In [ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call driv
e.mount("/content/drive", force_remount=True).
```

In [ ]:
```
df = pd.read_csv('/content/drive/MyDrive/CI/online+news+popularity/OnlineNewsPopula
df.describe()
```

Out[ ]:

	timedelta	n_tokens_title	n_tokens_content	n_unique_tokens	n_non_stop_words	n_non_
count	39644.000000	39644.000000	39644.000000	39644.000000	39644.000000	
mean	354.530471	10.398749	546.514731	0.548216	0.996469	
std	214.163767	2.114037	471.107508	3.520708	5.231231	
min	8.000000	2.000000	0.000000	0.000000	0.000000	
25%	164.000000	9.000000	246.000000	0.470870	1.000000	
50%	339.000000	10.000000	409.000000	0.539226	1.000000	
75%	542.000000	12.000000	716.000000	0.608696	1.000000	
max	731.000000	23.000000	8474.000000	701.000000	1042.000000	

8 rows × 60 columns

Podstawowe informacje

Dane poddane analizie pochodzą ze strony z gatunku rozrywki i informacji - mashable.com. Zostały one zebrane przez 4 osoby niezwiązane ze stroną źródłową.

Głównym celem tworzenia zbioru danych było przewidywanie zasięgów poszczególnych artykułów poprzez liczbę udostępnień danego artykułu.

Charakterystyka danych

Liczba instancji - 39797

Liczba atrybutów - 61 - w tym

binarne

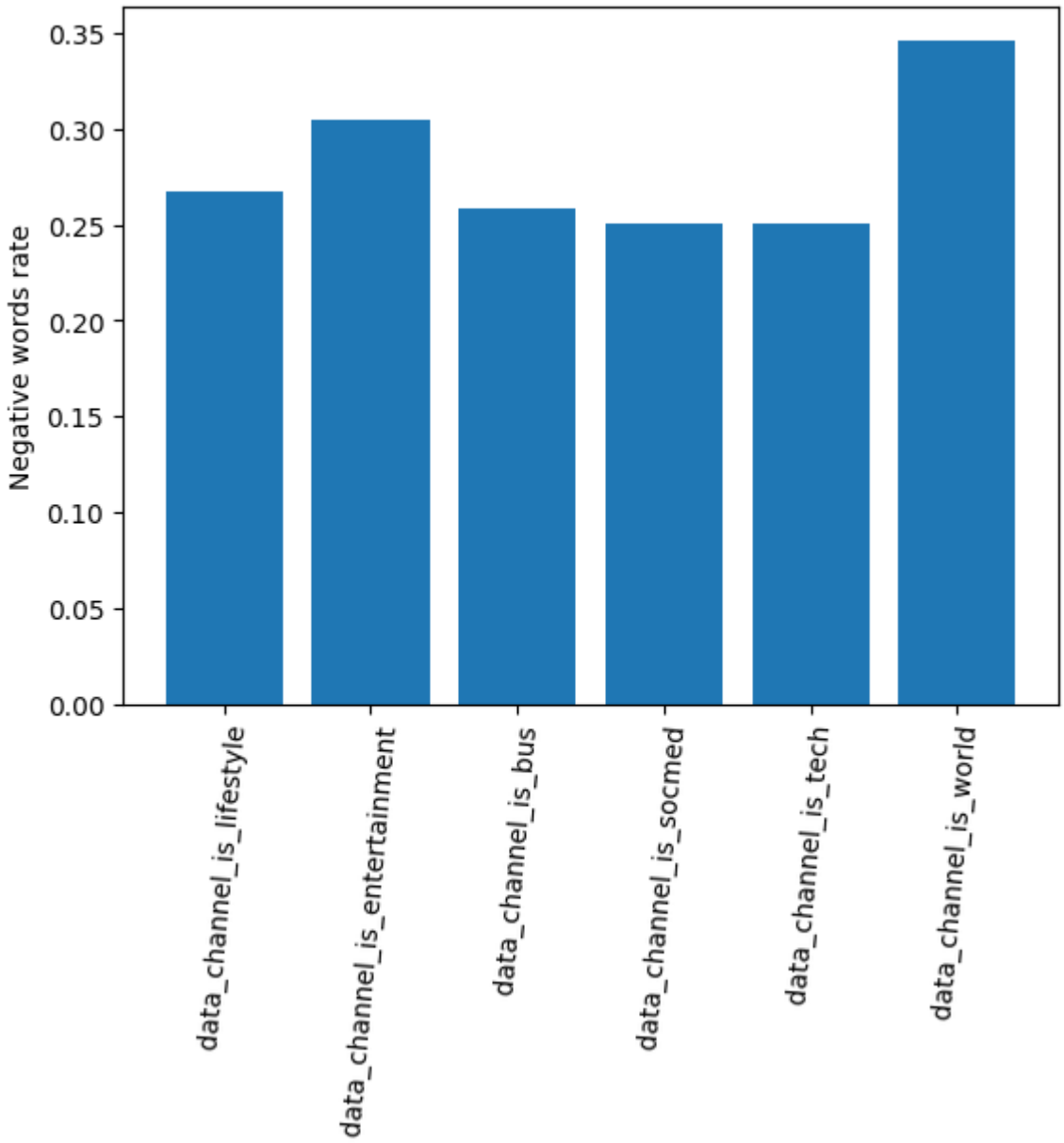
liczbowe

nominalne

Brakujące wartości - brak

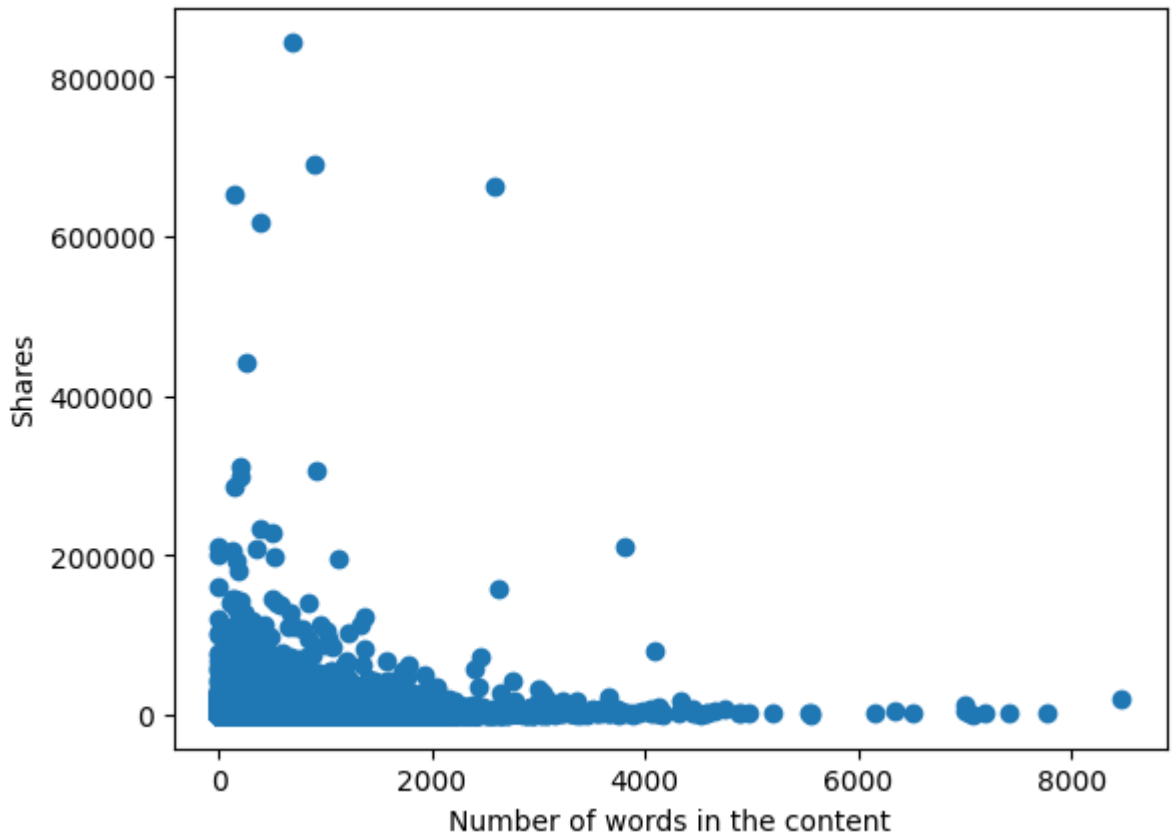
Wizualizacja danych

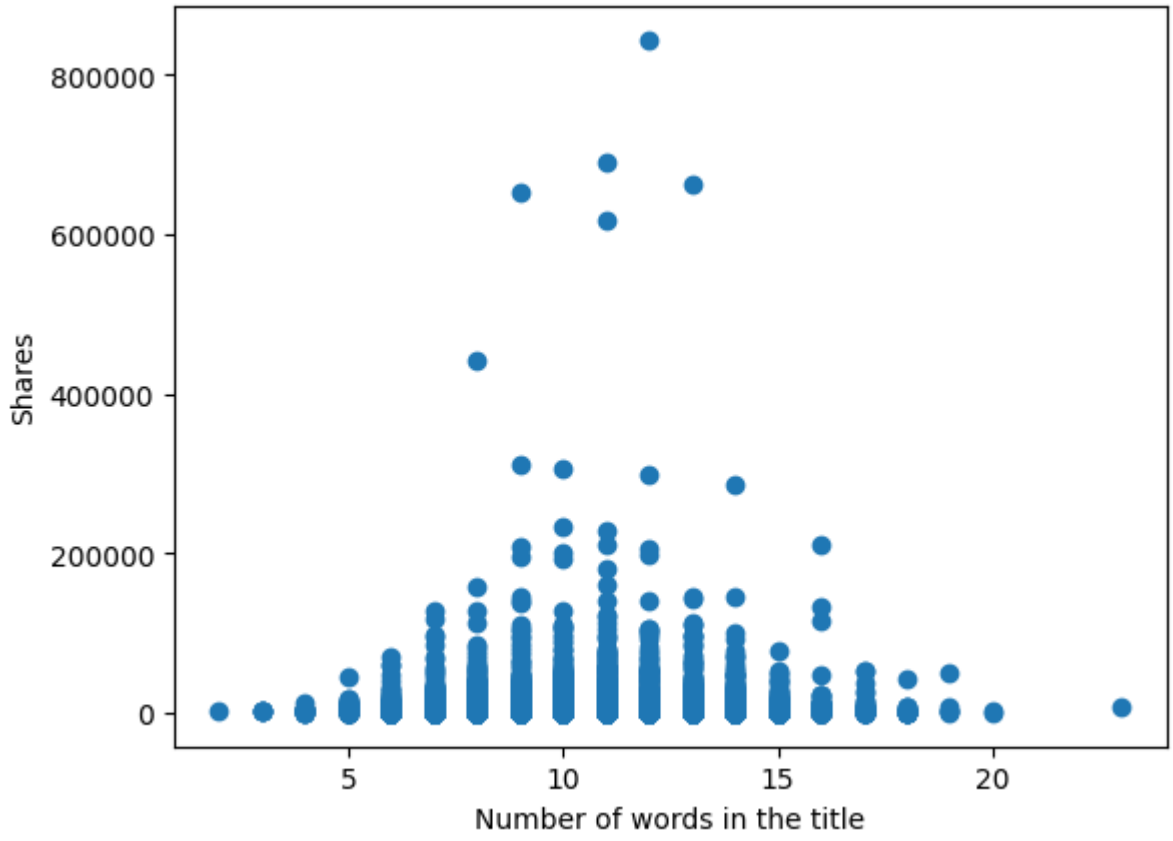
In [ ]:
```
x_axis_labels = [' data_channel_is_lifestyle', ' data_channel_is_entertainment',
                 ' data_channel_is_bus',  ' data_channel_is_socmed',
                 ' data_channel_is_tech', ' data_channel_is_world']

mean = []

for channel in x_axis_labels:
    filtered_data = df[df[channel] == 1]
    mean_value = filtered_data[' rate_negative_words'].mean()
    mean.append(mean_value)

plt.xticks(rotation=85)
plt.bar(x_axis_labels, mean)
plt.ylabel('Negative words rate')
plt.show()
```



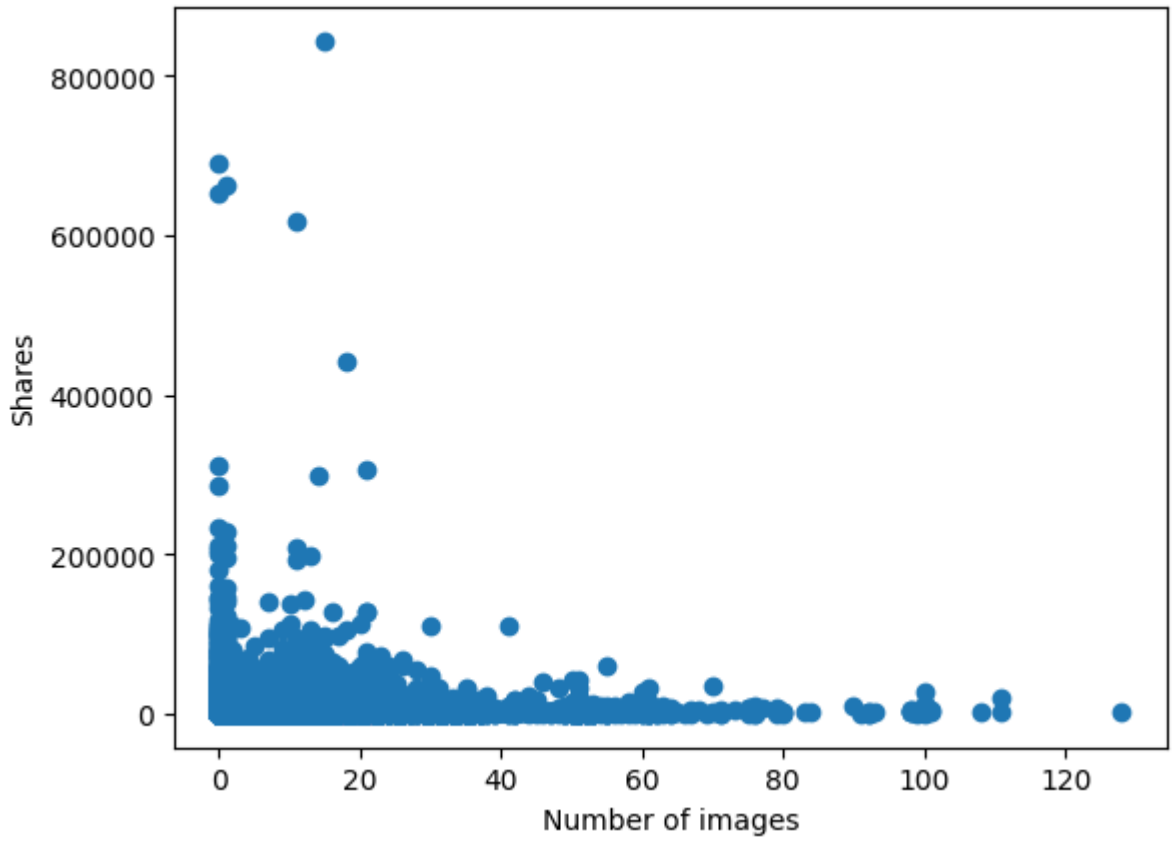
In [ ]:
```
plt.scatter(df[' n_tokens_content'], df[' shares'])
plt.xlabel('Number of words in the content')
plt.ylabel('Shares')
plt.show()

print()

plt.scatter(df[' n_tokens_title'], df[' shares'])
```

```
plt.xlabel('Number of words in the title')
plt.ylabel('Shares')
plt.show()
```





In [ ]:
```
plt.scatter(df[' num_imgs'], df[' shares'])
plt.xlabel('Number of images')
plt.ylabel('Shares')
plt.show()
```



Przygotowanie danych

In [ ]:
```
#Sprawdzamy kompletność danych
df.isnull().sum()
```

Out[ ]:
```
url                             0
 timedelta                      0
 n_tokens_title                 0
 n_tokens_content               0
 n_unique_tokens                0
                               ..
 title_subjectivity             0
 title_sentiment_polarity       0
 abs_title_subjectivity         0
 abs_title_sentiment_polarity   0
 shares                         0
Length: 61, dtype: int64
```

In [ ]:
```
df.describe()
```

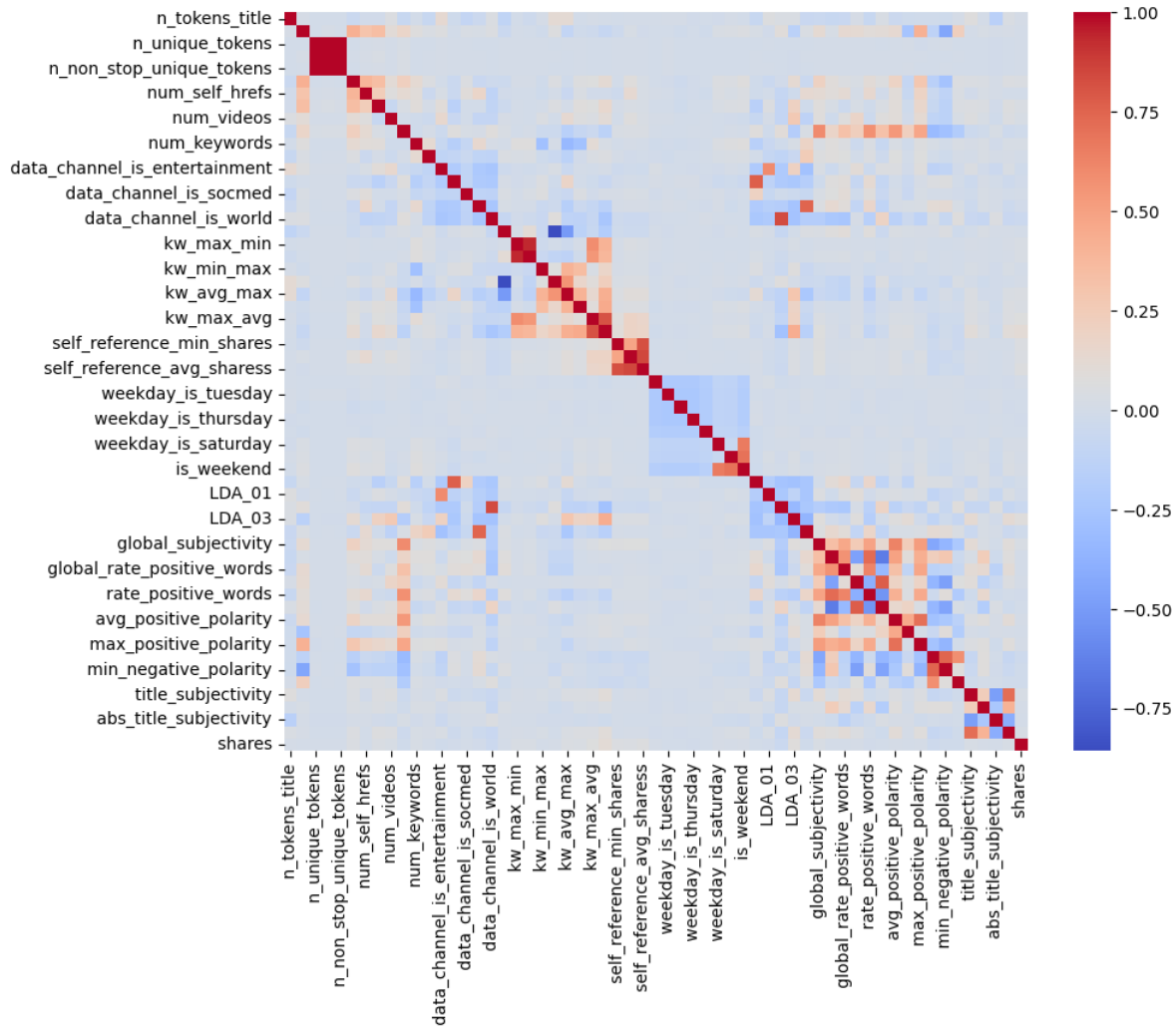
Out[ ]:

	timedelta	n_tokens_title	n_tokens_content	n_unique_tokens	n_non_stop_words	n_non_
count	39644.000000	39644.000000	39644.000000	39644.000000	39644.000000	
mean	354.530471	10.398749	546.514731	0.548216	0.996469	
std	214.163767	2.114037	471.107508	3.520708	5.231231	
min	8.000000	2.000000	0.000000	0.000000	0.000000	
25%	164.000000	9.000000	246.000000	0.470870	1.000000	
50%	339.000000	10.000000	409.000000	0.539226	1.000000	
75%	542.000000	12.000000	716.000000	0.608696	1.000000	
max	731.000000	23.000000	8474.000000	701.000000	1042.000000	

8 rows × 60 columns

In [ ]:
```
# Usuwamy zbędne kolumny
df = df.drop('url', axis=1)
df = df.drop(' timedelta', axis=1)
```

In [ ]:
```
# Tworzymy mapę korelacji
import seaborn as sns
import matplotlib.pyplot as plt

corr_matrix = df.corr()

plt.figure(figsize=(10, 8))
sns.heatmap(corr_matrix, annot=False, cmap='coolwarm')
plt.show()

# znajdywanie 10 najlepszych korelacji
corr_last_column = corr_matrix[df.columns[-1]]
top_corr = corr_last_column.sort_values(ascending=False).head(11)
print(top_corr)

top_corr = corr_last_column.sort_values(ascending=True).head(11)
print(top_corr)
```




```
shares                          1.000000
kw_avg_avg                      0.110413
LDA_03                          0.083771
kw_max_avg                      0.064306
self_reference_avg_sharess      0.057789
self_reference_min_shares       0.055958
self_reference_max_shares       0.047115
num_hrefs                       0.045404
kw_avg_max                      0.044686
kw_min_avg                      0.039551
num_imgs                        0.039388
Name:  shares, dtype: float64
 LDA_02                           -0.059163
 data_channel_is_world            -0.049497
 avg_negative_polarity            -0.032029
 average_token_length             -0.022007
 max_negative_polarity            -0.019300
 min_negative_polarity            -0.019297
 data_channel_is_entertainment    -0.017006
 LDA_04                           -0.016622
 data_channel_is_tech             -0.013253
 rate_positive_words              -0.013241
 data_channel_is_bus              -0.012376
Name:  shares, dtype: float64
```

Jak widać na mapie korelacji w zbiorze danych, liczba udostępnień danego artykułu jest bardzo słabo skorelowana z pozostałymi atrybutami. Z powyższych obserwacji wynika prawdopodobna niska skuteczność naszej sieci.

Podział danych

Ogólnie przyjętą praktyką jest stosowanie proporcji w zakresie 60-80% danych treningowych, 10-20% danych walidacyjnych i 10-20% danych testowych.

W naszym projekcie najlepsze rezultaty otrzymaliśmy przy wyborze:

- 70% - dane treningowe
- 15% - dane walidujące
- 15% - dane testowe

In [ ]:
```
# Usuwamy outliery
Q1 = df[' shares'].quantile(0.25)
Q3 = df[' shares'].quantile(0.75)
IQR = Q3 - Q1

filter = (df[' shares'] >= Q1 - 1.5 * IQR) & (df[' shares'] <= Q3 + 1.5 *IQR)
df = df.loc[filter]
```

Na podstawie macierzy korelacji wybieramy te kolumny, które w jakikolwiek sposób wykazują korelację z liczbą udostępnień artykułu.

In [ ]:
```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error

# Tworzymy tablicę z danymi wyjściowymi
number_of_shares = df[' shares']

# Wybieramy kolumny na których będziemy uczyć naszą sieć
print(df.columns)

cols_to_take = [' num_hrefs', ' data_channel_is_entertainment',
                ' data_channel_is_socmed', ' data_channel_is_tech',
                ' data_channel_is_world', ' kw_min_avg', ' kw_avg_avg',
                ' weekday_is_friday', ' is_weekend', ' LDA_00', ' LDA_01',
                ' LDA_02', ' LDA_04']

df = df[cols_to_take]

# Wybieramy kolumny do normalizacji
# (nie uwzględniamy kolumn z wearościami binarnymi)
cols_to_normalize = [' kw_min_avg', ' kw_avg_avg', ' LDA_00',
                     ' LDA_01', ' LDA_02', ' LDA_04']

# Normalizacja danych
scaler = StandardScaler()
df[cols_to_normalize] = scaler.fit_transform(df[cols_to_normalize])
scaler_shares = StandardScaler()
number_of_shares = scaler_shares.fit_transform(
    number_of_shares.values.reshape(-1, 1))

# Podział zbioru danych na dane treningowe,
# walidacyjne i testowe w proporcji 70%-15%-15%
data_train, data_rest, shares_train, shares_rest = train_test_split(
    df, number_of_shares, test_size=0.3, random_state=43)
data_test, data_validate, shares_test, shares_validate = train_test_split(
    data_rest, shares_rest, test_size=0.5, random_state=43)
```

```
Index([' n_tokens_title', ' n_tokens_content', ' n_unique_tokens',
       ' n_non_stop_words', ' n_non_stop_unique_tokens', ' num_hrefs',
       ' num_self_hrefs', ' num_imgs', ' num_videos', ' average_token_length',
       ' num_keywords', ' data_channel_is_lifestyle',
       ' data_channel_is_entertainment', ' data_channel_is_bus',
       ' data_channel_is_socmed', ' data_channel_is_tech',
       ' data_channel_is_world', ' kw_min_min', ' kw_max_min', ' kw_avg_min',
       ' kw_min_max', ' kw_max_max', ' kw_avg_max', ' kw_min_avg',
       ' kw_max_avg', ' kw_avg_avg', ' self_reference_min_shares',
       ' self_reference_max_shares', ' self_reference_avg_sharess',
       ' weekday_is_monday', ' weekday_is_tuesday', ' weekday_is_wednesday',
       ' weekday_is_thursday', ' weekday_is_friday', ' weekday_is_saturday',
       ' weekday_is_sunday', ' is_weekend', ' LDA_00', ' LDA_01', ' LDA_02',
       ' LDA_03', ' LDA_04', ' global_subjectivity',
       ' global_sentiment_polarity', ' global_rate_positive_words',
       ' global_rate_negative_words', ' rate_positive_words',
       ' rate_negative_words', ' avg_positive_polarity',
       ' min_positive_polarity', ' max_positive_polarity',
       ' avg_negative_polarity', ' min_negative_polarity',
       ' max_negative_polarity', ' title_subjectivity',
       ' title_sentiment_polarity', ' abs_title_subjectivity',
       ' abs_title_sentiment_polarity', ' shares'],
      dtype='object')
```

```
<ipython-input-24-b38178f510d9>:26: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stabl
e/user_guide/indexing.html#returning-a-view-versus-a-copy
  df[cols_to_normalize] = scaler.fit_transform(df[cols_to_normalize])
```

In [ ]:
```
print(data_train.shape)
print(number_of_shares.shape)
print(min(number_of_shares))
print(max(number_of_shares))
df.head()
```

```
(24572, 13)
(35103, 1)
[-1.51468194]
[3.46998132]
```

Out[ ]:

	num_hrefs	data_channel_is_entertainment	data_channel_is_socmed	data_channel_is_tech	data_ch
0	4.0	1.0	0.0	0.0	
1	3.0	0.0	0.0	0.0	
2	3.0	0.0	0.0	0.0	
3	9.0	1.0	0.0	0.0	
4	19.0	0.0	0.0	1.0	

Pierwsza próba stworzenia sieci

Przy pierwszej próbie tworzenia modelu sieci zdecydowaliśmy się na model 3 - warstwowy z jedną warstwą ukrytą o 100 neuronach. By uniknąć zjawiska overfittingu ustawiliśmy parametr "Dropout", tak by 20% neuronów zostało pominiętych w danym kroku uczenia się

sieci. W warstwie ukrytej, jak i wejściowej funkcja aktywacji została ustawiona na "ReLU" która stosowana jest do uczenia się nieliniowych zależności między parametrami.

In [ ]:
```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping

# Zdefiniuj model
model = Sequential()

# Dodaj warstwy
model.add(
  Dense(
  500,input_dim=data_train.shape[1],activation='relu',kernel_regularizer='l2'))
#model.add(Dropout(0.2))
model.add(Dense(100, activation='relu'))  # Warstwa ukryta
model.add(Dropout(0.2))
model.add(Dense(1, activation='linear'))  # Warstwa wyjściowa

# Kompiluj model
model.compile(loss='mean_squared_error', optimizer='adam')

# Dodaj Early Stopping
early_stopping = EarlyStopping(monitor='val_loss', patience=10)
```

Trening sieci

Do uczenia naszej sieci wykorzystujemy algorytm optymalizatora "Adam" - łączy on cechy Stochastic Gradient Descent z adaptacyjnym skalowaniem współczynników uczenia się. Model uczy się na przestrzeni 10 epok, w przypadku większej ilości nie było widocznej poprawy dokładności sieci. Argument batch_size wskazuje co ile próbek danych wagi dopasowania są aktualizowane.

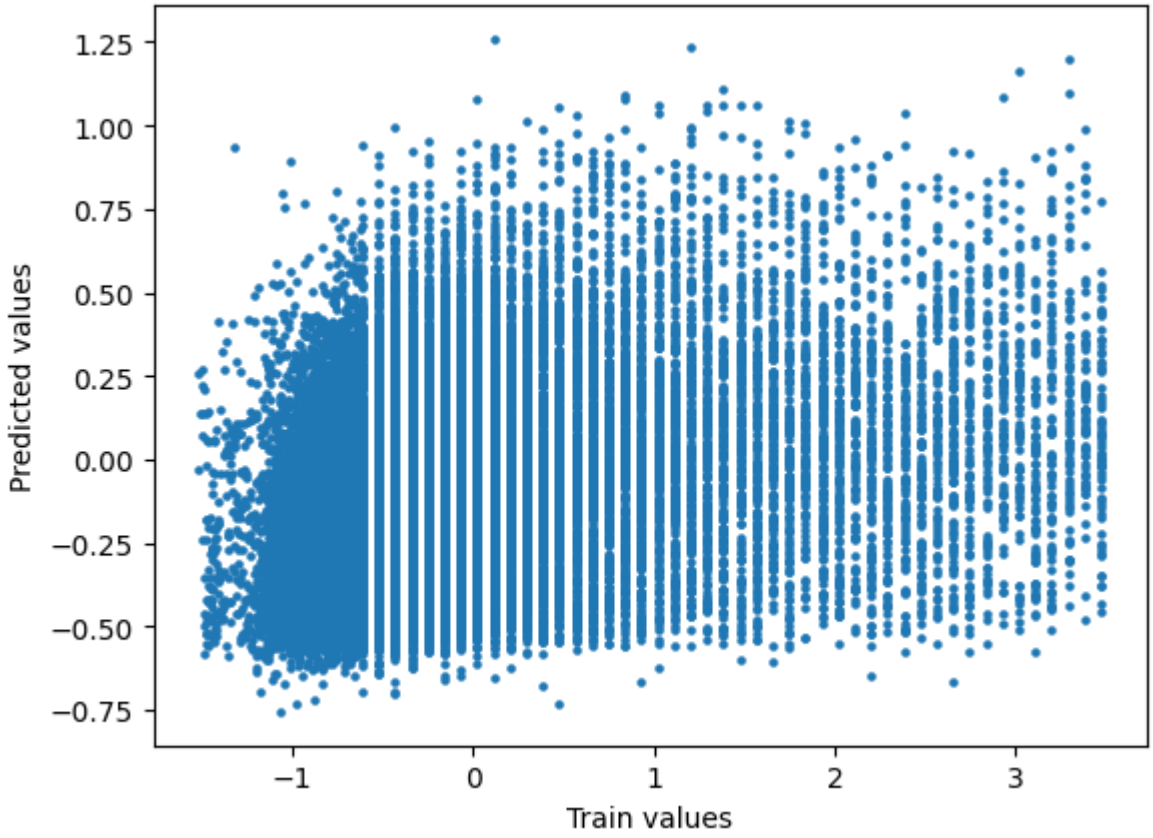
In [ ]:
```
# Trenuj model
model.fit(data_train, shares_train, epochs=10, batch_size=32, validation_data=(
    data_validate, shares_validate), callbacks=[early_stopping])
```

```
Epoch 1/10
768/768 [==============================] - 2s 2ms/step - loss: 1.0437 - val_loss:
0.9379
Epoch 2/10
768/768 [==============================] - 2s 2ms/step - loss: 0.9559 - val_loss:
0.9148
Epoch 3/10
768/768 [==============================] - 2s 3ms/step - loss: 0.9457 - val_loss:
0.9169
Epoch 4/10
768/768 [==============================] - 2s 2ms/step - loss: 0.9383 - val_loss:
0.9158
Epoch 5/10
768/768 [==============================] - 1s 2ms/step - loss: 0.9319 - val_loss:
0.8932
Epoch 6/10
768/768 [==============================] - 1s 2ms/step - loss: 0.9242 - val_loss:
0.8990
Epoch 7/10
768/768 [==============================] - 1s 2ms/step - loss: 0.9191 - val_loss:
0.9149
Epoch 8/10
768/768 [==============================] - 1s 2ms/step - loss: 0.9186 - val_loss:
0.8896
Epoch 9/10
768/768 [==============================] - 1s 2ms/step - loss: 0.9142 - val_loss:
0.9030
Epoch 10/10
768/768 [==============================] - 1s 2ms/step - loss: 0.9129 - val_loss:
0.8900
```

Out[ ]:
```
<keras.src.callbacks.History at 0x7be1e36c7cd0>
```

In [ ]:
```
predicted_shares_train = model.predict(data_train)
plt.scatter(shares_train, predicted_shares_train, s=5)
plt.xlabel('Train values')
plt.ylabel('Predicted values')
plt.show()

print(
f"Mean squared error:{mean_squared_error(shares_train,predicted_shares_train)}")
```

```
768/768 [==============================] - 1s 2ms/step
```

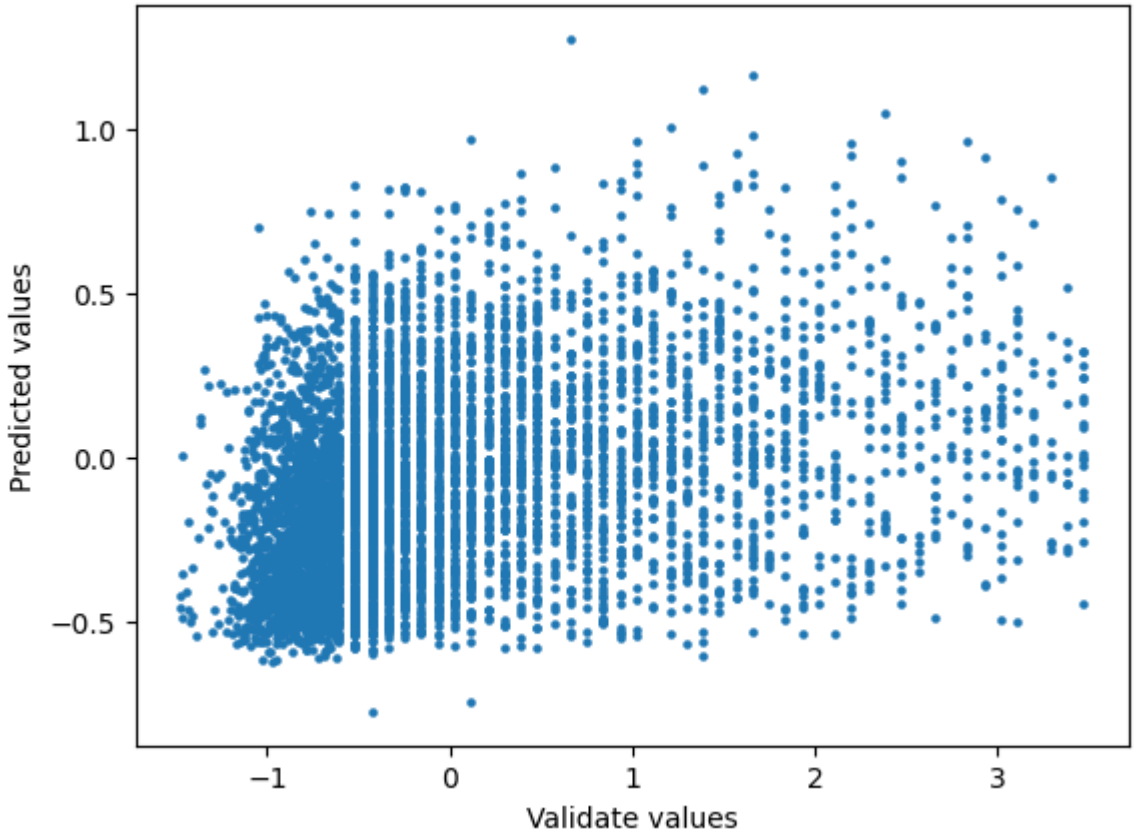



```
Mean squared error: 0.9051027933732326
```

Wykres powyżej zawiera zestawienie wartości faktycznych z wartościami przewidzianymi z wartości treningowych. Brak liniowości tego wykresu świadczy o tym, że nasza sieć się nie uczy.

In [ ]:
```
# Przewidywanie danych
predicted_shares = model.predict(data_validate)

# Tworzenie wykresu
plt.scatter(shares_validate, predicted_shares, s=5)
plt.xlabel('Validate values')
plt.ylabel('Predicted values')
plt.show()

print(
f"Mean squared error:{mean_squared_error(shares_validate, predicted_shares)}")
```

```
165/165 [==============================] - 1s 6ms/step
```




```
Mean squared error: 0.8884951562172818
```

Jak widać powyżej na wykresie przewidywanych wartości "shares" dokładność sieci jest bardzo słaba. Spowodowane jest to prawdopodobnie niską korelacją między atrybutami w zbiorze danych i przewidywanej ilości udostępnień.

Druga próba stworzenia sieci - dostrajanie parametrów

W drugim modelu sieci zdecydowaliśmy się na wariant 4 warstwowy z dwoma warstwami ukrytymi po odpowiednio 200 i 500 neuronów w każdej z nich. Wybór takich wartości podyktowany był testami, w których najlepsze wyniki osiągaliśmy przy takiej dystrybucji neuronów. Oprócz zmian w warstwach ukrytych, zwiększyliśmy liczbę epok z 10 do 15 co nieznacznie polepszyło dokładność naszego modelu. Parametr "Dropout" pozostał bez zmian na poziomie 20%, tak jak w pierwotnej wersji sieci.

In [ ]:
```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping

# Zdefiniuj model
model = Sequential()

# Dodaj warstwy
model.add(
Dense(
1000, input_dim=data_train.shape[1],activation='relu',kernel_regularizer='l2'))
#model.add(Dropout(0.2))  # Warstwa Dropout
model.add(Dense(200, activation='relu'))  # Warstwa ukryta
model.add(Dense(500, activation='relu'))  # Warstwa ukryta
```

```
model.add(Dropout(0.2))  # Warstwa Dropout
model.add(Dense(1, activation='linear'))  # Warstwa wyjściowa

# Kompiluj model
model.compile(loss='mean_squared_error', optimizer='adam')

# Dodaj Early Stopping
early_stopping = EarlyStopping(monitor='val_loss', patience=10)

# Trenuj model
model.fit(
    data_train, shares_train, epochs=15, batch_size=32, validation_data=(
        data_validate, shares_validate), callbacks=[early_stopping])
```

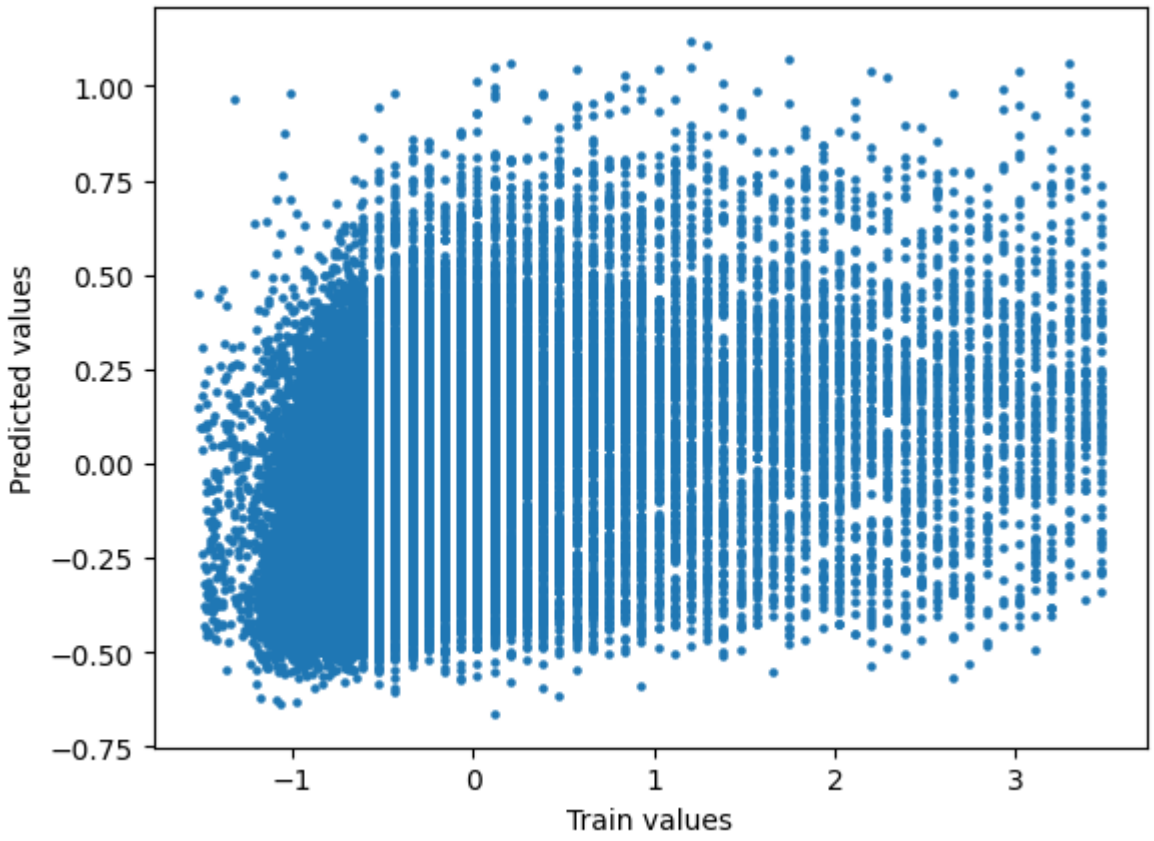
```
Epoch 1/15
768/768 [==============================] - 10s 11ms/step - loss: 0.9980 - val_los
s: 0.9238
Epoch 2/15
768/768 [==============================] - 6s 8ms/step - loss: 0.9549 - val_loss:
0.9502
Epoch 3/15
768/768 [==============================] - 9s 11ms/step - loss: 0.9366 - val_loss:
0.9087
Epoch 4/15
768/768 [==============================] - 6s 8ms/step - loss: 0.9288 - val_loss:
0.8977
Epoch 5/15
768/768 [==============================] - 7s 9ms/step - loss: 0.9258 - val_loss:
0.9027
Epoch 6/15
768/768 [==============================] - 8s 10ms/step - loss: 0.9239 - val_loss:
0.8990
Epoch 7/15
768/768 [==============================] - 6s 8ms/step - loss: 0.9187 - val_loss:
0.8910
Epoch 8/15
768/768 [==============================] - 8s 11ms/step - loss: 0.9155 - val_loss:
0.8916
Epoch 9/15
768/768 [==============================] - 6s 8ms/step - loss: 0.9149 - val_loss:
0.8931
Epoch 10/15
768/768 [==============================] - 8s 11ms/step - loss: 0.9122 - val_loss:
0.8851
Epoch 11/15
768/768 [==============================] - 7s 9ms/step - loss: 0.9105 - val_loss:
0.8851
Epoch 12/15
768/768 [==============================] - 7s 9ms/step - loss: 0.9102 - val_loss:
0.8859
Epoch 13/15
768/768 [==============================] - 8s 10ms/step - loss: 0.9104 - val_loss:
0.9015
Epoch 14/15
768/768 [==============================] - 6s 8ms/step - loss: 0.9086 - val_loss:
0.8863
Epoch 15/15
768/768 [==============================] - 9s 11ms/step - loss: 0.9078 - val_loss:
0.8839
```

Out[ ]: <keras.src.callbacks.History at 0x7c8174f28a00>

Zgodnie z przewidywaniami, mimo zmian w sieci i próbie zwiększenia dokładności modelu słabo skorelowane dane uniemożliwiają dalszą poprawę wyniku

In [ ]:
```
predicted_shares_train = model.predict(data_train)
plt.scatter(shares_train, predicted_shares_train, s=5)
plt.xlabel('Train values')
plt.ylabel('Predicted values')
plt.show()

print(
f"Mean squared error:{mean_squared_error(shares_train,predicted_shares_train)}")
```

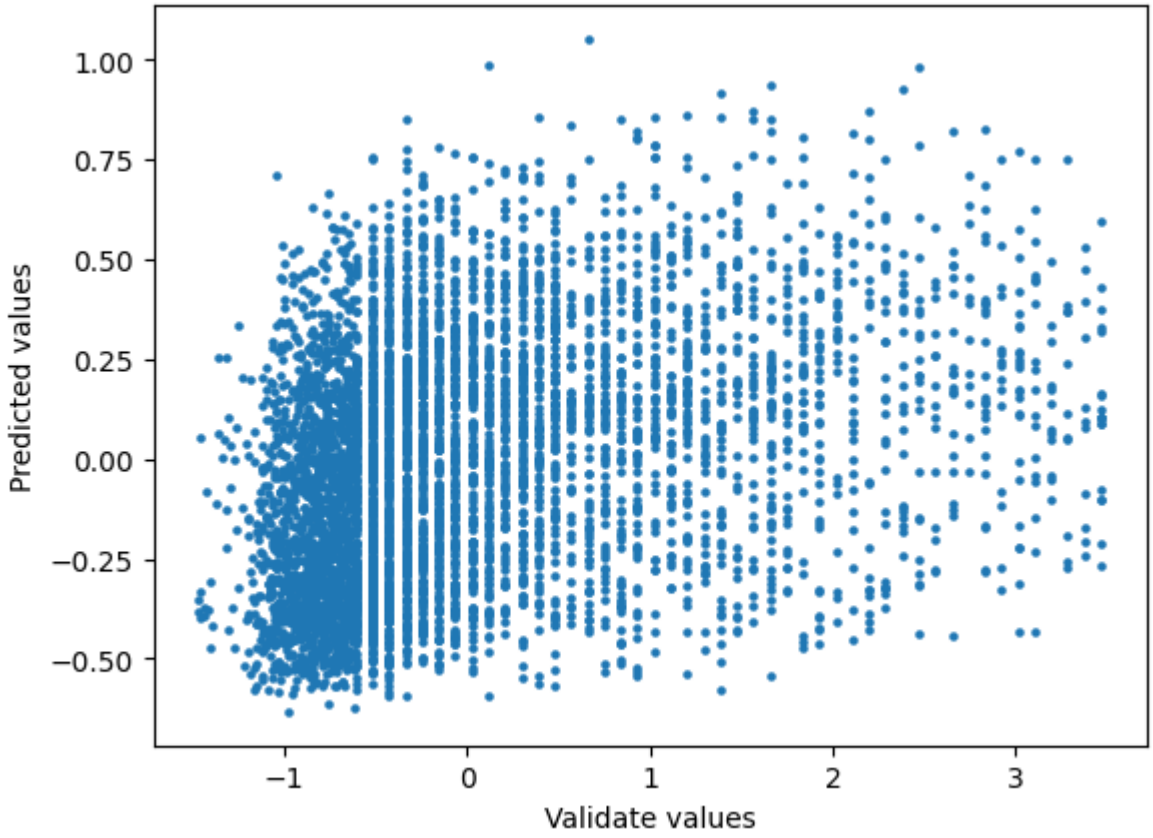
```
768/768 [==============================] - 3s 4ms/step
```




```
Mean squared error: 0.8910823318496843
```

In [ ]:
```
# Przewidywanie danych
predicted_shares = model.predict(data_validate)

# Tworzenie wykresu
plt.scatter(shares_validate, predicted_shares, s=5)
plt.xlabel('Validate values')
plt.ylabel('Predicted values')
plt.show()

print(
f"Mean squared error:{mean_squared_error(shares_validate,predicted_shares)}")
```

```
165/165 [==============================] - 0s 1ms/step
```




```
Mean squared error: 0.8775823240140392
```

Wnioski

Niestety pomimo prób dostrojenia hiperparamatrów naszej sieci, nie udało się osiągnąć satysfakcjonujących wyników. Nawet na wykresach porównujących wartości przewidziane z wartości treningowych, wyniki są błędne. Oznacza to, że nasza sieć nie jest w stanie się niczego nauczyć. Jak pisaliśmy wyżej, sądzimy że jest to spowodowane znikomą liczbą jakichkolwiek korelacji z wartością udostępnień, którą chcemy przewidzieć.