Temat projektu:

Wielowymiarowe prognozowanie szeregów czasowych za pomocą sieci transformers

Autorzy:

Michał Orlewski, Przemysław Rewiś, Łukasz Wajda

1. Temat i założenia projektu

Celem projektu było opracowanie sieci typu transformers do modelowania i predykcji wielowymiarowych szeregów czasowych w oparciu o zestaw danych COVID-19 w Polsce.

Projekt zrealizowano w języku Python z wykorzystaniem biblioteki Keras (zdefiniowanie oraz trening sieci), a także numpy, pandas, matplotlib oraz sklearn. Do ewaluacji modelu zastosowano różne podziały danych na trenujące i testowe oraz różne metryki.

2. Przygotowanie danych

Po wstępnym przeanalizowaniu zestawu danych stwierdzono, że w projekcie wykorzystane zostaną dane:

- nowe przypadki,
- suma wykonanych testów,
- nowe zgony,
- nowe wyzdrowienia,
- aktywne przypadki,
- osoby hospitalizowane,
- zajęte respiratory,
- osoby objęte kwarantanną.

Powyższe dane zostały zapisane do pliku data.csv dla każdego z dni w okresie 03.03.2020 - 10.03.2022. Kolejnym krokiem było wczytanie danych oraz narysowanie wykresów dla wszystkich parametrów za pomocą funkcji:

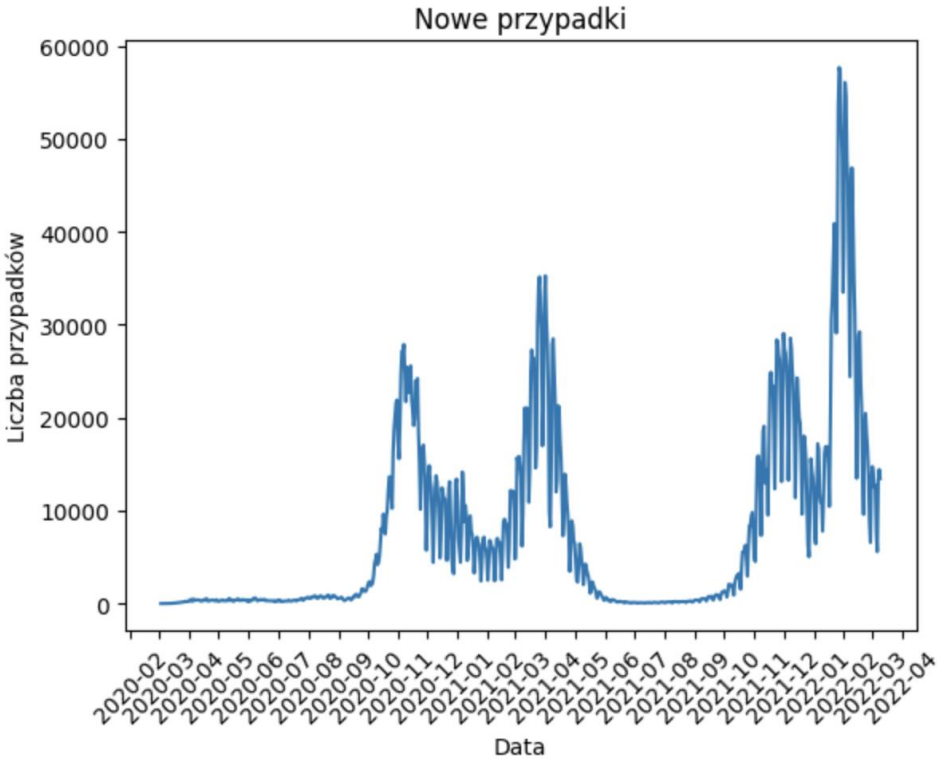
```
def draw_plot(x,y,title,x_label,y_label):
  fig, ax = plt.subplots()
  ax.plot(x,y)
  ax.set_xlabel(x_label)
  ax.set_ylabel(y_label)
  ax.set_title(title)
  ax.xaxis.set_major_locator(mdates.MonthLocator())
  ax.xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m'))
  plt.xticks(rotation=45)
  plt.show()
```

Funkcja rysująca wykres

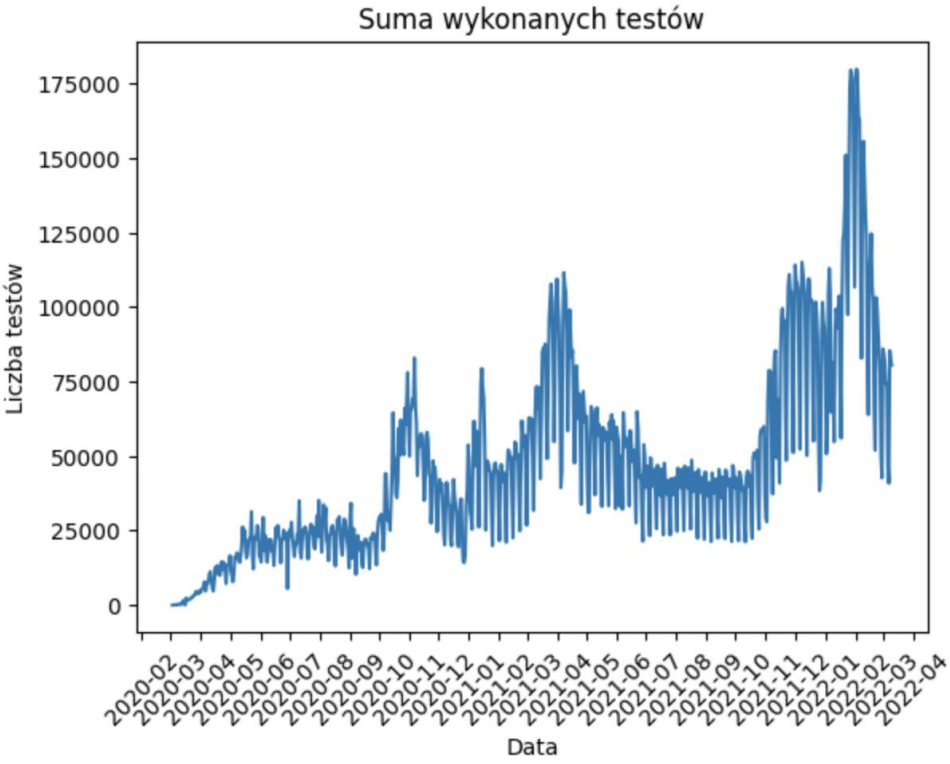
Po ich przeanalizowaniu uznano, że do modelowania i predykcji zostaną wykorzystane ostatecznie tylko poniższe dane:

- nowe przypadki,
- suma wykonanych testów,
- nowe wyzdrowienia,
- aktywne przypadki.

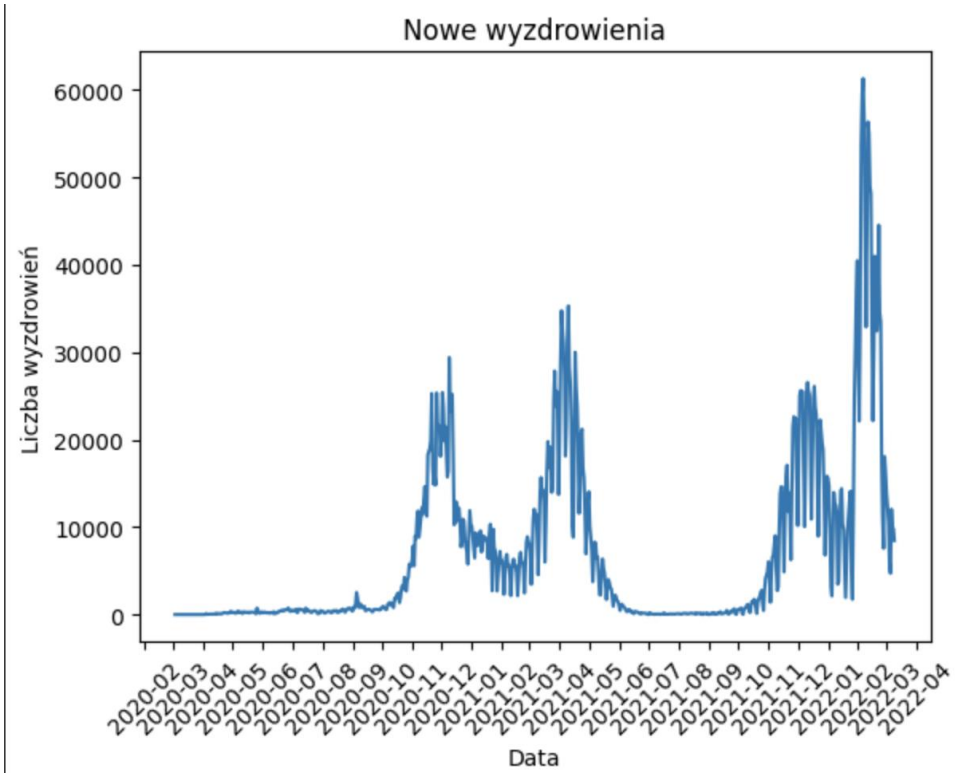
Ich wykresy wyglądają następująco:



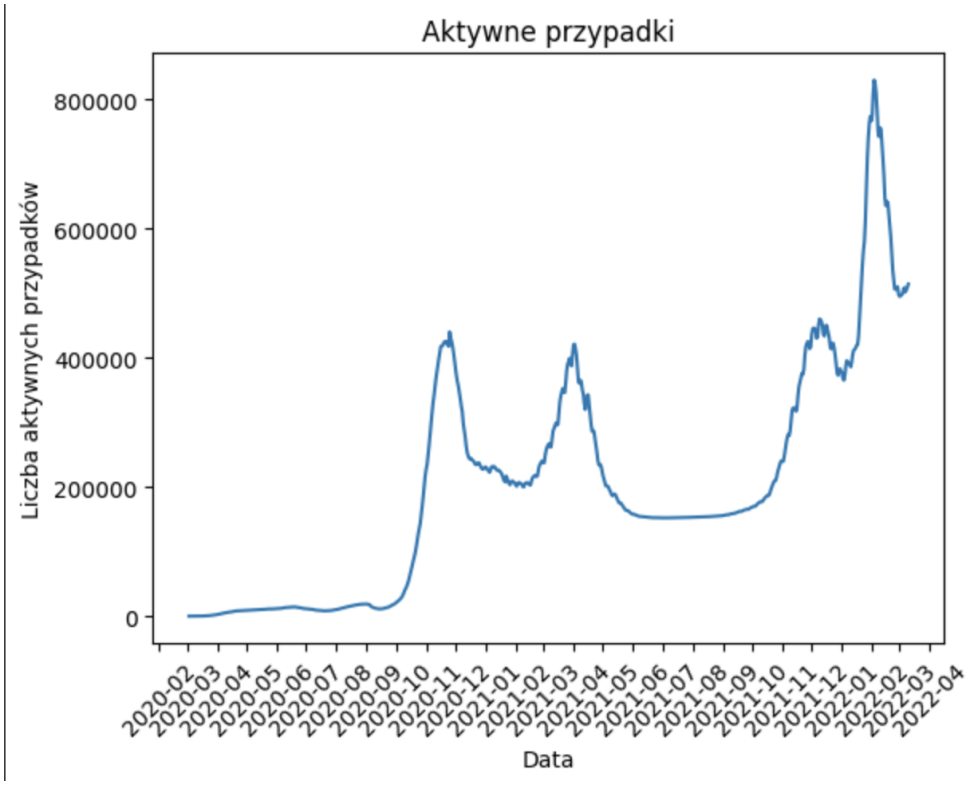
Wykres nowych przypadków



Wykres sumy wykonanych testów

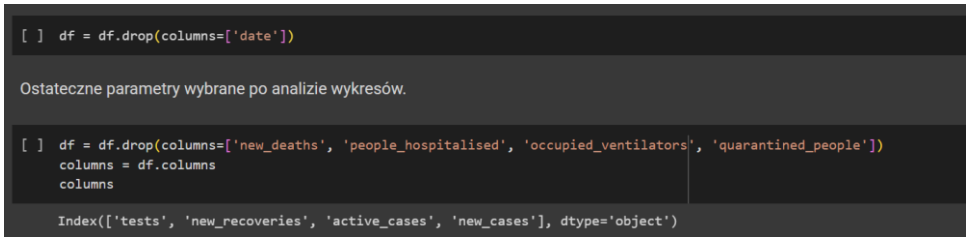


Wykres nowych wyzdrowień



Wykres aktywnych przypadków

W pierwszej części preprocessing danych usunięta zostaje kolumna ‘date’ or pozostałe nieużywane kolumny.



Preprocessing danych cz. 1

Następnie w kolumnie ‘tests’, zawierającej informacje o sumie wykonanych testów wartości zerowe zostały zastąpione wartościami z poprzedniego wiersza (metoda backfill) w tej samej kolumnie. Zamiana wartości zerowych na wartości z poprzedniego wiersza może być stosowana w celu uzupełnienia brakujących danych na podstawie dostępnych informacji. Taka zamiana może być przydatna, gdy istnieje tendencja do powtarzających się wartości w danej kolumnie i chcemy skorzystać z dostępnych informacji do uzupełnienia brakujących wartości. Następnie dane zostały przeskalowane za pomocą Min-Max Scalera. Min-Max Scaler przekształca wartości cech w zakresie od 0 do 1. Jego działanie polega na odejmowaniu najmniejszej wartości cechy i dzieleniu przez różnicę pomiędzy największą i najmniejszą wartością cechy. Przeprowadzenie skalowania cech jest szczególnie przydatne, gdy wartości w różnych kolumnach mają różne zakresy. Skalowanie umożliwia doprowadzenie wartości do wspólnego zakresu, co może poprawić działanie modeli uczenia maszynowego oraz porównywalność i interpretację danych.

Preprocessing danych cz. 2

Przed treningiem modelu konieczne było odpowiednie przygotowanie danych w tablicach X_train, X_val, X_test, y_train, y_val oraz y_test. Przygotowanie to polegało na utworzeniu par X, y, gdzie X składał się z danych dla liczby kolejnych dni określonej parametrem *seq_len*, a y zawierał dane z kolejnego dnia (który był przewidywaną zmienną docelową).

```
# Arrange train data into X_train and y_train
X_train, y_train = [], []
for i in range(seq_len, len(train_data)):
  X_train.append(train_data[i-seq_len:i])
  y_train.append(train_data[:, num_features-1][i])
X_train, y_train = np.array(X_train), np.array(y_train)

# Arrange validation data into X_val and y_val
X_val, y_val = [], []
for i in range(seq_len, len(val_data)):
  X_val.append(val_data[i-seq_len:i])
  y_val.append(val_data[:, num_features-1][i])
X_val, y_val = np.array(X_val), np.array(y_val)

# Arrange test data into X_test and y_test
X_test, y_test = [], []
for i in range(seq_len, len(test_data)):
    X_test.append(test_data[i-seq_len:i])
    y_test.append(test_data[:, num_features-1][i])
X_test, y_test = np.array(X_test), np.array(y_test)
```

Przygotowanie danych treningowych, walidacyjnych i testowych

3. Opis architektury i działania modelu

Do zdefiniowania modelu stworzono 4 klasy pomocnicze: Time2Vector, SingleAttention, MultiAttention oraz TransformerEncoder.

Time2Vector[1] to warstwa embeddingowa mająca na celu zakodowanie informacji o czasie w sekwencyjnych danych, poprzez reprezentację czasu w postaci wektora. Autorzy pracy w której zaproponowano metodą Time2Vector określili dwa warunki jakim powinna spełniać reprezentacja czasu: powinna uwzględniać wzorce zarówno okresowe (np. zmienność pogody w zależności od pory roku) jak i nie okresowe (np. wzrost prawdopodobieństwa zachorowania wraz z wiekiem) oraz powinna być niewrażliwa na skalowanie czasu.

Autorzy pracy zaproponowali poniższą funkcję:

$$\mathbf{t2v}(\tau)[i] = \begin{cases} \omega_i \tau + \varphi_i, & \text{if } i = 0. \\ \mathcal{F}(\omega_i \tau + \varphi_i), & \text{if } 1 \leq i \leq k. \end{cases}$$

Matematyczna reprezentacja Time2Vector

$\omega_i\tau + \varphi_i$ oznacza nie okresową lub liniową część wektora czasu (jest to zwykła prosta)

$F(\omega_i\tau + \varphi_i)$ oznacza okresową część wektora czasu (za funkcję reprezentującą tą okresowość najczęściej przyjmuje się funkcję sinusoidalną.

Poniżej pokazano implementację warstwy Time2Vector w kerasie:

```
class Time2Vector(Layer):
  '''Embedding Layer representing time as a vector'''
  def __init__(self, seq_len, **kwargs):
    super(Time2Vector, self).__init__()
    self.seq_len = seq_len # length of a time sequence in each input

  def build(self, input_shape):
    '''Initializes weights for periodic and non-periodic features'''
    self.weights_linear = self.add_weight(name='weight_linear',
                                shape=(int(self.seq_len),),
                                initializer='uniform',
                                trainable=True)

    self.bias_linear = self.add_weight(name='bias_linear',
                                shape=(int(self.seq_len),),
                                initializer='uniform',
                                trainable=True)

    self.weights_periodic = self.add_weight(name='weight_periodic',
                                shape=(int(self.seq_len),),
                                initializer='uniform',
                                trainable=True)

    self.bias_periodic = self.add_weight(name='bias_periodic',
                                shape=(int(self.seq_len),),
                                initializer='uniform',
                                trainable=True)

  def call(self, x):
    '''Calculates periodic and non-periodic features'''
    x = tf.math.reduce_mean(x[:,:,:4], axis=-1)
    time_linear = self.weights_linear * x + self.bias_linear # non-periodic feature
    time_linear = tf.expand_dims(time_linear, axis=-1) # Add dimension

    time_periodic = tf.math.sin(tf.multiply(x, self.weights_periodic) + self.bias_periodic) # periodic feature
    time_periodic = tf.expand_dims(time_periodic, axis=-1) # Add dimension
    return tf.concat([time_linear, time_periodic], axis=-1)

  def get_config(self):
    config = super().get_config().copy()
    config.update({'seq_len': self.seq_len})
    return config
```

Warstwa Time2Vector

Funkcja *build()* inicjalizuje macierze wag dla cech okresowych i nie okresowych.

Funkcja *call()* wylicza macierze cech okresowych (*time_periodic*) oraz nie okresowych (*time_linear*) i zwraca je. Należy zwrócić uwagę że w przypadku cech okresowych, zastosowano funkcję sinusoidalną.

Sieci typu transformers korzystają z mechanizmu self-attention, pozwalającego modelowi skupić się na istotnych częściach danych aby poprawić jakość ich działania. W tym celu potrzebne były warstwy *SingleAttention* oraz *MultiAttention.* Mechanizm self-attention pozwala na połączenie ze sobą wszystkich kroków szeregu czasowego i wykrycie zależności długoterminowych. Dodatkowo, wszystkie te procesy są zrównoleglone, co umożliwia szybszy czas uczenia sieci.

Warstwa SingleAttention przyjmuje 3 wejścia: query, key oraz value. Każde z tych wejść przechodzi przez własne warstwy gęste. Następnie w funkcji *call()* wyliczany jest wyjściowa wartość attention.

```python
class SingleAttention(Layer):
  '''Single attention layer'''
  def __init__(self, d_k, d_v):
    super(SingleAttention, self).__init__()
    self.d_k = d_k
    self.d_v = d_v

  def build(self, input_shape):
    ''' Initialized dense networks for each of inputs'''
    self.query = Dense(self.d_k,
                       input_shape=input_shape,
                       kernel_initializer='glorot_uniform',
                       bias_initializer='glorot_uniform')

    self.key = Dense(self.d_k,
                     input_shape=input_shape,
                     kernel_initializer='glorot_uniform',
                     bias_initializer='glorot_uniform')

    self.value = Dense(self.d_v,
                       input_shape=input_shape,
                       kernel_initializer='glorot_uniform',
                       bias_initializer='glorot_uniform')

  def call(self, inputs):
    ''' Calculated attention output '''
    q = self.query(inputs[0]) # represents positions of interest in the sequence
    k = self.key(inputs[1]) # represents information about different positions

    attn_weights = tf.matmul(q, k, transpose_b=True) # dot product measures similarity between query and key
    attn_weights = tf.map_fn(lambda x: x/np.sqrt(self.d_k), attn_weights) # scaling the dot product (to stabilize gradients)
    attn_weights = tf.nn.softmax(attn_weights, axis=-1) # normalize attention weights to 1 using softmax

    v = self.value(inputs[2]) # represents features
    attn_out = tf.matmul(attn_weights, v) # emphasizes the value vectors that are most relevant to each query position
    return attn_out # return relevant information form the value vector
```

Warstwa SingleAttention

Warstwa *MultiAttention* ma na celu połączenie wyników z określonej liczby warstw *SingleAttention* oraz poddaniu tego wyniku nieliniowej transformacji (przy użyciu warstwy gęstej). Tak więc funkcja *build()* tworzy listę warstw *SingleAttention,* a funkcja *call()* wylicza wartości wyjściowe każdej z nich, scala wyniki, wrzuca je do sieci gęstej i zwraca wynik.

```python
class MultiAttention(Layer):
  '''Multi attention layer'''
  def __init__(self, d_k, d_v, n_heads):
    super(MultiAttention, self).__init__()
    self.d_k = d_k
    self.d_v = d_v
    self.n_heads = n_heads
    self.attn_heads = list()

  def build(self, input_shape):
    ''' Creates n_heads SingleAttention layers and a Dense layer'''
    for n in range(self.n_heads):
      self.attn_heads.append(SingleAttention(self.d_k, self.d_v))

    self.linear = Dense(input_shape[0][-1],
                        input_shape=input_shape,
                        kernel_initializer='glorot_uniform',
                        bias_initializer='glorot_uniform')

  def call(self, inputs):
    ''' Concatenates and lineary transforms output from each SingleAttention layer'''
    attn = [self.attn_heads[i](inputs) for i in range(self.n_heads)]
    concat_attn = tf.concat(attn, axis=-1)
    multi_linear = self.linear(concat_attn)
    return multi_linear
```

Warstwa MultiAttention

Warstwa *TransformerEncoder* opakowuje warstwy *SingleAttention* oraz *MultiAttention.* Każda warstwa składa się z warstwy *MultiAttention, Dropout, LayerNormalization,* a także z sieci jednokierunkowej (w tym przypadku warstw *Conv1D).*

```python
class TransformerEncoder(Layer):
  def __init__(self, d_k, d_v, n_heads, ff_dim, dropout=0.1, **kwargs):
    super(TransformerEncoder, self).__init__()
    self.d_k = d_k
    self.d_v = d_v
    self.n_heads = n_heads
    self.ff_dim = ff_dim
    self.attn_heads = list()
    self.dropout_rate = dropout

  def build(self, input_shape):
    # self-attention part
    self.attn_multi = MultiAttention(self.d_k, self.d_v, self.n_heads)
    self.attn_dropout = Dropout(self.dropout_rate)
    self.attn_normalize = LayerNormalization(input_shape=input_shape, epsilon=1e-6)

    # feed forward part
    self.ff_conv1D_1 = Conv1D(filters=self.ff_dim, kernel_size=1, activation='relu')
    self.ff_conv1D_2 = Conv1D(filters=input_shape[0][-1], kernel_size=1)
    self.ff_dropout = Dropout(self.dropout_rate)
    self.ff_normalize = LayerNormalization(input_shape=input_shape, epsilon=1e-6)

  def call(self, inputs):
    attn_layer = self.attn_multi(inputs)
    attn_layer = self.attn_dropout(attn_layer)
    attn_layer = self.attn_normalize(inputs[0] + attn_layer)

    ff_layer = self.ff_conv1D_1(attn_layer)
    ff_layer = self.ff_conv1D_2(ff_layer)
    ff_layer = self.ff_dropout(ff_layer)
    ff_layer = self.ff_normalize(inputs[0] + ff_layer)
    return ff_layer

  def get_config(self):
    config = super().get_config().copy()
    config.update({'d_k': self.d_k,
                   'd_v': self.d_v,
                   'n_heads': self.n_heads,
                   'ff_dim': self.ff_dim,
                   'attn_heads': self.attn_heads,
                   'dropout_rate': self.dropout_rate})
    return config
```

Warstwa TransformerEncoder

Model tworzony jest przy pomocy funkcji *create_model(),* i składa się z warstwy embedding'owej *Time2Vector* oraz z 3 warstw *TransformerEncoder (*a także dodatkowych warstw *Dropout* oraz *Dense).*

```python
def create_model():
    '''Initialize time and transformer layers'''
    time_embedding = Time2Vector(seq_len)
    attn_layer1 = TransformerEncoder(d_k, d_v, n_heads, ff_dim)
    attn_layer2 = TransformerEncoder(d_k, d_v, n_heads, ff_dim)
    attn_layer3 = TransformerEncoder(d_k, d_v, n_heads, ff_dim)

    '''Construct model'''
    in_seq = Input(shape=(seq_len, num_features))
    x = time_embedding(in_seq)
    x = Concatenate(axis=-1)([in_seq, x])
    x = attn_layer1((x, x, x))
    x = attn_layer2((x, x, x))
    x = attn_layer3((x, x, x))
    x = GlobalAveragePooling1D(data_format='channels_first')(x)
    x = Dropout(0.1)(x)
    x = Dense(64, activation='relu')(x)
    x = Dropout(0.1)(x)
    out = Dense(1, activation='linear')(x)

    model = Model(inputs=in_seq, outputs=out)
    model.compile(loss='mse', optimizer='adam', metrics=['mae', 'mape'])
    return model
```

Funkcja tworząca model

Ostatecznie w modelu zastosowano następujące parametry:

```
[ ]  d_k = 256
     d_v = 256
     n_heads = 12
     ff_dim = 256

     seq_len = 28
     num_features = 4
```

4. Sposoby podziału danych

Do testowania użyto kilku podejść podziału danych na uczące oraz testujące. Podział danych w modelowaniu i prognozowaniu szeregów czasowych jest trudny ze względu na niestacjonarność danych i zależności między poprzednimi a kolejnymi obserwacjami.

W naszym projekcie wykorzystaliśmy trzy sposoby podział danych:

- K-Fold,
- TimeSeriesSplit,
- BlockingTimeSeriesSplit.

Użyte przez nas metody wykorzystują różne sposoby podziału danych na zbiór uczący i testowy. Jest to ważne, aby zapewnić realistyczną ocenę modelu. Jednak klasyczna walidacja krzyżowa nie jest odpowiednia dla szeregów czasowych.

4.1 Użyte metryki

Odpowiedni wybór metryk do oceny modelu ma bardzo duży wpływ na wydajność i zbieżność modelu. W dla naszego modelu użyliśmy funkcję straty najczęściej używaną w zadaniach regresji, w ocenie jakości modeli prognozujących, są to:

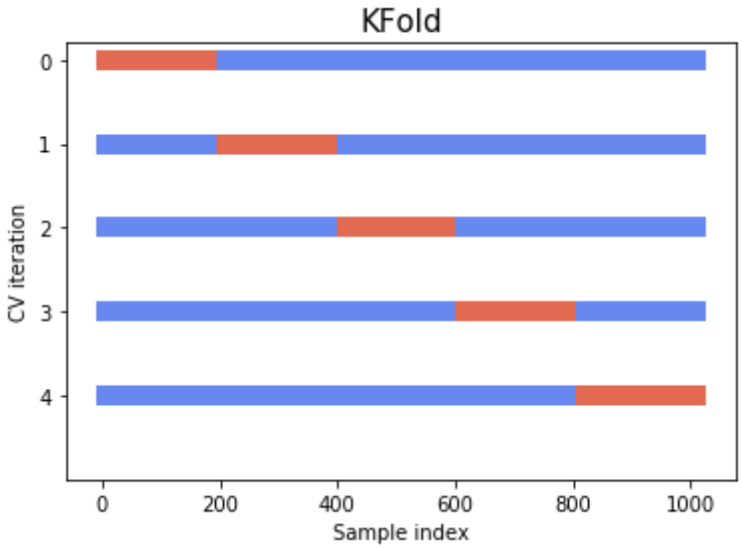
- Błąd średniokwadratowy (MSE): Mierzy średnią kwadratową różnicę między przewidywanymi a rzeczywistymi wartościami. Pomaga wykryć większe błędy predykcji.
- Średni błąd absolutny (MAE): Mierzy średnią absolutną różnicę między przewidywanymi a rzeczywistymi wartościami. Jest bardziej odporny na wartości odstające niż MSE.
- Współczynnik determinacji (R^2): Mierzy, jak dobrze model dostosowuje się do danych i wyjaśnia zmienność. Wartość 1 oznacza idealne dopasowanie modelu, a wartość 0 oznacza brak zgodności.
- Współczynnik zgodności (index of agreement): Ocenia stopień zgodności między przewidywanymi a rzeczywistymi wartościami. Wartość 1 oznacza doskonałe dopasowanie, a wartość 0 oznacza brak zgodności. Współczynnik zgodności (index of agreement) pomiędzy prawdziwymi wartościami (y_true) a przewidywanymi wartościami (y_pred) opiera się na wzorze na sumę kwadratów różnic pomiędzy tymi wartościami.

- Średni absolutny błąd procentowy (MAPE): Mierzy średnią procentową różnicę między przewidywanymi a rzeczywistymi wartościami, uwzględniając wartość rzeczywistą.

4.2 Metoda k-fold

Metoda K-Fold Cross Validation (walidacja krzyżowa K-krotna) polega na podziale danych na K równych części, zwanych foldami. Następnie model jest trenowany K razy, każdorazowo używając jednego z foldów jako zbioru testowego, a pozostałe foldy jako zbiory uczące. Ostateczne wyniki są uśredniane, aby uzyskać ocenę wydajności modelu. Ta metoda jest szeroko stosowana, gdy mamy wystarczającą ilość danych do podziału na zbiory uczące i testujące.

Dla lepszego zrozumienia metod przedstawiono wykresy. Oś pozioma reprezentuje rozmiar zbioru treningowego, a oś pionowa oznacza iteracje cross-validation. Foldy używane do treningu są oznaczone na niebiesko, a foldy używane do testowania są oznaczone na pomarańczowo. Możemy intuicyjnie interpretować oś poziomą jako linię postępu czasowego, ponieważ nie mieszaliśmy danych i zachowaliśmy ich chronologiczny porządek.



Wykres podziału danych dla kolejnych przebiegów podziału metodą K-fold

W projekcie najpiew przeprowadzamy ocenę modelu przy użyciu metody KFold. Pętla iteruje przez kolejne podziały danych na część treningową i testową. Wewnątrz tej pętli:

- Dane są podzielone na zbiory treningowe i testowe.
- Dane treningowe są przekształcane do postaci X_train (wejście) i y_train (wartość docelowa).
- Dane testowe są przekształcane do postaci X_test (wejście) i y_test (wartość docelowa).
- Model jest tworzony i trenowany na danych treningowych.

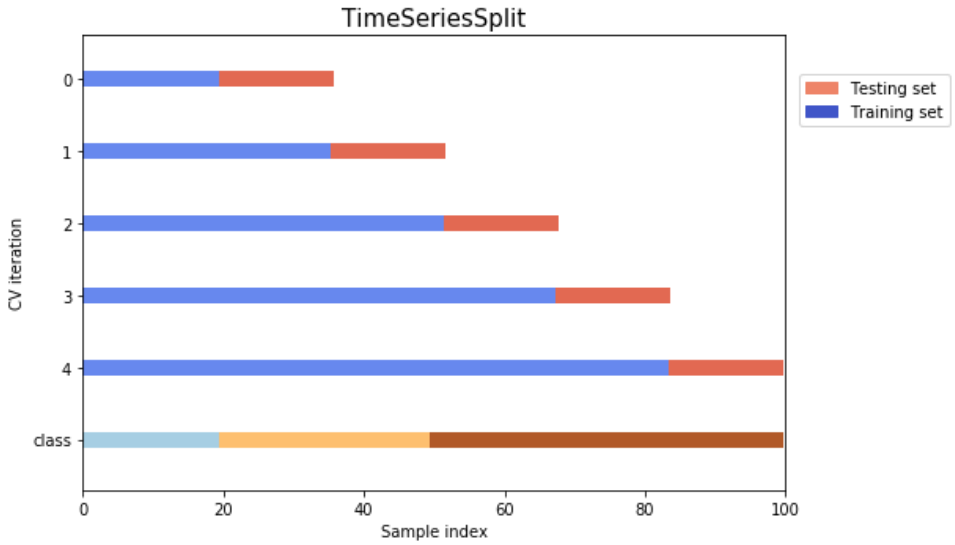
- Model jest oceniany na danych testowych za pomocą funkcji evaluate_model. Funkcja evaluate_model używa zdefiniowanych funkcji do obliczenia różnych miar oceny modelu: MSE, MAE, R^2, index of agreement i MAPE. Zwraca ostatecznie wartości metryk dla pojedynczego treningu jako wynik.
- Na koniec wyniki metryk są uśredniane

```
mse_scores = []
mae_scores = []
r2_scores = []
index_of_agreement_scores = []
mape_scores = []

kfold = KFold(n_splits=n_splits)

for train_index, test_index in kfold.split(df):

    # Split data into train and test
    train_data = df.iloc[train_index].values
    test_data = df.iloc[test_index].values

    # Arrange train data into X_train and y_train
    X_train, y_train = [], []
    for i in range(seq_len, len(train_data)):
        X_train.append(train_data[i-seq_len:i])
        y_train.append(train_data[:, num_features-1][i])
    X_train, y_train = np.array(X_train), np.array(y_train)

    # Arrange test data into X_test and y_test
    X_test, y_test = [], []
    for i in range(seq_len, len(test_data)):
        X_test.append(test_data[i-seq_len:i])
        y_test.append(test_data[:, num_features-1][i])
    X_test, y_test = np.array(X_test), np.array(y_test)

    # Build and train model
    model = create_model()
    model.fit(X_train, y_train, batch_size=batch_size, epochs=num_epochs, verbose=0)

    # Evaluate model
    y_pred = model.predict(X_test)
    mse, mae, r2, index_of_agreement, mape = evaluate_model(y_test, y_pred)
    mse_scores.append(mse)
    mae_scores.append(mae)
    r2_scores.append(r2)
    index_of_agreement_scores.append(index_of_agreement)
    mape_scores.append(mape)

    print(f"MSE: {mse}, MAE: {mae}, R^2: {r2}, Index of Agreement: {index_of_agreement}, Mean Absolute Percentage Error: {mape} ")

# Print evaluation results for the current split method
print("Split Method: KFold")
print_evaluation_results_extended(mse_scores, mae_scores, r2_scores, index_of_agreement_scores, mape_scores)
```

Ocena modelu metodą podziału KFold

4.3 Metoda TimeSeriesSplit

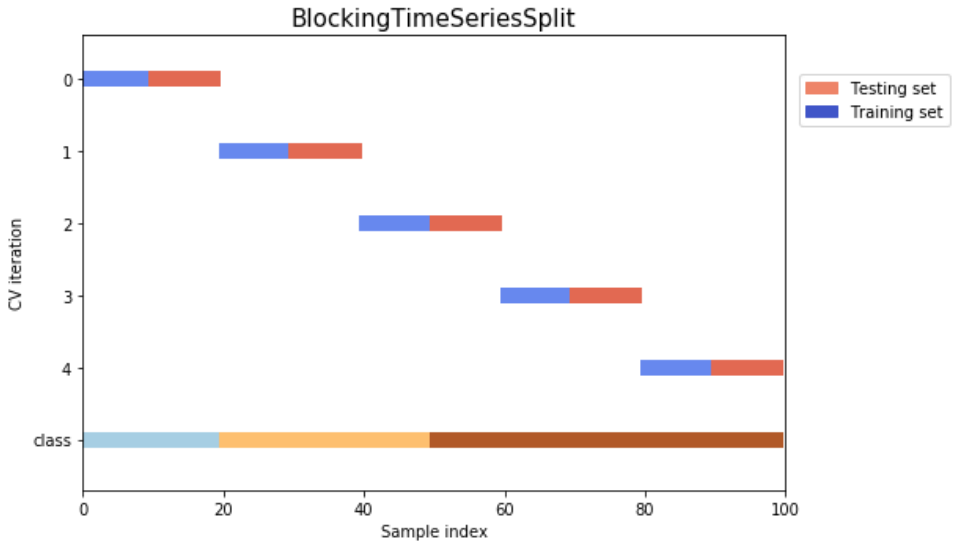
Time Series Split (podział szeregów czasowych) jest szczególnie przydatny w analizie danych sekwencyjnych, gdzie kolejność danych ma znaczenie. W tej metodzie dane są dzielone na sekwencje czasowe w kolejności chronologicznej. Najnowsze obserwacje są wykorzystywane jako zbiór testowy, a pozostałe dane są używane jako zbiór uczący. Takie podejście pozwala na ocenę wydajności modelu na danych przyszłych, które są bardziej realistyczne. Analogicznie jak dla metody K-Fold wykonaliśmy podobny zestaw operacji oraz zebraliśmy te same metryki.



Wykres podziału danych dla kolejnych przebiegów podziału metodą Time Series Split

4.4 Metoda BlockingTimeSeriesSplit

Ostatnią zastosowaną przez nas metodą podziału danych był Blocking Time Series Split. Jest to metoda podobna do Time Series Split, ale uwzględniająca blokowanie danych. Czasami w danych szeregach czasowych można zaobserwować bloki, w których wartości mają podobne wzorce lub właściwości. W takim przypadku, aby uniknąć wpływu bloków na podział danych, można użyć BlockingTimeSeriesSplit. Polega on na podziale danych na bloki i zachowaniu ich integralności podczas podziału na zbiory uczące i testujące. Dzięki temu można zabezpieczyć, że cały blok znajduje się albo w zbiorze uczącym, albo w zbiorze testowym.



Wykres podziału danych dla kolejnych przebiegów podziału metodą Blocking Time Series Split

W naszych danych nie zaobserwowaliśmy bloków, w których wartości mają podobne wzorce lub właściwości, więc ta metoda ta nie okazał się dobrym wyborem. Z uwaga na to, że funkcja jest mało popularna została usunięta z biblioteki sklearn i wymagała własnej implementacji.

```python
class BlockingTimeSeriesSplit():
    def __init__(self, n_splits, margin=0):
        self.n_splits = n_splits
        self.margin = margin

    def get_n_splits(self, X, y, groups):
        return self.n_splits

    def split(self, X, y=None, groups=None):
        n_samples = len(X)
        k_fold_size = n_samples // self.n_splits
        indices = np.arange(n_samples)

        for i in range(self.n_splits):
            start = i * k_fold_size
            stop = start + k_fold_size
            mid = int(0.8 * (stop - start)) + start
            yield indices[start: mid], indices[mid + self.margin: stop]
```

Własna implementacja BlockingTimeSeriesSplit

5. Ocena modelu

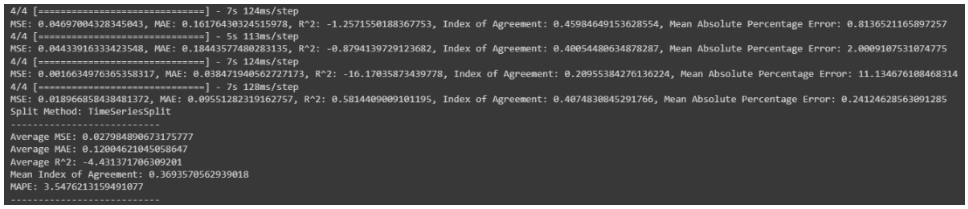
Przy różnych próbach treningowych różnych i różnych ustawień parametru seq_len = 14, 28, 56, 112, stwierdzono ostatecznie, że przy wartości 28, model osiąga najlepsze wyniki. Wówczas metryki dla różnych podziałów danych prezentują się w następujący sposób:

Dla metody K-fold:

```
5/5 [==============================] - 8s 203ms/step
MSE: 0.0005964504686736187, MAE: 0.016564312498601227, R^2: -69.46495503403904, Index of Agreement: 0.11286226944059186, Mean Absolute Percentage Error: 2.727848211681467
5/5 [==============================] - 6s 127ms/step
MSE: 0.003172726187862796, MAE: 0.04607952183962474, R^2: 0.7442336691141762, Index of Agreement: 0.41124654057230836, Mean Absolute Percentage Error: 0.3014231442432841
5/5 [==============================] - 6s 121ms/step
MSE: 0.0015546422468116004, MAE: 0.02833987450774349, R^2: 0.83836972197923, Index of Agreement: 0.3009702192164977, Mean Absolute Percentage Error: 5.837940080374225
5/5 [==============================] - 10s 128ms/step
MSE: 0.019071091204781652, MAE: 0.09152796838890084, R^2: 0.612168034352762, Index of Agreement: 0.3891838998174971, Mean Absolute Percentage Error: 0.31572831112221805
Split Method: KFold
--------------------------
Average MSE: 0.006098727527032417
Average MAE: 0.045627919308717575
Average R^2: -16.817545902148215
Mean Index of Agreement: 0.3035657322617238
MAPE: 2.2957349368552986
--------------------------
```

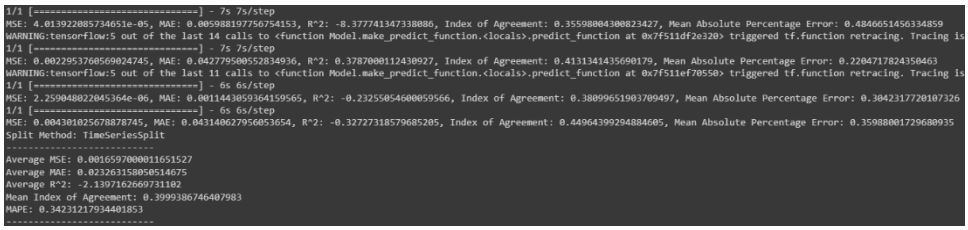
- MSE jest w granicach normy.
- MAE również mieści się w zakresie normy.
- R^2 (współczynnik determinacji) ma ujemne wartości, co wskazuje na to, że model nie jest w pełni adekwatny do danych. Należy zwrócić uwagę na możliwość poprawy modelu.
- Index of Agreement (współczynnik zgodności) i MAPE (średni błąd procentowy) mają różne wartości dla różnych podziałów, ale ich wartości są w granicach normy.

Dla metody TimeSplitSeries:



- MSE ma nieco wyższe wartości.
- MAE ma również dość wysoką wartość.
- R^2 ma ujemne wartości, co może wskazywać na niedostateczne dopasowanie modelu do danych.
- Index of Agreement i MAPE mają różne wartości dla różnych podziałów, ale ich wartości są w granicach normy.

Dla metody BlockingTimeSplitSeries:



- MSE jest niską wartością i sugeruje dobre dopasowanie modelu.
- MAE ma również niską wartośc.
- R^2 ma ujemne wartości, co może wskazywać na niedostateczne dopasowanie modelu do danych.
- Index of Agreement i MAPE mają różne wartości dla różnych podziałów, ale ich wartości są w granicach normy.

Podsumowując, wyniki metryk dla różnych podziałów danych są w różnym stopniu zgodne z normą. Należy zwrócić szczególną uwagę na wartości R^2, która okazała się ujemna. Może to być spowodowane np. nieliniową relacją między danymi. W takiej sytuacji R^2 nie jest dobrym wyznacznikiem działania modelu i należy się skupić na pozostałych metrykach.

Na tym etapie warto wspomnieć parę wad podziału KFold stosowanego do szeregów czasowych:

1. W niektórych iteracjach, dane testowe występują przed danymi treningowymi
2. W niektórych iteracjach dane treningowe występują po danych testowych, co jest problematyczne ponieważ model jest w stanie podejrzeć co się dzieje w przyszłości
3. W niektórych iteracjach występują luki w seriach treningowych

Z powyższych powodów, przy ocenie modelu skupiliśmy się bardziej na podziałach TimeSeriesSplit i BlockingTimeSeriesSplit, które są przystosowane do szeregów czasowych i nie mają powyższych wad.

6. Trening ostatecznej wersji modelu

Po ocenie modelu z wykorzystaniem różnych podziałów danych, przeszliśmy do przygotowania danych do ostatecznego treningu modelu. Dane podzielono w następujący sposób: dane treningowe (70%), dane walidacyjne (10%), dane testowe (20%).

```
[ ]  #Split df into df_train and df_test

     split_index_1 = sorted(df.index.values)[-int(0.3*len(df))]
     split_index_2 = sorted(df.index.values)[-int(0.2*len(df))]

     df_train = df[(df.index < split_index_1)] # 70% of data
     df_val = df[(df.index >=split_index_1) & (df.index < split_index_2)] # 10% of data
     df_test = df[(df.index >= split_index_2)] # 20% of data

     train_data = df_train.values
     val_data = df_val.values
     test_data = df_test.values

     print('Training data shape: {}'.format(train_data.shape))
     print('Validation data shape: {}'.format(val_data.shape))
     print('Test data shape: {}'.format(test_data.shape))

     Training data shape: (517, 4)
     Validation data shape: (74, 4)
     Test data shape: (147, 4)
```

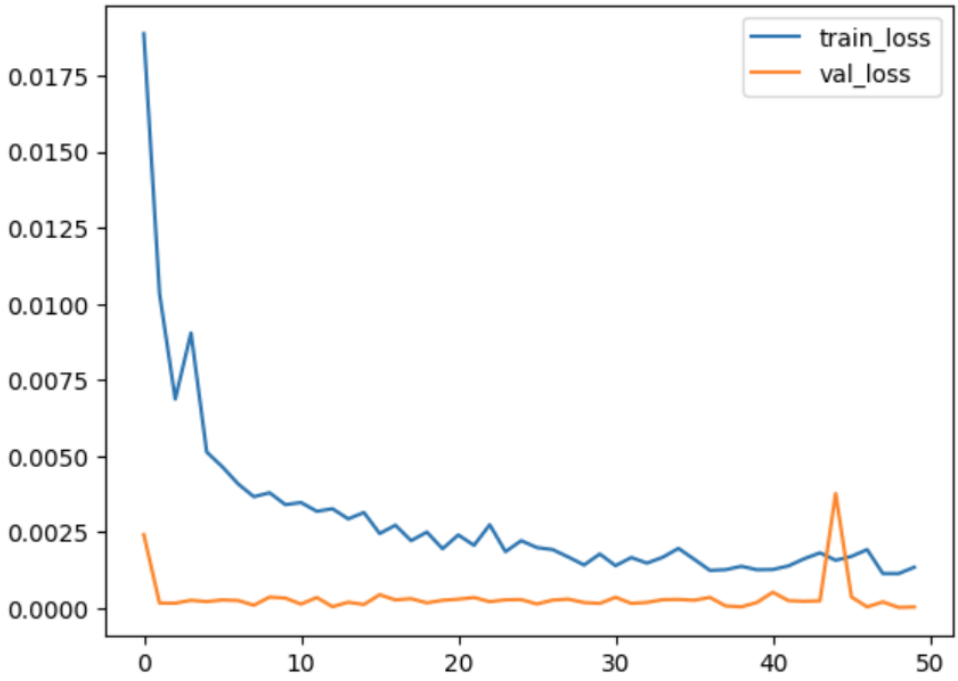
Podział danych przed ostatnim treningiem

Trening modelu trwał 50 epok, a po każdej epoce zapisywano najlepszą wersję modelu w oparciu o wartość funkcji straty na danych walidacyjnych.

```
[ ]  batch_size_final = 16
     num_epochs_final = 50

     callback = tf.keras.callbacks.ModelCheckpoint('model.hdf5',
                                                   monitor='val_loss',
                                                   save_best_only=True,
                                                   verbose=1)

[ ]  history = final_model.fit(X_train, y_train, batch_size=batch_size_final,
                               epochs=num_epochs_final, validation_data=(X_val, y_val),
                               callbacks=[callback], verbose=1)
```

Uruchomienie treningu i zapisywanie modelu

Poniższy wykres przedstawia wartości funkcji straty dla danych treningowych i walidacyjnych podczas treningu.



Wartości funkcji straty podczas treningu

7. Podsumowanie wyników i wnioski

Wyniki metryk dla modelu są następujące:

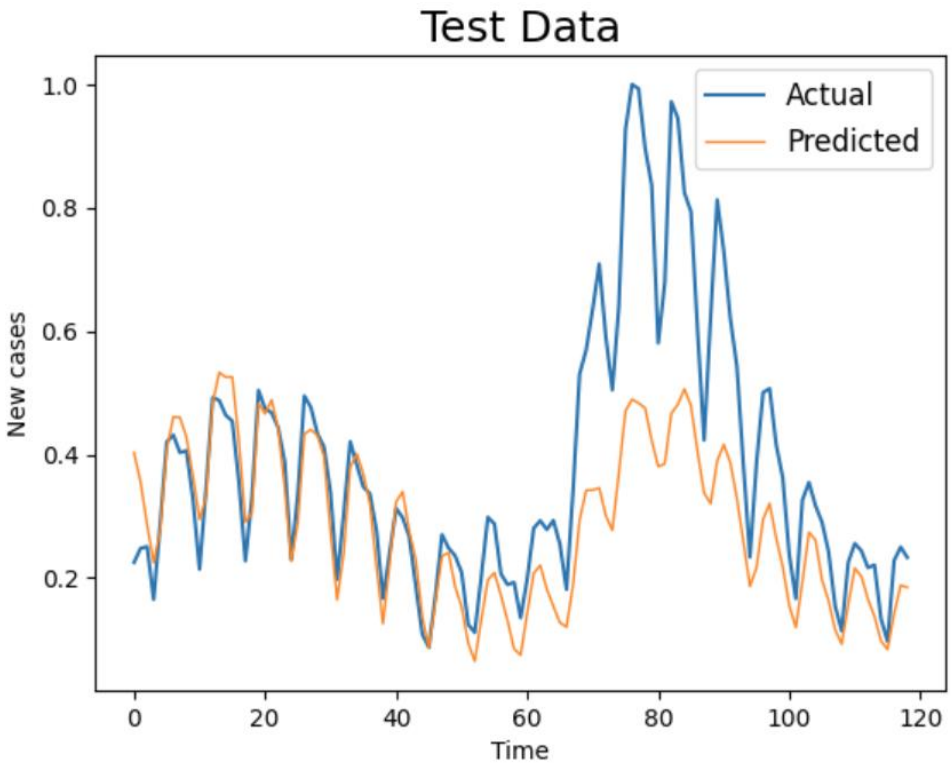
```
16/16 [==============================] - 4s 230ms/step
Final Model Evaluation Results (Train Set):
---------------------------
Average MSE: 0.0008339403432611366
Average MAE: 0.016155907724206567
Average R^2: 0.955644013952674
Mean Index of Agreement: 0.3762976617355004
MAPE: 0.3050161655376442
---------------------------
```

Metryki modelu dla danych trenujących

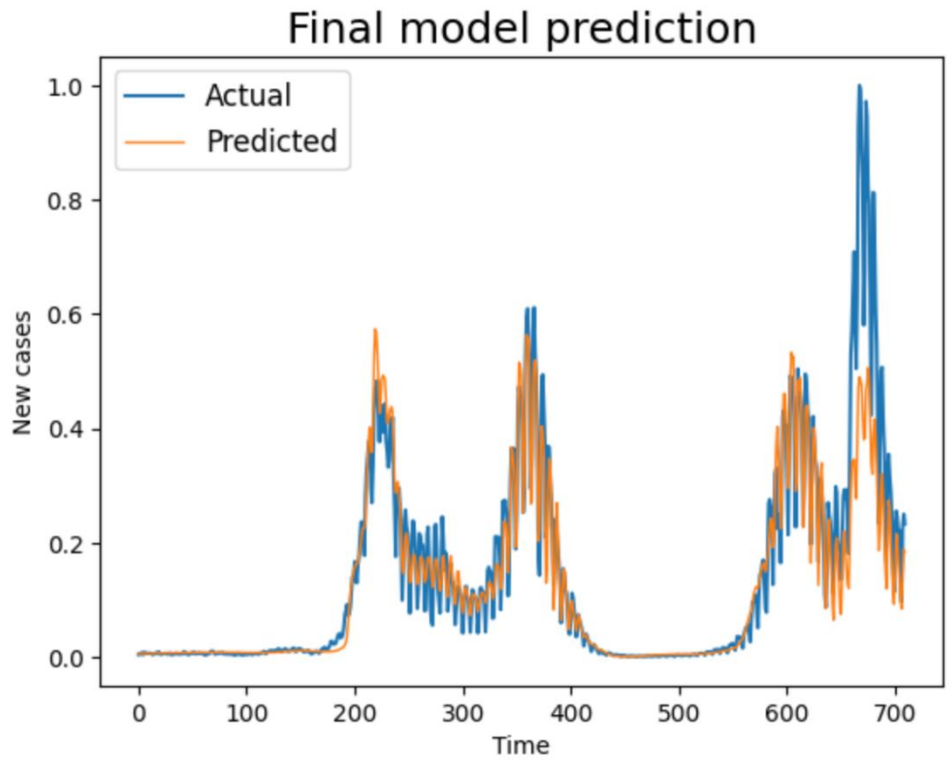
```
4/4 [==============================] - 0s 115ms/step
Final Model Evaluation Results:
---------------------------
Average MSE: 0.02882426452261324
Average MAE: 0.11396714894893353
Average R^2: 0.3639084601361432
Mean Index of Agreement: 0.38569917725240055
MAPE: 0.26393492300640614
---------------------------
```

Metryki modelu dla danych testowych

Poniższe wykresy przedstawiają wyniki prognozowania ilości nowych przypadków.



Wyniki modelu na zbiorze testowym



Wyniki modelu na wszystkich danych

Uzyskane wyniki są dobre. Model dobrze poradził sobie z przewidzeniem trendów zmian nowych zachorowań. Niestety model nie poradził sobie w przypadku, zaistnienia niespodziewanego ogromnego przyrostu zachorowań.

Sieci Transformers zyskały w ostatnich latach na popularności ze względu na ich wyjątkową wydajność. Połączenie mechanizmu samouwagi, równoległości i kodowania pozycyjnego pod jednym dachem zwykle zapewnia przewagę nad klasycznymi modelami LSTM i CNN podczas pracy nad zadaniami, w których wymagana jest semantyczna ekstrakcja cech z dużych zbiorów danych. Sieci typu Transformers składają się z wielu warstw transformatorowych, które operują na sekwencjach danych bez użycia rekurencji. Transformator zawiera mechanizm uwagi i warstwy połączone w pełni, co umożliwia efektywne modelowanie zależności między odległymi elementami sekwencji. Sieci LSTM natomiast są rekurencyjnymi sieciami neuronowymi opartymi na komórkach pamięci, które pozwalają przechowywać informacje z wcześniejszych kroków czasowych. To umożliwia uwzględnienie kontekstu historycznego przy predykcji kolejnych wartości w szeregu czasowym. Jeśli chodzi o uwzględnianie kontekstu, sieci typu Transformers są zdolne do efektywnego uwzględniania globalnych zależności między elementami sekwencji dzięki mechanizmowi uwagi. Mogą brać pod uwagę szeroki kontekst z przeszłości, co jest szczególnie korzystne w przypadku predykcji szeregów czasowych o złożonych wzorcach. Z kolei sieci LSTM są dobrze przystosowane do uwzględniania lokalnych zależności w sekwencjach czasowych dzięki rekurencyjnym połączeniom. Mogą przechowywać i przekazywać informacje między krokami czasowymi, umożliwiając uwzględnienie kontekstu historycznego i sekwencyjnego charakteru danych. Jeśli chodzi o obliczeniową złożoność, sieci typu Transformers mogą być bardziej wymagające obliczeniowo, zwłaszcza przy dużych sekwencjach danych. Mechanizm uwagi wymaga obliczenia wag dla wszystkich par elementów sekwencji, co prowadzi do wyższych wymagań obliczeniowych. Z kolei sieci LSTM są bardziej wydajne obliczeniowo, ponieważ przetwarzają sekwencję danych krok po kroku, co oznacza, że przewidywanie kolejnej wartości w szeregu czasowym wymaga tylko jednej iteracji przez sieć. Podsumowując, sieci typu Transformers są bardziej elastyczne w modelowaniu złożonych wzorców czasowych, takich jak długotrwałe zależności, nieliniowe trendy i interakcje między różnymi elementami szeregu czasowego. Sieci LSTM są bardziej efektywne w uwzględnianiu lokalnych zależności i modelowaniu sekwencyjnych wzorców czasowych. Wybór odpowiedniego typu sieci zależy od charakteru danych i analizowanego problemu.

8. Źródła:

- https://towardsdatascience.com/multivariate-time-series-forecasting-with-transformers-384dc6ce989b
- https://docs.google.com/spreadsheets/d/1ierEhD6gcq51HAm433knjnVwey4ZE5DCnu1bW7PRG3E/edit#gid=1309014089
- https://hub.packtpub.com/cross-validation-strategies-for-time-series-forecasting-tutorial/
- https://towardsdatascience.com/dont-use-k-fold-validation-for-time-series-forecasting-30b724aaea64
- https://medium.com/@soumyachess1496/cross-validation-in-time-series-566ae4981ce4
- https://towardsdatascience.com/stock-predictions-with-state-of-the-art-transformer-and-time-embeddings-3a4485237de6
- https://keras.io/api/
- https://scikit-learn.org/stable/modules/classes.html