Projekt – sieci neuronowe	Data złożenia projektu: 20.04.2023
Numer grupy projektowej: -	Imię i nazwisko I: Piotr Sobol Imię i nazwisko II: Szymon Rogowski

Przewidywanie cen samochodów marki Volkswagen

1. Opis problemu i danych

Ideą projektu było zastosowanie technik sztucznej inteligencji do rozwiązania praktycznego problemu **regresji**, jakim jest predykcja cen samochodów na podstawie ich podstawowych parametrów. Serwisem oferującym największą liczbę danych w tym zakresie jest popularny polski serwis OTOMOTO(https://www.otomoto.pl/).

Projekt dostępny jest na repozytorium github: projekt.

Parametry predykcji:

- Rocznik pojazdu – dane ilościowe
- Aktualny przebieg – dane ilościowe
- Pojemność silnika – dane ilościowe
- Model pojazdu – dane jakościowe
- Typ paliwa – dane jakościowe
- Informacja pojawiająca się przy artykule czy cena danego modelu jest powyżej, poniżej, czy równa średniej cenie pojazdów o zbliżonych parametrach. – dane jakościowe
- *Cena* – dane ilościowe – **zmienna wyjściowa dla której będą przewidywane wartości**

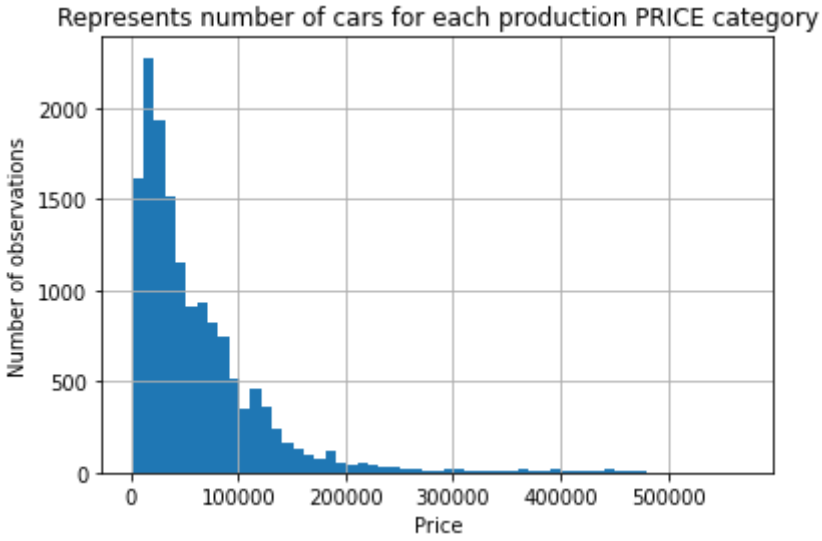
Ponieważ predykcja cen wszystkich pojazdów w serwisie nie byłaby miarodajna, ograniczono się jedynie do marki Volkswagen, ze względu na nasze doświadczenie i różnorodność danych.

W celu pobrania danych z serwisu zaimplementowano *WebScrapper* pobierający opisane powyżej dane z serwisu OTOMOTO. Aplikacja jest oparta na bibliotece beautifulsoup4 i została dołączona do repozytorium projektu. Stworzono również klasę do automatycznej obróbki pobieranych danych i konwersji na zmienne typu *int*. Zbiór danych jest podzielony na 6 kolumn (poza ceną), z których połowa jest danymi jakościowymi, a połowa ilościowymi.

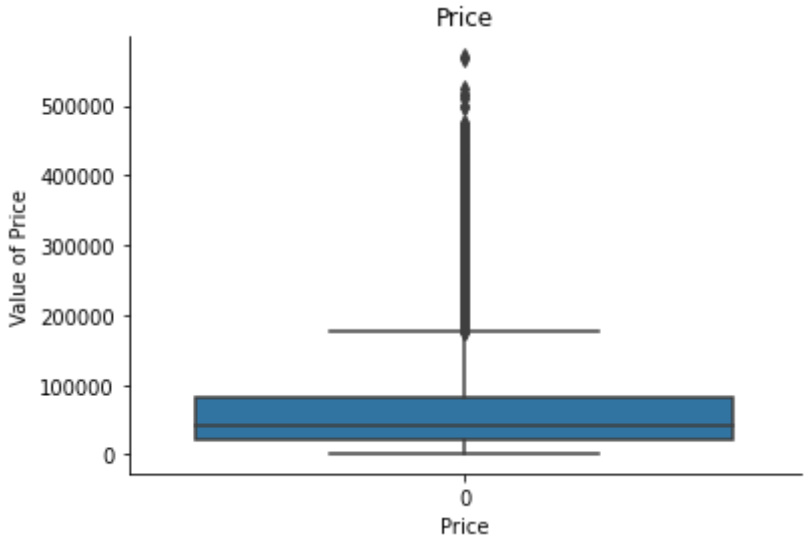
Poniżej pokazana są podstawowe statystyki dla datatsetu:

	Price	Year	Mileage	Tank capacity	Fuel type	Model	Estimation
count	14915.000000	14915.000000	14915.000000	14915.000000	14915.000000	14915.000000	14915.000000
mean	61420.578478	2012.424137	172886.681931	1755.932618	4.436876	27.893530	1.410325
std	61593.454821	6.757177	100170.759677	395.745487	2.952525	13.300906	1.071621
min	1500.000000	1958.000000	1.000000	77.000000	1.000000	1.000000	0.000000
25%	20000.000000	2008.000000	104343.000000	1498.000000	1.000000	16.000000	0.000000
50%	41999.000000	2013.000000	177000.000000	1896.000000	7.000000	32.000000	2.000000
75%	82900.000000	2018.000000	234350.000000	1968.000000	7.000000	39.000000	2.000000
max	570000.000000	2023.000000	700000.000000	5998.000000	7.000000	50.000000	3.000000

Wymiary oczyszczonego zbioru: 14105 wierszy i 7 kolumn.



Rys.1 Histogram dla zmiennej wyjściowej.



Rys.2 Box plot dla zmiennej wyjściowej.

Najniższa wartość zmiennej wyjściowej: 1 500 zł

Najwyższa wartość zmiennej wyjściowej: 570 000 zł

Uśredniona wartość zmiennej wyjściowej: 61 420 zł

Odchylenie standardowe zmiennej wyjściowej: 61 593

2. Obróbka danych

Kolejno zastosowano wybrane elementy kompleksowego i całkowitego procesu pre-processingu danych:

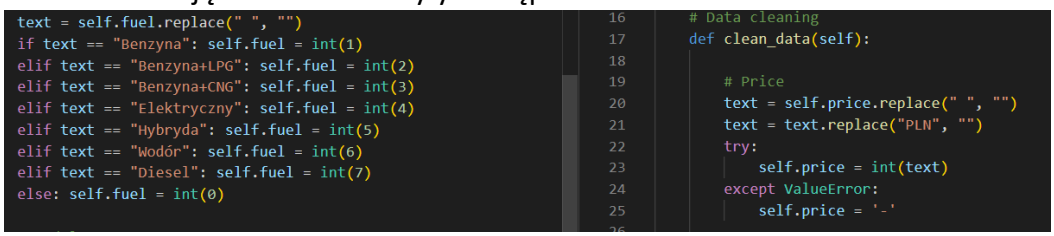
- data cleaning - usuwania z danych wszelkich niedokładności, błędów, duplikatów, braków danych i niepotrzebnych informacji, outlier detection
- data transformation (polega na zmianie formy danych na inny format lub strukturę, aby ułatwić analizę, w tym normalizacja, skalowanie).
- data integration (proces łączenia danych z różnych źródeł i konwertowania ich na spójną strukturę, w naszym wypadku można by doszukiwać się podobnego działania w WebScrapperze który zaciąga dane z różnych tagów html)
- data reduction / dimension reduction - zmniejszeniu liczby wymiarów w danych, zachowując przy tym jak najwięcej informacji, tak by infrastruktura i oprogramowanie były wstanie przetworzyć potężne ilości danych, mogą to być: analiza głównych składowych, selekcja cech, redukcja bazy danych i redukcja objętości danych. W naszym wypadku nie używane.

Skrypt Python do *WebScrappingu* zaciągnął ze strony OTOMOTO 17 899 rekordów. Podczas *preprocessingu* i dalszej obróbki danych odrzucono 2 984 rekordów, a podczas *outliers detection* odpadło dalsze 810. Łącznie odrzucono 21% pobranego zbioru. A więc zmiany od 17 899 do 14 915, aż do finalnej liczby wierszy 14 105.

Procesy obróbki danych zastosowane w projekcie:

- Standaryzacja / normalizacja danych – w naszym wypadku jest to standaryzacja danych przy pomocy klasy StandardScaler z pakietu scikit-learn. Standaryzacja to proces polegający na przekształceniu wartości numeryczne w taki sposób, aby miały średnią wartość 0 i odchylenie standardowe równoważne 1. Podobnie jak normalizacja (proces przeskalowania danych numerycznych w taki sposób, aby mieściły się w przedziale [0, 1]). Celem tychże procesów jest przeskalowanie danych liczbowych taki, aby różne zmienne w danych były porównywalne i mieściły się w podobnej skali (różne labele mają różne skale zmiennych). Sama w sobie klasa StandardScaler oblicza średnią i odchylenie standardowe danych treningowych/uczących, a następnie skaluje dane treningowe i testowe wyliczonym współczynnikiem.

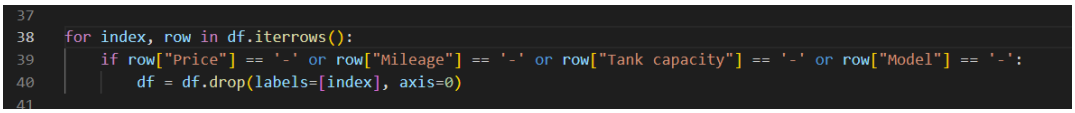
- Data transformation – ten proces był realizowany w skrypcie Python. W tym celu stworzono specjalną klasę reprezentującą pojedynczy rekord i zainicjowano ją wartościami tekstowymi z *WebScrappera*. Zaimplementowano specjalne metody do konwersji wartości atrybutów klasy na wartości liczbowe. Jeżeli wartości danych były ewidentnie nieprawidłowe, zastępowano je znakiem ‘-‘. Wszystkie wiesze zawierające takie znaki były następnie odrzucane.



```
text = self.fuel.replace(" ", "")
if text == "Benzyna": self.fuel = int(1)
elif text == "Benzyna+LPG": self.fuel = int(2)
elif text == "Benzyna+CNG": self.fuel = int(3)
elif text == "Elektryczny": self.fuel = int(4)
elif text == "Hybryda": self.fuel = int(5)
elif text == "Wodór": self.fuel = int(6)
elif text == "Diesel": self.fuel = int(7)
else: self.fuel = int(0)
```

```
16    # Data cleaning
17    def clean_data(self):
18
19        # Price
20        text = self.price.replace(" ", "")
21        text = text.replace("PLN", "")
22        try:
23            self.price = int(text)
24        except ValueError:
25            self.price = '-'
```

Rys.3 Przykładowa konwersja i selekcja danych przy procesie transformacji.



```
37
38  for index, row in df.iterrows():
39      if row["Price"] == '-' or row["Mileage"] == '-' or row["Tank capacity"] == '-' or row["Model"] == '-':
40          df = df.drop(labels=[index], axis=0)
41
```

Rys.4 Fragment kodu przedstawiający odrzucanie błędnych danych.

- *Outliers detection* – jest to pojęcie opisujące proces wyszukiwania i usuwania nietypowych obserwacji w zbiorze danych, które różnią się znacząco od reszty danych. Wartości obserwacji których skala znacznie się różni od reszty zbioru tworzą szum na datasecie który wpływa na powiększenie zjawiska overfiting między innymi.
  Implementacja polegała na obliczaniu wartości *z-score* dla każdej kolumny w ramce danych. Wartość *z-score* dla danej wartości to liczba określająca o ile standardowych odchyleń dana wartość różni się od średniej wartości w tej kolumnie. Ustawiono próg *z-score,* powyżej którego wartości są uznawane za odstające.
  Metoda z-score opiera się na standardowej normalizacji danych, w której każda wartość w zbiorze danych jest przeskalowana w taki sposób, aby miała średnią równą 0 i odchylenie standardowe równie 1. Następnie wyliczane są z-score dla każdej wartości w zbiorze danych, czyli liczba standardowych odchyleń, jakie dana wartość różni się od średniej. Wartości z-score większe lub mniejsze niż ustalony próg (u nas 3) uznaje się za wartości odstające.
  Zaś metoda IQR (interquartile range) opiera się na medianie (to wartość, która dzieli zbiór danych na dwie równe części) i rozstępie międzykwartylowym (to różnica między pierwszym, a trzecim kwartylem (25% i 75% percentylem) – boxplot, wykres wąsiasty. Wartości, które są mniejsze niż pierwszy kwartyl minus 1,5 razy rozstęp lub większe niż trzeci kwartyl plus 1,5 razy rozstęp uznaje się za wartości odstające.

```
z_scores = (df - df.mean()) / df.std()
threshold = 3
```

```
df = df[(np.abs(z_scores) < threshold).all(axis=1)]
```

Rys.5 Fragment kodu odpowiedzialny metodę z-score.

```
# Interquartile Range (IQR)
Q1 = data.quantile(0.25)
Q3 = data.quantile(0.75)
IQR = Q3 - Q1
data = data[~((data < (Q1 - 1.5 * IQR)) | (data > (Q3 + 1.5 * IQR))).any(axis=1)]
```

Rysunek 1: Przykładowa implementacja metody IQR

Łączna liczba rekordów odrzuconych podczas preprocessingu: 3794.

Dane podzielono na zbiór uczący i testujący za pomocą funkcji *train_test_split()* biblioteki *sklearn*. Funkcja ta losowo dzieli zbiór danych na dwa podzbiory: zbiór treningowy i testowy. Zbiór treningowy jest wykorzystywany do trenowania modelu, a zbiór testowy służy do oceny jakości modelu na danych, które nie były używane podczas treningu.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                    random_state=42, shuffle=True)
```

Rys. Implementacja podziału danych na zbiór testowy i treningowy.

X_train i y_train będą służyć do trenowania modelu, a X_test i y_test będą użyte do testowania modelu. Test_size określa proporcję zbioru testowego do zbioru treningowego (w tym przypadku 0.2, co oznacza, że 20% danych zostanie użyte do testowania, a 80% do treningu). Random_state to ziarno losowości, które zapewnia, że podział danych jest taki sam przy każdym uruchomieniu skryptu, a shuffle=True oznacza, że dane uczące będą mieszane przed każdą epoką (iteracją) uczenia modelu, co zwiększa losowość i uniemożliwia wybór wierszy w jakiejś konkretnej kolejności – zapobieganie sytuacji, w której algorytm uczenia maszynowego uczy się kolejności występowania danych uczących, a nie ich właściwych cech – cykliczność danych uczących. Zapobiegać to może również skośności modelu, czyli sytuacji w której poziom odchylenia prognoz generowanych przez model od rzeczywistych wartości. Model z wysokim poziomem skośności ma tendencję do przewidywania wartości, które są daleko od rzeczywistych wartości.

3. Opis zastosowanych sieci neuronowych

Zastosowane typy architektur sieci neuronowych:

Wykorzystaliśmy 3 razy klasyczne sieci neuronowe, najbardziej podstawowe czyli MLP (multilayer perceptron) różniące się parametrami takimi jak: liczba warstw, neuronów, regularyzacja, aktywacja czy metodami optymalizacji. Oraz dwa typy sieci rekurencyjnych RNN (recurrent neural network) tj.: LSTM oraz GRU.

Aspekty techniczne to przede wszystkim:

- Numpy, Pandas – analiza numeryczna, obróbka macierzy, przygotowywanie danych, praca z matrixami
- Matplotlib, Seaborn – wykresy i wizualizacja danych
- Statsmodel – analiza statystyczna, wiele wbudowanych funkcjonalności

Oraz same sieci neuronowe:

- tensorflow – bardzo zaawansowana biblioteka do tworzenia szerokiej maści modeli uczenia maszynowego oraz sieci neuronowych oparta o jednostkę tensora, czyli podstawowa jednostka przepływu danych przez model w postaci matrixów n-tych wymiarów. Posiada rzecz jasna wiele wbudowanych, przydatnych funkcji które są implementacją matematyki opisującej modele. Przydatna w całej gamie zastosowań uczenia maszynowego, w tym głębokiego.
- Keras – bazuje na Tensorflow, właściwie jest nakładką do Tensorflow. Znacznie łatwiejszy niż wspominany Tensorflow i bardziej „user friendly" co czyni go jednocześnie znacznie mniej zaawansowanym i ograniczającym pełnoskalowe możliwości pisania skomplikowanych modeli. Zaleca do pisania modeli deep learningowych.
- Scikit-Learn – biblioteka podobna do Keras, myśle że jednak mniej zaawansowana, dobra dla początkujących. Często używana do analizy danych, wizualizacji danych oraz przetwarzania wstępnego. Raczej polecana do zagadnień tradycyjnego uczenia maszynowego.

Dokładny opis poszczególnych sieci neuronowych:

Najbardziej pierwotny, pojedynczy neuron, a właściwie perceptron (Rosenblata) uczy się na zasadzie zmiany wag, mówiąc że sieć się uczy mamy na myśli dobranie najlepszych wartości wag dla proponowanych danych. Wartość wyjściowa z która jest obliczana na każdym neuronie jest sumą wartości z warstw poprzedniej pomnożonej przez wagi oraz nałożonej na ten wynik funkcje aktywacji, co zwraca nam finalny output neuronu.

Sieci neuronowe składa się z trzech warstw:

- Warstwa wejściowa / input layer (czerwone neurony – kółeczka) – zbiera dane i przesyła je dalej (każdy neuron z warstwy wejściowej przesyła dane do każdego neuronu w warstwie ukrytej)

- Warstwa ukryta / hidden layer (szare neurony) – są to stany pośrednie. Tutaj przede wszystkim zachodzi proces uczenia się i szukana liniowych i nieliniowych zależności. Może być wiele warstw ukrytych. Im sieć ma więcej warstw ukrytych tym może znaleźć głębsze zależności.

- Warstwa wyjściowa / output layer (zielone neurony) – zwracany jest wynik.

Krótki opis dwóch najpopularniejszych schematów uczenia się sieci:

- FeedForward - polega na tym, że na początku losujemy wagi. Następnie zgodnie z wcześniejszym obrazkiem z sieci idziemy od lewej do prawej. Wagi możemy poprawić aby dostać lepszy wynik. wektor wejściowy przekazywany jest przez warstwy neuronów aż do wyjścia sieci, bez żadnego wpływu na proces uczenia. W tym procesie każda warstwa neuronów wykonuje operację liniową, zwykle mnożąc wektor wejściowy przez macierz wag, a następnie przekazując wynik przez funkcję aktywacji.
- BackPropagation / BackProp - informacja o błędzie predykcji jest propagowana wstecz przez sieć, a następnie wykorzystywana do aktualizacji wag. Proces ten rozpoczyna się od obliczenia błędu predykcji na wyjściu sieci, a następnie propagacji tego błędu wstecz przez każdą warstwę sieci. W wyniku propagacji błędu dla każdej warstwy obliczane są gradienty wag, które są wykorzystywane do aktualizacji wag (aktualizowane w kierunku przeciwnym do gradientu funkcji kosztu). W ten sposób określamy wpływ każdej wagi na wynik. Proces ten jest powtarzany n-razy aż do osiągnięcia zakładanego progu błędu, czyli minimalizacji funkcji kosztu.

Krótki opis przygotowanych sieci:

- ❖ Sieć MLP – jeśli każdy węzeł poprzedniej warstwy jest połączony z węzłem każdej następnej warstwy to taką sieć nazywamy właśnie MLP. Tak jak wspominałem na początku paragrafu, w każdej warstwie sieci MLP znajdują się neurony, które obliczają ważoną sumę wejść, a następnie przekształcają wynik za pomocą funkcji aktywacji. W każdej kolejnej warstwie neurony korzystają z wyniku z poprzedniej warstwy jako wejście.
  W procesie uczenia sieci MLP wykorzystywana jest propagacja wsteczna. Funkcje aktywacji, takie jak sigmoidalna, tangens hiperboliczny lub ReLU, są wykorzystywane do wprowadzenia nieliniowości do sieci MLP, ale i nie tylko, co umożliwia jej przetwarzanie bardziej skomplikowanych wzorców.
  Proces uczenia składa się z następujących kroków:
  1. Inicjalizacja wag, zwykle losowo.

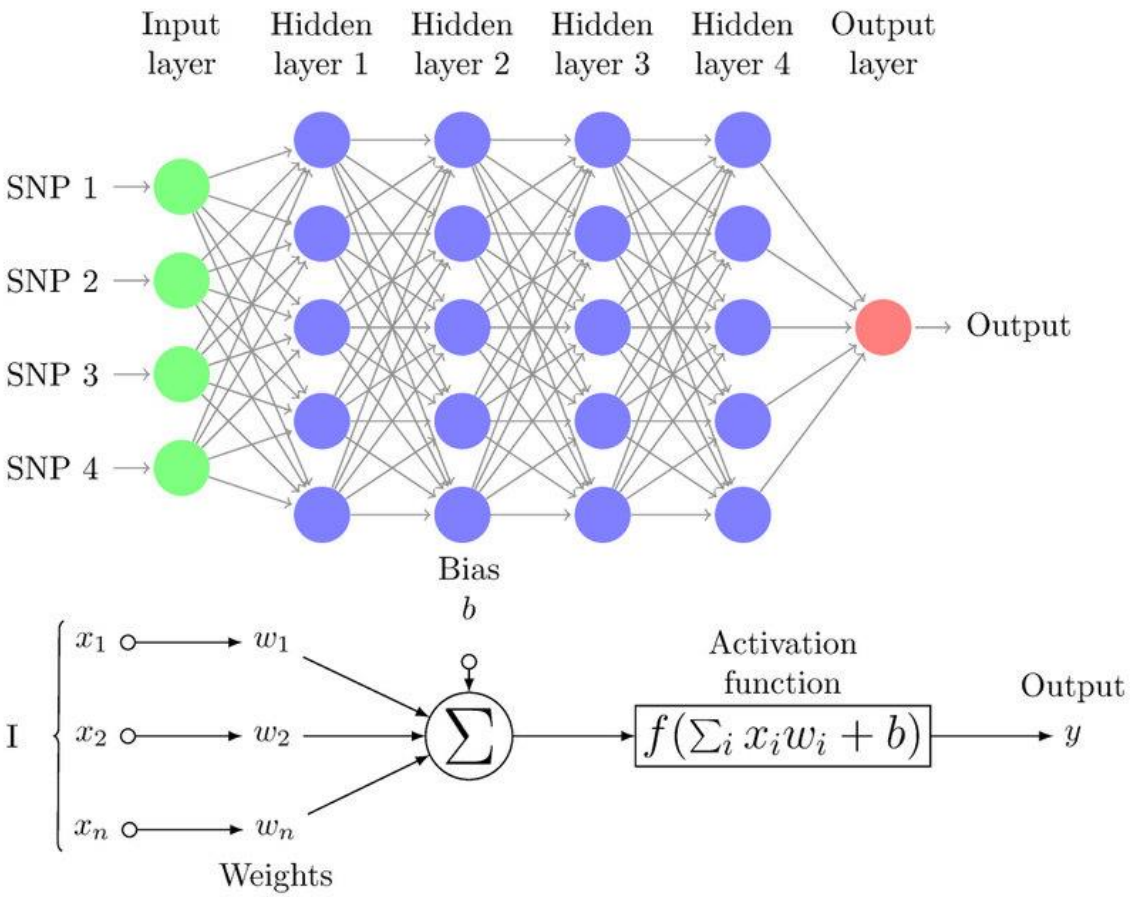
2. Feedforward – następnie dane przekazywane są w przód, do kolejnych neuronów gdzie zachodzą operacje opisane powyżej.
3. Obliczanie błędu – obliczanie błędu między wyliczonymi wynikami, a oczekiwaniami.
4. Backpropagation – po obliczeniu błędu, jest on przekazywany wstecz sieci do wyznacznie gradientów wag.
5. Aktualizacja wag – wyznaczanie gradientów które pomagają w minimalizacji błędów przez właśnie aktualizacje i dostosowanie wag do danych (np. Adam optimizer).
6. Powtórka procesu – aż do momentu osiągnięcia zamierzonego progu błędu
   Implementacja:

```
def MLP_architecture(neurons_lstm, neurons_dense1,
neurons_dense2, activation, l2_val):
    modelMLP = tf.keras.Sequential([
        tf.keras.layers.Dense(neurons_lstm,
input_shape=(X_train.shape[1],), activation=activation),
        tf.keras.layers.Dropout(l2_val),
        tf.keras.layers.Dense(neurons_dense1,
activation=activation),
        tf.keras.layers.Dropout(l2_val),
        tf.keras.layers.Dense(neurons_dense2,
activation=activation),
        tf.keras.layers.Dense(1)
    ])
    return modelMLP
```

- Sieć MLP v2 – opis dokładnie jak powyżej, różnica jedynie w liczbie warstw i neuronów
  Implementacja:

```
def MLP_architecture_v2(neurons_lstm, neurons_dense1, neurons_dense2,
neurons_dense3, neurons_dense4, activation, l2_val):
    modelMLPv2 = tf.keras.Sequential([
        tf.keras.layers.Dense(neurons_lstm,
input_shape=(X_train.shape[1],), activation=activation),
        tf.keras.layers.Dropout(l2_val),
        tf.keras.layers.Dense(neurons_dense1, activation=activation),
        tf.keras.layers.Dropout(l2_val),
        tf.keras.layers.Dense(neurons_dense2, activation=activation),
        tf.keras.layers.Dropout(l2_val),
        tf.keras.layers.Dense(neurons_dense3, activation=activation),
        tf.keras.layers.Dropout(l2_val),
        tf.keras.layers.Dense(neurons_dense4, activation=activation),
        tf.keras.layers.Dense(1)
    ])
    modelMLPv2.compile(loss='mse', optimizer='adam')
    return modelMLPv2
```

Architektura MLP:



Wywołania sieci MLP:

Wywołanie sieci MLP dla 3 warstw gęstych (warstwa w pełni połączona, w której każdy neuron jest połączony z każdym neuronem w warstwie poprzedniej i w warstwie następnej, klasyczny perceptron w którym manipulujemy wagami). Kolejno mamy 128, 64 i 32 neurony w każdej z tych warstw. Ostatecznie, sieć ma jedną warstwę wyjściową z jednym neuronem, która zwraca pojedynczą wartość wyjściową. Funkcja aktywacji nie jest podana dla ostatniej warstwy, ponieważ model ma zwrócić wartości liczbowe, a nie prawdopodobieństwa. Istnieją również inne paramaetry takie jak optymalizator „Adam” czy funkcja aktywacji „relu” oraz współczynnik regularyzacji wynoszący 0.01 i użyty w warstwie dropout, która ma za zadanie regularyzować sieć poprzez losowe wyłączanie neuronów podczas treningu, co pomaga w uniknięciu overfittingu. Czy też parametr batch_size który określa liczbę próbek, które są przetwarzane jednocześnie podczas jednej iteracji treningowej. Przetwarzanie danych w mniejszych partiach jest bardziej wydajne, ponieważ umożliwia to wykorzystanie pamięci GPU w bardziej efektywny sposób, a także zapewnia losowość w wyborze próbek do treningu.

```
model_MLP = MLP_architecture(neurons_lstm=128, neurons_dense1=64,
neurons_dense2=32, activation='relu', l2_val=0.01)
model_MLP.compile(
    optimizer='adam',
    loss='mse'
)
# "History" object contains information about the training process,
including the loss and metrics values recorded during training and
validation at each epoch.
history_MLP = model_MLP.fit(
    X_train_scaled,
    y_train_scaled,
    epochs=5,
    validation_data=(X_test_scaled, y_test_scaled),
    batch_size=32,
    validation_split=0.2
)
```

Kolejne wywołanie sieci MLP z taką samą konfiguracją tylko większą liczbą neuronów w każdej z nich; 128, 128 i 64 neurony:

```
model_MLP_more_neurons = MLP_architecture(neurons_lstm=128,
neurons_dense1=128, neurons_dense2=64, activation='relu', l2_val=0.01)
model_MLP_more_neurons.compile(
optimizer = 'adam',
loss = 'mse'
)
# "History" object contains information about the training process,
including the loss and metrics values recorded during training and
validation at each epoch.
history_MLP_more_neurons = model_MLP_more_neurons.fit(
X_train_scaled,
y_train_scaled,
epochs = 5,
validation_data = (X_test_scaled, y_test_scaled),
batch_size = 32,
validation_split = 0.2
)
```

Ostatnie wywołanie sieci MLP, dla większej liczby warstw z taką samą konfiguracją pozostałych współczynników:

```
model_MLP_v2 = MLP_architecture_v2(neurons_lstm=128, neurons_dense1=64,
neurons_dense2=32, neurons_dense3=16, neurons_dense4=16, activation='relu',
l2_val=0.01)
model_MLP_v2.compile(
    optimizer='adam',
    loss='mse'
)
history_MLP_v2 = model_MLP_v2.fit(
    X_train_scaled,
    y_train_scaled,
    epochs=5,
    validation_data=(X_test_scaled, y_test_scaled),
    batch_size=32,
    validation_split=0.2
)
```

- Sieć LSTM (Long Short-Term Memory) – W rekurencyjnych sieciach (RNN) każdy neuron przetwarza dane wejściowe wraz z danymi wyjściowymi z poprzedniej iteracji czasowej (sygnały z wyjścia sieci trafiają ponownie na jej wejścia, generując nowe sygnały aż do ustabilizowania się sygnałów wyjściowych w neuronie). W ten sposób sieć "pamięta" dane z poprzednich iteracji i może uwzględnić je w obliczeniach dla bieżącej iteracji. Zaś LSTM to specjalny rodzaj sieci rekurencyjnej, który został zaprojektowany w celu rozwiązania problemu zanikającego gradientu, który może wystąpić w standardowych RNN. W RNN, gradienty mogą zanikać, gdy sieć jest uczona na długich sekwencjach, ponieważ błąd propagowany jest przez wiele iteracji.
  W LSTM sieć przetwarza dane w postaci sekwencji (kolejnych wartości), zamiast pojedynczych wektorów (w przeciwieństwie do MLP), ale zamiast jednego stanu, posiada trzy specjalne bramki (w przeciwieństwie do RNN): zapominającą, wejściową i wyjściową.
  Bramki te pozwalają sieci na decyzję, które informacje należy przechować i uwzględnić w obliczeniach, a które odrzucić. Również są uczone przy pomocy propagacji wstecznej. Poprzez bramkowani, sieć jest w stanie przechowywać informacje przez dłuższy czas, filtrować niepotrzebne i dostosowywać się do zmieniających się warunków.
  - wejściowa – kontroluje, które informacje powinny zostać dodane do pamięci długoterminowej. Jej działanie opiera się na sigmoidalnej funkcji aktywacji, która przetwarza wektor wejściowy oraz wektor stanu ukrytego (hidden state) z poprzedniej jednostki. Wyjście z bramki wejściowej jest mnożone przez wektor wartości, które powinny zostać dodane do pamięci. Jeśli wartość jest bliska 1 to jest zatrzymywane, jeśli bliska 0 to „usuwana”.
  - zapomnienia – usuwa niepotrzebne informacje z pamięci długoterminowej, wykorzystuje te same wektory oraz funkcje sigmoidalną.
  - wyjściowa – decyduje które informacje powinny zostać przekazane na wyjście z jednostki LSTM
  Proces uczenia składa się z następujących kroków:
  1. Inicjalizacja wag – tak jak w MLP.
  2. FeedForward – Analogicznie jak w MLP, z tym że w przypadku LSTM, przetwarzanie danych odbywa się sekwencyjnie, tzn.: sieć otrzymuje kolejno po jednym elemencie sekwencji wejściowej. W każdej chwili czasowej sekwencja wejściowa przekazywana jest przez bramkę wejściową, która decyduje, które informacje powinny zostać zapamiętane, a które odrzucone.
  3. Obliczanie błędu - tak jak w MLP.
  4. Backprop – tu występuje odmiana tego procesu nazwana Backpropagation Through Time (BPTT) , który polega na propagacji błędu wstecz, aby wyznaczyć gradienty wag. Jednak w przeciwieństwie do MLP, LSTM musi uwzględnić wpływ każdego elementu sekwencji na cały wynik. W związku z tym stosuje się wspomnianą technikę BPTT, która polega na propagacji błędu wstecz przez całą sekwencję danych.
  5. Aktualizacja wag – tak jak w MLP.

6. Powtórka procesu - tak jak w MLP.

Implementacja:

```
def LSTM_architecture(neurons_lstm, neurons_dense, activation,
l2_val):
    model_LSTM = tf.keras.Sequential([
        tf.keras.layers.LSTM(neurons_lstm, input_shape=(1,
X_train.shape[-1])),
        tf.keras.layers.Dense(neurons_dense, activation=activation,
kernel_regularizer=tf.keras.regularizers.l2(l2_val)),
        tf.keras.layers.Dense(neurons_dense, activation=activation),
        tf.keras.layers.Dense(1)
    ])
    model_LSTM.compile(
        optimizer='adam',
        loss='mean_squared_error'
    )
    return model_LSTM
```

Wywołanie sieci – analogicznie jak w MLP:

```
model_LSTM = LSTM_architecture(neurons_lstm=64, neurons_dense=32,
activation='relu', l2_val=0.01)


# "History" object contains information about the training process,
including the loss and metrics values recorded during training and
validation at each epoch.
history_LSTM = model_LSTM.fit(
    X_train_3D,
    y_train_scaled,
    epochs=5,
    validation_data=(X_test_3D, y_test_scaled),
    batch_size=32,
    validation_split=0.2
)
```

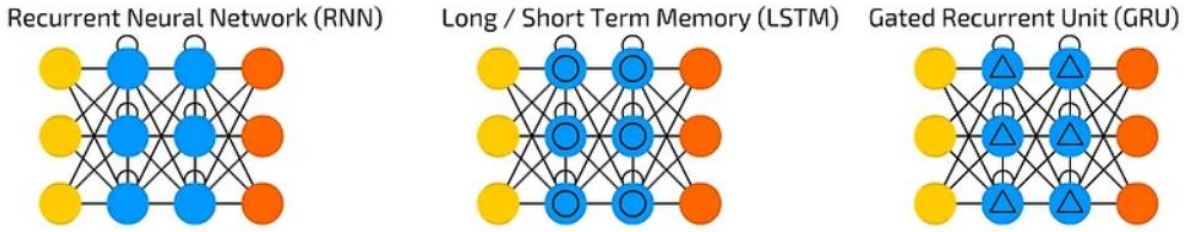
- Sieć GRU – Tak jak w LSTM, w GRU również występują bramki. Jednak występują tylko dwa rodzaje bramek: aktualizacji i resetowania.
  Bramka aktualizacji w sieci GRU jest kontrolerem, tego w jakim stopniu przeszła informacja z poprzedniego do obecnego stanu. Bramka ta decyduje, które wartości z poprzedniego stanu powinny być uwzględnione w obecnym stanie.
  Natomiast resetowania decyduje, jak bardzo poprzednie wartości wpłyną na obliczenia aktualny stan.
  Podobnie jak w sieciach LSTM, w GRU wyjście z jednego stanu stanowi wejście dla kolejnego stanu. Jednak w przeciwieństwie do sieci LSTM, sieć GRU korzysta tylko z jednego ukrytego stanu, co ułatwia jej implementację i szkolenie.
  Proces uczenia składa się z następujących kroków:
  Analogicznie jak w LSTM. Różnica istnieje jedynie w ilości bramek i drobnych szczegółach. Sieć GRU jest szybsza i prostsza, natomiast LSTM jest bardziej wydajna i może lepiej radzić sobie z skomplikowanymi wzorcami szeregowymi, sekwencyjnymi.

Implementacja:

```
def GRU_architecture(neurons_lstm=128, neurons_dense=64,
activation='relu', l2_val=0.01):
    model_GRU = tf.keras.Sequential([
        tf.keras.layers.GRU(neurons_lstm, input_shape=(1,
X_train.shape[-1])),
        tf.keras.layers.Dense(neurons_dense, activation=activation ,
kernel_regularizer=tf.keras.regularizers.l2(l2_val)),
        tf.keras.layers.Dense(1)
    ])
    model_GRU.compile(
        optimizer='adam',
        loss='mse'
    )
    return model_GRU
```

Wszystkie sieci zostały wywołane dla tej samej liczby epok (5, dla krótkiego czasu działania), tak aby jak najlepiej sprawdzić ich wyniki i je porównać ze sobą – dać im tyle samo czasu na analizę danych. Mimo wszystko wszystkie sieci osiągnęły raczaj dobre wyniki dla jednak trochę skąpych danych (jedynie 6 kolumn uczących, z których tak naprawdę 3 miały sensowną korelację z ceną).

Architektura RNN, LSTM oraz GRU:



Żółty – output cel, niebieski – recurrent cel, pomarańczowy – output cel, kółko – memory cel, trójkąt – different memory cell

Wywołanie sieci analogiczne jak w przypadku powyższych architektur:

```
model_GRU = GRU_architecture(neurons_lstm=128, neurons_dense=64,
activation='relu', l2_val=0.01)


# "History" object contains information about the training process,
including the loss and metrics values recorded during training and
validation at each epoch.
history_GRU = model_GRU.fit(
    X_train_3D,
    y_train_scaled,
    epochs=5,
    validation_data=(X_test_3D, y_test_scaled),
    batch_size=32,
    validation_split=0.2
)
```

Taka sama konfiguracja poza liczbą neuronów i liczbą warstw neuronów wynika z faktu chęci przeprowadzenia eksperymentu, czy liczba neuronów i warstw ma wpływ na na uzyskane wyniki. Pozwoliło to też na znormalizowanie rozbieżności architektur do jakiegoś wspólnego mianownika i obiektywne porównanie wyników.

Napisaliśmy również przy pomocy Keras Tunnera funkcje której zadaniem było określenie najlepszych hiperparametrów dal zadanej sieci neuronowej.

4. Dyskusja wyników oraz wnioski

Poniżej zestawiam wyniki ewaluacji dla różnych typów oraz architektur sieci neuronowych:

	Model	LSTM	MLP	GRU	MLP_n	MLP_v2
0	Test Loss	0.157339	0.122684	0.171457	0.124195	0.133567
1	Train Loss	0.148817	0.111648	0.156005	0.110768	0.117514
2	Test RMSE	17722.218448	16105.330624	18446.676204	16204.211709	16804.464315
3	Train RMSE	17206.361889	15363.889574	17538.885756	15303.230076	15762.351432
4	Test MAE	10758.804800	9608.717833	11749.649902	9762.929915	10030.230648
5	Train MAE	10495.009111	9191.617924	11255.889124	9369.289170	9485.370443
6	Test Pearson Coef	0.928197	0.937557	0.915900	0.937316	0.934641
7	Train Pearson Coef	0.935102	0.943057	0.924487	0.944440	0.941952
8	Test % Error	0.258669	0.237804	0.288580	0.267353	0.252003
9	Train % Error	0.259011	0.226996	0.287611	0.266490	0.239292

Opis wybranych metryk:

Test Loss i Train Loss odnoszą się do funkcji straty (ang. loss function) używanej podczas procesu uczenia, a więc mierzą, jak dobrze model dopasował się do danych treningowych i testowych. W przypadku funkcji straty, im mniejsza wartość, tym lepiej.

Test RMSE (Root Mean Squared Error) i Train RMSE to metryki, które mierzą różnicę między rzeczywistą wartością a wartością przewidywaną przez model. RMSE mierzy średni błąd kwadratowy, a więc im mniejsza wartość, tym lepiej model przewiduje wyniki.

Test MAE (Mean Absolute Error) i Train MAE to inna metryka, która mierzy średnią różnicę między wartością przewidywaną przez model a rzeczywistą wartością. MAE jest bardziej odporny na wartości odstające niż RMSE i jest bardziej interpretowalny. Im mniejsza wartość MAE, tym lepiej model przewiduje wyniki.

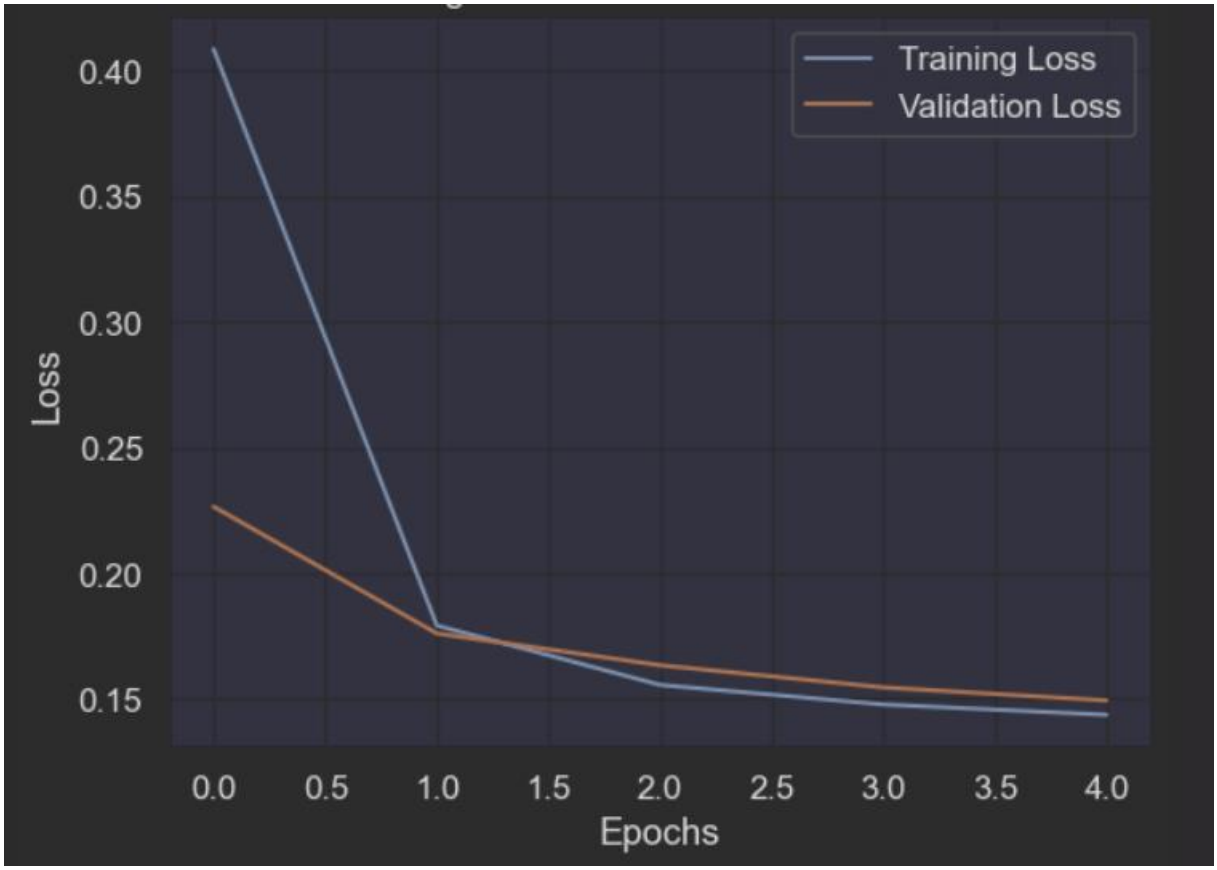
Test Pearson Coef i Train Pearson Coef odnoszą się do korelacji między rzeczywistymi wartościami a przewidywanymi przez model. Wartości korelacji wahać się będą w przedziale od -1 do 1, gdzie wartości bliższe 1 wskazują na silną korelację.

Test % Error i Train % Error to metryki, które mierzą procentową różnicę między rzeczywistą wartością a wartością przewidywaną przez model. Im mniejsza wartość % Error, tym lepiej model przewiduje wyniki.

Opis wyników:
Na przykład, wyniki dla RMSE są na poziomie 16 000-18 000, co oznacza, że przeciętny błąd przewidywania dla danych testowych wynosi właśnie 16 000-18 000. Ponadto, wyniki dla współczynnika korelacji Pearsona są na poziomie 0.92-0.94, co sugeruje silną korelację między wartościami przewidywanymi, a rzeczywistymi wartościami. W tabeli wyników podano wartości błędu procentowego dla każdego z modeli. Jak można zauważyć, wyniki wahają się w granicach 0.22-0.25, co sugeruje, że modele mają przeciętnie około 22-25% błąd w przewidywaniu wartości dla danych testowych.

Myślę że zwiększenie epok, wykorzystanie wyników KerasTunnera oraz manipulacja innymi parametrami zdecydowanie mogło by wpłynąć znacznie lepiej na i tak dobre wyniki – nawet dla relatywnie małej kolumn uczących (jedynie 6).

Ponad to wykresy uzyskane na wskutek wyników są dobre co najmniej, między innymi: funkcja kosztu dla kolejnych epok, dla danych treningowych i uczących pokrywają się od pewnego momentu co sugeruje good fitting modelu.



Całość projektu jest dostępna na GitHubie.

Dalsza propozycje rozwoju:

Przede wszystkim uważamy że można poprawić samo wykonanie modelu, zamknięcie większej ilości kodu w funkcje i mniejsza powtarzalność. Ponad to poprawić działanie

funkcji na bazie Keras Tunner ponieważ obecna zwraca dość dziwne wyniki optymalnych parametrów z danego gridu inputowego parametrów.

Kolejno, dość oczywista opcja, można by udoskonalić model poprawę skuteczności. Można by również rozszerzyć zakresy poszukiwań na więcej marek czy serwisów internetowych. Można by również model zintegrować z jakąś aplikacją webową, która umożliwi użytkownikom przewidywanie cen samochodów na podstawie danych wprowadzonych przez użytkownika, czy też stworzenie REST API, które będzie umożliwiało innym aplikacjom korzystanie z modelu przewidującego ceny. Można by również rozpiąć model na serwery/klastry podpiąć pod niego większe zasoby co zwiększy moc obliczeniową, przyspieszy działanie i potencjalnie zwróci dokładniejsze wyniki.

Można, by również wdrożyć model przewidujący ceny samochodów Volkswagen w kontenerze Dockera przykładowo, co pozwoli na łatwe uruchamianie i skalowanie aplikacji na różnych środowiskach.

Finalnie największym i najciekawszym według nas pomysłem, który mógłby podsumować powyższe sugestie, było by opracowanie systemu rekomendacji, który pozwoli na sugestie najlepszych ofert dla użytkowników, biorąc pod uwagę ich preferencje i budżet.