić go żeby działał sobie tak długo jak chcemy. W przypadku mojego modelu jest on dla mnie dość zadowalający, ale nie oznacza końca pracy, a dopiero początek – zwłaszcza jeśli chcę go później wykorzystać w realnych zastosowaniach.

Bibliografia

1. Szeliga M., Wykłady realizowane w trakcie przedmiotu „Zastosowanie Metod Uczenia Maszynowego” w formie multimedialnej i PDF, WSZiB Kraków, 2024
2. Król-Nowak A., Kotarba K.: „Podstawy uczenia maszynowego”, AGH Kraków, 2022
3. Szeliga M., „Praktyczne uczenie maszynowe”, PWN SA Warszawa, 2019
4. <https://learn.microsoft.com/en-us/training/modules/introduction-to-machine-learning/>
5. <https://learn.microsoft.com/en-us/training/modules/fundamentals-machine-learning/>
6. <https://developers.google.com/machine-learning/crash-course/linear-regression?hl=pl>
7. <https://learn.microsoft.com/en-us/azure/machine-learning/component-reference/decision-forest-regression?view=azureml-api-2>
8. https://en.wikipedia.org/wiki/Random_forest
9. <https://www.kaggle.com/datasets/dawidcegielski/house-prices-in-poland/data>