

```

In [ ]: ###necessary libraries###
import numpy as np
import pandas as pd
from seglearn.transform import FeatureRep, SegmentXYForecast, last
from subprocess import check_output
from keras.layers import Dense, Activation, Dropout, Input, LSTM, Flatten
from keras.models import Model
from sklearn.metrics import r2_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
import matplotlib.pyplot as plt
from numpy import newaxis
import glob
import os
from datetime import datetime
import math
from numpy.random import seed
import tensorflow as tf
import warnings
from sklearn.exceptions import DataConversionWarning

model_seed = 100
# ensure same output results
seed(101)
tf.random.set_seed(model_seed)

# file where csv files lies
path = r'C:\Users\victo\Master_Thesis\merging_data\bmw\hourly\merged_files'
all_files = glob.glob(os.path.join(path, "*.csv"))

# read files to pandas frame
list_of_files = []

for filename in all_files:
    list_of_files.append(pd.read_csv(filename,
                                     sep=',',
                                     )
                        )

# Concatenate all content of files into one DataFrames
concatenate_dataframe = pd.concat(list_of_files,
                                  ignore_index=True,
                                  axis=0,
                                  )

# print(concatenate_dataframe)

### analysis with flair sentiment content
new_df_flair_content = concatenate_dataframe[['OPEN',
                                              'HIGH',
                                              'LOW',
                                              'CLOSE',
                                              'VOLUME',
                                              'flair_sentiment_content_score']]

new_df_flair_content['flair_sentiment_content_score'] = new_df_flair_content['flair_sentiment_content_score'].fillna(0)
new_df_flair_content[['OPEN', 'HIGH', 'LOW', 'CLOSE', 'VOLUME', 'flair_sentiment_content_score']].astype(np.float64)
# print(new_df)

# train, valid, test split

```

```
valid_test_size_split_flair_content = 0.1

X_train_flair_content, \
X_else_flair_content, \
y_train_flair_content, \
y_else_flair_content = train_test_split(new_df_flair_content,
                                       new_df_flair_content['OPEN'],
                                       test_size=valid_test_size_split_flair_content*2,
                                       shuffle=False)

X_valid_flair_content, \
X_test_flair_content, \
y_valid_flair_content, \
y_test_flair_content = train_test_split(X_else_flair_content,
                                       y_else_flair_content,
                                       test_size=0.5,
                                       shuffle=False)

warnings.filterwarnings(action='ignore', category=DataConversionWarning)

# normalize data
def minmax_scale_flair_content(df_x, series_y, normalizers_flair_content = None):
    features_to_minmax = ['OPEN', 'HIGH', 'LOW', 'CLOSE', 'VOLUME', 'flair_sentiment_content_score']

    if not normalizers_flair_content:
        normalizers_flair_content = {}

    for feat in features_to_minmax:
        if feat not in normalizers_flair_content:
            normalizers_flair_content[feat] = MinMaxScaler()
            normalizers_flair_content[feat].fit(df_x[feat].values.reshape(-1, 1))

        df_x[feat] = normalizers_flair_content[feat].transform(df_x[feat].values.reshape(-1, 1))

    series_y = normalizers_flair_content['OPEN'].transform(series_y.values.reshape(-1, 1))

    return df_x, series_y, normalizers_flair_content

X_train_norm_flair_content, \
y_train_norm_flair_content, \
normalizers_flair_content = minmax_scale_flair_content(X_train_flair_content,
                                                       y_train_flair_content)

X_valid_norm_flair_content, \
y_valid_norm_flair_content, \
_ = minmax_scale_flair_content(X_valid_flair_content,
                               y_valid_flair_content,
                               normalizers_flair_content=normalizers_flair_content)

X_test_norm_flair_content, \
y_test_norm_flair_content, \
_ = minmax_scale_flair_content(X_test_flair_content,
                               y_test_flair_content,
                               normalizers_flair_content=normalizers_flair_content)

# Creating target (y) and "windows" (X) for modeling
```

```

TIME_WINDOW_flair_content = 45
FORECAST_DISTANCE_flair_content = 9

segmenter_flair_content = SegmentXYForecast(width=TIME_WINDOW_flair_content,
                                             step=1,
                                             y_func=last,
                                             forecast=FORECAST_DISTANCE_flair_content,
                                             t

                                             )

X_train_rolled_flair_content, \
y_train_rolled_flair_content, \
_ = segmenter_flair_content.fit_transform([X_train_norm_flair_content.values],
                                         [y_train_norm_flair_content.flatten()])

X_valid_rolled_flair_content, \
y_valid_rolled_flair_content, \
_ = segmenter_flair_content.fit_transform([X_valid_norm_flair_content.values],
                                         [y_valid_norm_flair_content.flatten()])

X_test_rolled_flair_content, \
y_test_rolled_flair_content, \
_ = segmenter_flair_content.fit_transform([X_test_norm_flair_content.values],
                                         [y_test_norm_flair_content.flatten()])

# LSTM Model
first_lstm_size_flair_content = 75
second_lstm_size_flair_content = 40
dropout_flair_content = 0.1
EPOCHS_flair_content = 50
BATCH_SIZE_flair_content = 32
column_count_flair_content = len(X_train_norm_flair_content.columns)
# model with use of Funcational API of Keras
# input layer
input_layer_flair_content = Input(shape=(TIME_WINDOW_flair_content, column_count_flair_content))
# first LSTM layer
first_lstm_flair_content = LSTM(first_lstm_size_flair_content,
                                return_sequences=True,
                                dropout=dropout_flair_content)(input_layer_flair_content)
# second LSTM layer
second_lstm_flair_content = LSTM(second_lstm_size_flair_content,
                                return_sequences=False,
                                dropout=dropout_flair_content)(first_lstm_flair_content)
# output layer
output_layer_flair_content = Dense(1)(second_lstm_flair_content)
# creating Model
model_flair_content = Model(inputs=input_layer_flair_content, outputs=output_layer_flair_content)
# compile model
model_flair_content.compile(optimizer='adam', loss='mean_absolute_error')
# model summary
model_flair_content.summary()
print(' ')
print("-----")
print(' ')
# fitting model
hist_flair_content = model_flair_content.fit(x=X_train_rolled_flair_content,
                                             y=y_train_rolled_flair_content,

```

```

        batch_size=BATCH_SIZE_flair_content,
        validation_data=(X_valid_rolled_flair_
content,
                        y_valid_rolled_flair_
content

                        ),
        epochs=EPOCHS_flair_content,
        verbose=1,
        shuffle=False
    )

    print(' ')
    print("-----")
    print(' ')

    plt.plot(hist_flair_content.history['loss'], label='train_flair_content')
    plt.plot(hist_flair_content.history['val_loss'], label='test_flair_content')
    plt.legend()
    plt.show()
    print(' ')
    print("-----")
    print(' ')
    rms_LSTM_flair_content = math.sqrt(min(hist_flair_content.history['val_loss']))
    print(' ')
    print("-----")
    print(' ')
    # predicting stock prices
    predicted_stock_price_flair_content = model_flair_content.predict(X_test_rolled_flair_content)

    predicted_stock_price_flair_content = normalizers_flair_content['OPEN']\
        .inverse_transform(predicted_stock_price_flair_content).reshape(-1, 1)
    print(' ')
    print("Root mean squared error on valid:", rms_LSTM_flair_content)
    print(' ')
    print("-----")
    print(' ')
    print("Root mean squared error on valid inverse transformed from normalization:",
          normalizers_flair_content["OPEN"].inverse_transform(np.array([rms_LSTM_flair_content]).reshape(1, -1)))
    print(' ')
    print("-----")
    print(' ')
    print(predicted_stock_price_flair_content)

    ### analysis with flair header
    new_df_flair_header = concatenate_dataframe(['OPEN',
                                                'HIGH',
                                                'LOW',
                                                'CLOSE',
                                                'VOLUME',
                                                'flair_sentiment_header_score'])

    new_df_flair_header['flair_sentiment_header_score'] = new_df_flair_header['flair_sentiment_header_score'].fillna(0)
    new_df_flair_header[['OPEN', 'HIGH', 'LOW', 'CLOSE', 'VOLUME', 'flair_sentiment_header_score']].astype(np.float64)
    # print(new_df)

    # train, valid, test split
    valid_test_size_split_flair_header = 0.1

    X_train_flair_header, \
    X_else_flair_header, \

```

```
y_train_flair_header, \
y_else_flair_header = train_test_split(new_df_flair_header,
                                       new_df_flair_header['OPEN'],
                                       test_size=valid_test_size_split_flair_header
                                       *2,
                                       shuffle=False)

X_valid_flair_header, \
X_test_flair_header, \
y_valid_flair_header, \
y_test_flair_header = train_test_split(X_else_flair_header,
                                       y_else_flair_header,
                                       test_size=0.5,
                                       shuffle=False)

warnings.filterwarnings(action='ignore', category=DataConversionWarning)

# normalize data
def minmax_scale_flair_header(df_x, series_y, normalizers_flair_header = None):
    features_to_minmax = ['OPEN', 'HIGH', 'LOW', 'CLOSE', 'VOLUME', 'flair_sentimen
t_header_score']

    if not normalizers_flair_header:
        normalizers_flair_header = {}

    for feat in features_to_minmax:
        if feat not in normalizers_flair_header:
            normalizers_flair_header[feat] = MinMaxScaler()
            normalizers_flair_header[feat].fit(df_x[feat].values.reshape(-1, 1))

        df_x[feat] = normalizers_flair_header[feat].transform(df_x[feat].values.res
hape(-1, 1))

        series_y = normalizers_flair_header['OPEN'].transform(series_y.values.reshape(-
1, 1))

    return df_x, series_y, normalizers_flair_header

X_train_norm_flair_header, \
y_train_norm_flair_header, \
normalizers_flair_header = minmax_scale_flair_header(X_train_flair_header,
                                                    y_train_flair_header
                                                    )

X_valid_norm_flair_header, \
y_valid_norm_flair_header, \
_ = minmax_scale_flair_header(X_valid_flair_header,
                              y_valid_flair_header,
                              normalizers_flair_header=normalizers_flair_header
                              )

X_test_norm_flair_header, \
y_test_norm_flair_header, \
_ = minmax_scale_flair_header(X_test_flair_header,
                              y_test_flair_header,
                              normalizers_flair_header=normalizers_flair_header
                              )

# Creating target (y) and "windows" (X) for modeling
TIME_WINDOW_flair_header = 45
FORECAST_DISTANCE_flair_header = 9

segmenter_flair_header = SegmentXYForecast(width=TIME_WINDOW_flair_header,
```

```

        step=1,
        y_func=last,
        forecast=FORECAST_DISTANCE_flair_header
    )

X_train_rolled_flair_header, \
y_train_rolled_flair_header, \
_ = segmenter_flair_header.fit_transform([X_train_norm_flair_header.values],
                                         [y_train_norm_flair_header.flatten()])

X_valid_rolled_flair_header, \
y_valid_rolled_flair_header, \
_ = segmenter_flair_header.fit_transform([X_valid_norm_flair_header.values],
                                         [y_valid_norm_flair_header.flatten()])

X_test_rolled_flair_header, \
y_test_rolled_flair_header, \
_ = segmenter_flair_header.fit_transform([X_test_norm_flair_header.values],
                                         [y_test_norm_flair_header.flatten()])

# LSTM Model
first_lstm_size_flair_header = 75
second_lstm_size_flair_header = 40
dropout_flair_header = 0.1
EPOCHS_flair_header = 50
BATCH_SIZE_flair_header = 32
column_count_flair_header = len(X_train_norm_flair_header.columns)
# model with use of Funcational API of Keras
# input layer
input_layer_flair_header = Input(shape=(TIME_WINDOW_flair_header, column_count_flair_header))
# first LSTM layer
first_lstm_flair_header = LSTM(first_lstm_size_flair_header,
                               return_sequences=True,
                               dropout=dropout_flair_header)(input_layer_flair_header)
# second LSTM layer
second_lstm_flair_header = LSTM(second_lstm_size_flair_header,
                                return_sequences=False,
                                dropout=dropout_flair_header)(first_lstm_flair_header)
# output layer
output_layer_flair_header = Dense(1)(second_lstm_flair_header)
# creating Model
model_flair_header = Model(inputs=input_layer_flair_header, outputs=output_layer_flair_header)
# compile model
model_flair_header.compile(optimizer='adam', loss='mean_absolute_error')
# model summary
model_flair_header.summary()
print(' ')
print("-----")
print(' ')
# fitting model
hist_flair_header = model_flair_header.fit(x=X_train_rolled_flair_header,
                                           y=y_train_rolled_flair_header,
                                           batch_size=BATCH_SIZE_flair_header,
                                           validation_data=(X_valid_rolled_flair_header,
                                                             y_valid_rolled_flair_header))

```

```

        ),
        epochs=EPOCHS_flair_header,
        verbose=1,
        shuffle=False
    )

    print(' ')
    print("-----")
    print(' ')

    plt.plot(hist_flair_header.history['loss'], label='train_flair_header')
    plt.plot(hist_flair_header.history['val_loss'], label='test_flair_header')
    plt.legend()
    plt.show()
    print(' ')
    print("-----")
    print(' ')
    rms_LSTM_flair_header = math.sqrt(min(hist_flair_header.history['val_loss']))
    print(' ')
    print("-----")
    print(' ')
    # predicting stock prices
    predicted_stock_price_flair_header = model_flair_header.predict(X_test_rolled_flair_header)

    predicted_stock_price_flair_header = normalizers_flair_header['OPEN']\
        .inverse_transform(predicted_stock_price_flair_header).reshape(-1, 1)
    print(' ')
    print("Root mean squared error on valid:", rms_LSTM_flair_header)
    print(' ')
    print("-----")
    print(' ')
    print("Root mean squared error on valid inverse transformed from normalization:",
          normalizers_flair_header["OPEN"].inverse_transform(np.array([rms_LSTM_flair_header]).reshape(1, -1)))
    print(' ')
    print("-----")
    print(' ')
    print(predicted_stock_price_flair_header)

    ### analysis with textblob sentiment content
    new_df_textblob_content = concatenate_dataframe[['OPEN',
                                                    'HIGH',
                                                    'LOW',
                                                    'CLOSE',
                                                    'VOLUME',
                                                    'polarity_textblob_sentiment_content']]

    new_df_textblob_content['polarity_textblob_sentiment_content'] = new_df_textblob_content['polarity_textblob_sentiment_content'].fillna(0)
    new_df_textblob_content[['OPEN', 'HIGH', 'LOW', 'CLOSE', 'VOLUME', 'polarity_textblob_sentiment_content']].astype(np.float64)
    # print(new_df)

    # train, valid, test split
    valid_test_size_split_textblob_content = 0.1

    X_train_textblob_content, \
    X_else_textblob_content, \
    y_train_textblob_content, \
    y_else_textblob_content = train_test_split(new_df_textblob_content,
                                              new_df_textblob_content['OPEN'],
                                              test_size=valid_test_size_split_textblob

```

```

_content*2,

                                shuffle=False)

X_valid_textblob_content, \
X_test_textblob_content, \
y_valid_textblob_content, \
y_test_textblob_content = train_test_split(X_else_textblob_content,
                                            y_else_textblob_content,
                                            test_size=0.5,
                                            shuffle=False)

warnings.filterwarnings(action='ignore', category=DataConversionWarning)

# normalize data
def minmax_scale_textblob_content(df_x, series_y, normalizers_textblob_content = No
ne):
    features_to_minmax = ['OPEN', 'HIGH', 'LOW', 'CLOSE', 'VOLUME', 'polarity_textb
lob_sentiment_content']

    if not normalizers_textblob_content:
        normalizers_textblob_content = {}

    for feat in features_to_minmax:
        if feat not in normalizers_textblob_content:
            normalizers_textblob_content[feat] = MinMaxScaler()
            normalizers_textblob_content[feat].fit(df_x[feat].values.reshape(-1,
1))

            df_x[feat] = normalizers_textblob_content[feat].transform(df_x[feat].value
s.reshape(-1, 1))

            series_y = normalizers_textblob_content['OPEN'].transform(series_y.values.resha
pe(-1, 1))

        return df_x, series_y, normalizers_textblob_content

X_train_norm_textblob_content, \
y_train_norm_textblob_content, \
normalizers_textblob_content = minmax_scale_textblob_content(X_train_textblob_conte
nt,
                                                                y_train_textblob_conte
nt

X_valid_norm_textblob_content, \
y_valid_norm_textblob_content, \
_ = minmax_scale_textblob_content(X_valid_textblob_content,
                                y_valid_textblob_content,
                                normalizers_textblob_content=normalizers_textblob
_content

X_test_norm_textblob_content, \
y_test_norm_textblob_content, \
_ = minmax_scale_textblob_content(X_test_textblob_content,
                                y_test_textblob_content,
                                normalizers_textblob_content=normalizers_textblob
_content

# Creating target (y) and "windows" (X) for modeling
TIME_WINDOW_textblob_content = 45
FORECAST_DISTANCE_textblob_content = 9

```



```

segmenter_textblob_content = SegmentXYForecast(width=TIME_WINDOW_textblob_content,
                                                step=1,
                                                y_func=last,
                                                forecast=FORECAST_DISTANCE_textblob_
content
                                                )

X_train_rolled_textblob_content, \
y_train_rolled_textblob_content, \
_ = segmenter_textblob_content.fit_transform([X_train_norm_textblob_content.value
s],
                                                [y_train_norm_textblob_content.flatten
()])
                                                )

X_valid_rolled_textblob_content, \
y_valid_rolled_textblob_content, \
_ = segmenter_textblob_content.fit_transform([X_valid_norm_textblob_content.value
s],
                                                [y_valid_norm_textblob_content.flatten
()])
                                                )

X_test_rolled_textblob_content, \
y_test_rolled_textblob_content, \
_ = segmenter_textblob_content.fit_transform([X_test_norm_textblob_content.values],
                                                [y_test_norm_textblob_content.flatten
()])
                                                )

# LSTM Model
first_lstm_size_textblob_content = 75
second_lstm_size_textblob_content = 40
dropout_textblob_content = 0.1
EPOCHS_textblob_content = 50
BATCH_SIZE_textblob_content = 32
column_count_textblob_content = len(X_train_norm_textblob_content.columns)
# model with use of Funcational API of Keras
# input layer
input_layer_textblob_content = Input(shape=(TIME_WINDOW_textblob_content, column_co
unt_textblob_content))
# first LSTM layer
first_lstm_textblob_content = LSTM(first_lstm_size_textblob_content,
                                   return_sequences=True,
                                   dropout=dropout_textblob_content)(input_layer_te
xtblob_content)
# second LSTM layer
second_lstm_textblob_content = LSTM(second_lstm_size_textblob_content,
                                   return_sequences=False,
                                   dropout=dropout_textblob_content)(first_lstm_te
xtblob_content)
# output layer
output_layer_textblob_content = Dense(1)(second_lstm_textblob_content)
# creating Model
model_textblob_content = Model(inputs=input_layer_textblob_content, outputs=output_
layer_textblob_content)
# compile model
model_textblob_content.compile(optimizer='adam', loss='mean_absolute_error')
# model summary
model_textblob_content.summary()
print(' ')
print("-----")
print(' ')

```

```

# fitting model
hist_textblob_content = model_textblob_content.fit(x=X_train_rolled_textblob_content,
                                                    y=y_train_rolled_textblob_content,
                                                    batch_size=BATCH_SIZE_textblob_content,
                                                    validation_data=(X_valid_rolled_textblob_content,
                                                                    y_valid_rolled_textblob_content),
                                                    epochs=EPOCHS_textblob_content,
                                                    verbose=1,
                                                    shuffle=False
                                                    )

print(' ')
print("-----")
print(' ')

plt.plot(hist_textblob_content.history['loss'], label='train_textblob_content')
plt.plot(hist_textblob_content.history['val_loss'], label='test_textblob_content')
plt.legend()
plt.show()
print(' ')
print("-----")
print(' ')
rms_LSTM_textblob_content = math.sqrt(min(hist_textblob_content.history['val_loss']
))
print(' ')
print("-----")
print(' ')
# predicting stock prices
predicted_stock_price_textblob_content = model_textblob_content.predict(X_test_rolled_textblob_content)

predicted_stock_price_textblob_content = normalizers_textblob_content['OPEN']\
.inverse_transform(predicted_stock_price_textblob_content).reshape(-1, 1)
print(' ')
print("Root mean squared error on valid:", rms_LSTM_textblob_content)
print(' ')
print("-----")
print(' ')
print("Root mean squared error on valid inverse transformed from normalization:",
      normalizers_textblob_content["OPEN"].inverse_transform(np.array([rms_LSTM_textblob_content]).reshape(1, -1)))
print(' ')
print("-----")
print(' ')
print(predicted_stock_price_textblob_content)

### analysis with textblob header
new_df_textblob_header = concatenate_dataframe(['OPEN',
                                                'HIGH',
                                                'LOW',
                                                'CLOSE',
                                                'VOLUME',
                                                'polarity_textblob_sentiment_header'
                                                ])

new_df_textblob_header = new_df_textblob_header.fillna(0)
new_df_textblob_header[['OPEN', 'HIGH', 'LOW', 'CLOSE', 'VOLUME', 'polarity_textblob_sentiment_header']].astype(np.float64)

```

```

print(new_df_textblob_header)

# train, valid, test split
valid_test_size_split_textblob_header = 0.1

X_train_textblob_header, \
X_else_textblob_header, \
y_train_textblob_header, \
y_else_textblob_header = train_test_split(new_df_textblob_header,
                                          new_df_textblob_header['OPEN'],
                                          test_size=valid_test_size_split_textblob_
header*2,
                                          shuffle=False)

X_valid_textblob_header, \
X_test_textblob_header, \
y_valid_textblob_header, \
y_test_textblob_header = train_test_split(X_else_textblob_header,
                                          y_else_textblob_header,
                                          test_size=0.5,
                                          shuffle=False)

warnings.filterwarnings(action='ignore', category=DataConversionWarning)

# normalize data
def minmax_scale_textblob_header(df_x, series_y, normalizers_textblob_header = Non
e):
    features_to_minmax = ['OPEN', 'HIGH', 'LOW', 'CLOSE', 'VOLUME', 'polarity_textb
lob_sentiment_header']

    if not normalizers_textblob_header:
        normalizers_textblob_header = {}

    for feat in features_to_minmax:
        if feat not in normalizers_textblob_header:
            normalizers_textblob_header[feat] = MinMaxScaler()
            normalizers_textblob_header[feat].fit(df_x[feat].values.reshape(-1, 1))

        df_x[feat] = normalizers_textblob_header[feat].transform(df_x[feat].values.
reshape(-1, 1))

        series_y = normalizers_textblob_header['OPEN'].transform(series_y.values.reshap
e(-1, 1))

    return df_x, series_y, normalizers_textblob_header

X_train_norm_textblob_header, \
y_train_norm_textblob_header, \
normalizers_textblob_header = minmax_scale_textblob_header(X_train_textblob_header,
                                                            y_train_textblob_header
)

X_valid_norm_textblob_header, \
y_valid_norm_textblob_header, \
_ = minmax_scale_textblob_header(X_valid_textblob_header,
                                y_valid_textblob_header,
                                normalizers_textblob_header=normalizers_textblob_h
eader
                                )

X_test_norm_textblob_header, \
y_test_norm_textblob_header, \
_ = minmax_scale_textblob_header(X_test_textblob_header,

```

```

        y_test_textblob_header,
        normalizers_textblob_header=normalizers_textblob_h
eader

    )

# Creating target (y) and "windows" (X) for modeling
TIME_WINDOW_textblob_header = 45
FORECAST_DISTANCE_textblob_header = 9

segmenter_textblob_header = SegmentXYForecast(width=TIME_WINDOW_textblob_header,
                                                step=1,
                                                y_func=last,
                                                forecast=FORECAST_DISTANCE_textblob_h
eader

                                                )

X_train_rolled_textblob_header, \
y_train_rolled_textblob_header, \
_ = segmenter_textblob_header.fit_transform([X_train_norm_textblob_header.values],
                                            [y_train_norm_textblob_header.flatten
()])

                                            )

X_valid_rolled_textblob_header, \
y_valid_rolled_textblob_header, \
_ = segmenter_textblob_header.fit_transform([X_valid_norm_textblob_header.values],
                                            [y_valid_norm_textblob_header.flatten
()])

                                            )

X_test_rolled_textblob_header, \
y_test_rolled_textblob_header, \
_ = segmenter_textblob_header.fit_transform([X_test_norm_textblob_header.values],
                                            [y_test_norm_textblob_header.flatten()]
                                            )

# LSTM Model
first_lstm_size_textblob_header = 75
second_lstm_size_textblob_header = 40
dropout_textblob_header = 0.1
EPOCHS_textblob_header = 50
BATCH_SIZE_textblob_header = 32
column_count_textblob_header = len(X_train_norm_textblob_header.columns)
# model with use of Funcational API of Keras
# input layer
input_layer_textblob_header = Input(shape=(TIME_WINDOW_textblob_header, column_coun
t_textblob_header))
# first LSTM layer
first_lstm_textblob_header = LSTM(first_lstm_size_textblob_header,
                                  return_sequences=True,
                                  dropout=dropout_textblob_header)(input_layer_text
blob_header)
# second LTSM layer
second_lstm_textblob_header = LSTM(second_lstm_size_textblob_header,
                                  return_sequences=False,
                                  dropout=dropout_textblob_header)(first_lstm_text
blob_header)
# output layer
output_layer_textblob_header = Dense(1)(second_lstm_textblob_header)
# creating Model
model_textblob_header = Model(inputs=input_layer_textblob_header, outputs=output_la
yer_textblob_header)
# compile model
model_textblob_header.compile(optimizer='adam', loss='mean_absolute_error')

```

```

# model summary
model_textblob_header.summary()
print(' ')
print("-----")
print(' ')
# fitting model
hist_textblob_header = model_textblob_header.fit(x=X_train_rolled_textblob_header,
                                                  y=y_train_rolled_textblob_header,
                                                  batch_size=BATCH_SIZE_textblob_header,
                                                  validation_data=(X_valid_rolled_textblob_header,
                                                                y_valid_rolled_textblob_header),
                                                  epochs=EPOCHS_textblob_header,
                                                  verbose=1,
                                                  shuffle=False
                                                  )

print(' ')
print("-----")
print(' ')

plt.plot(hist_textblob_header.history['loss'], label='train_textblob_header')
plt.plot(hist_textblob_header.history['val_loss'], label='test_textblob_header')
plt.legend()
plt.show()
print(' ')
print("-----")
print(' ')
rms_LSTM_textblob_header = math.sqrt(min(hist_textblob_header.history['val_loss']))
print(' ')
print("-----")
print(' ')
# predicting stock prices
predicted_stock_price_textblob_header = model_textblob_header.predict(X_test_rolled_textblob_header)

predicted_stock_price_textblob_header = normalizers_textblob_header['OPEN']\
    .inverse_transform(predicted_stock_price_textblob_header).reshape(-1, 1)
print(' ')
print("Root mean squared error on valid:", rms_LSTM_textblob_header)
print(' ')
print("-----")
print(' ')
print("Root mean squared error on valid inverse transformed from normalization:",
      normalizers_textblob_header["OPEN"].inverse_transform(np.array([rms_LSTM_textblob_header]).reshape(1, -1)))
print(' ')
print("-----")
print(' ')
print(predicted_stock_price_textblob_header)

### analysis with vader sentiment content
new_df_vader_content = concatenate_dataframe[['OPEN',
                                              'HIGH',
                                              'LOW',
                                              'CLOSE',
                                              'VOLUME',
                                              'compound_vader_articel_content']]

new_df_vader_content['compound_vader_articel_content'] = new_df_vader_content['compound_vader_articel_content'].fillna(0)

```

```
new_df_vader_content[['OPEN', 'HIGH', 'LOW', 'CLOSE', 'VOLUME', 'compound_vader_articel_content']].astype(np.float64)
# print(new_df)

# train, valid, test split
valid_test_size_split_vader_content = 0.1

X_train_vader_content, \
X_else_vader_content, \
y_train_vader_content, \
y_else_vader_content = train_test_split(new_df_vader_content,
                                       new_df_vader_content['OPEN'],
                                       test_size=valid_test_size_split_vader_content*2,
                                       shuffle=False)

X_valid_vader_content, \
X_test_vader_content, \
y_valid_vader_content, \
y_test_vader_content = train_test_split(X_else_vader_content,
                                       y_else_vader_content,
                                       test_size=0.5,
                                       shuffle=False)

warnings.filterwarnings(action='ignore', category=DataConversionWarning)

# normalize data
def minmax_scale_vader_content(df_x, series_y, normalizers_vader_content = None):
    features_to_minmax = ['OPEN', 'HIGH', 'LOW', 'CLOSE', 'VOLUME', 'compound_vader_articel_content']

    if not normalizers_vader_content:
        normalizers_vader_content = {}

    for feat in features_to_minmax:
        if feat not in normalizers_vader_content:
            normalizers_vader_content[feat] = MinMaxScaler()
            normalizers_vader_content[feat].fit(df_x[feat].values.reshape(-1, 1))

        df_x[feat] = normalizers_vader_content[feat].transform(df_x[feat].values.reshape(-1, 1))

    series_y = normalizers_vader_content['OPEN'].transform(series_y.values.reshape(-1, 1))

    return df_x, series_y, normalizers_vader_content

X_train_norm_vader_content, \
y_train_norm_vader_content, \
normalizers_vader_content = minmax_scale_vader_content(X_train_vader_content,
                                                         y_train_vader_content)

X_valid_norm_vader_content, \
y_valid_norm_vader_content, \
_ = minmax_scale_vader_content(X_valid_vader_content,
                               y_valid_vader_content,
                               normalizers_vader_content=normalizers_vader_content)

X_test_norm_vader_content, \
y_test_norm_vader_content, \
_ = minmax_scale_vader_content(X_test_vader_content,
```

```

        y_test_vader_content,
        normalizers_vader_content=normalizers_vader_content
    )

# Creating target (y) and "windows" (X) for modeling
TIME_WINDOW_vader_content = 45
FORECAST_DISTANCE_vader_content = 9

segmenter_vader_content = SegmentXYForecast(width=TIME_WINDOW_vader_content,
                                             step=1,
                                             y_func=last,
                                             forecast=FORECAST_DISTANCE_vader_content)

X_train_rolled_vader_content, \
y_train_rolled_vader_content, \
_ = segmenter_vader_content.fit_transform([X_train_norm_vader_content.values],
                                          [y_train_norm_vader_content.flatten()])

X_valid_rolled_vader_content, \
y_valid_rolled_vader_content, \
_ = segmenter_vader_content.fit_transform([X_valid_norm_vader_content.values],
                                          [y_valid_norm_vader_content.flatten()])

X_test_rolled_vader_content, \
y_test_rolled_vader_content, \
_ = segmenter_vader_content.fit_transform([X_test_norm_vader_content.values],
                                          [y_test_norm_vader_content.flatten()])

# LSTM Model
first_lstm_size_vader_content = 75
second_lstm_size_vader_content = 40
dropout_vader_content = 0.1
EPOCHS_vader_content = 50
BATCH_SIZE_vader_content = 32
column_count_vader_content = len(X_train_norm_vader_content.columns)
# model with use of Functional API of Keras
# input layer
input_layer_vader_content = Input(shape=(TIME_WINDOW_vader_content, column_count_vader_content))
# first LSTM layer
first_lstm_vader_content = LSTM(first_lstm_size_vader_content,
                                return_sequences=True,
                                dropout=dropout_vader_content)(input_layer_vader_content)
# second LSTM layer
second_lstm_vader_content = LSTM(second_lstm_size_vader_content,
                                return_sequences=False,
                                dropout=dropout_vader_content)(first_lstm_vader_content)
# output layer
output_layer_vader_content = Dense(1)(second_lstm_vader_content)
# creating Model
model_vader_content = Model(inputs=input_layer_vader_content, outputs=output_layer_vader_content)
# compile model
model_vader_content.compile(optimizer='adam', loss='mean_absolute_error')
# model summary
model_vader_content.summary()
print(' ')

```

```

print("-----")
print(' ')
# fitting model
hist_vader_content = model_vader_content.fit(x=X_train_rolled_vader_content,
                                             y=y_train_rolled_vader_content,
                                             batch_size=BATCH_SIZE_vader_content,
                                             validation_data=(X_valid_rolled_vader_
content,
                                                             y_valid_rolled_vader_
content
                                                             ),
                                             epochs=EPOCHS_vader_content,
                                             verbose=1,
                                             shuffle=False
)

print(' ')
print("-----")
print(' ')

plt.plot(hist_vader_content.history['loss'], label='train_vader_content')
plt.plot(hist_vader_content.history['val_loss'], label='test_vader_content')
plt.legend()
plt.show()
print(' ')
print("-----")
print(' ')
rms_LSTM_vader_content = math.sqrt(min(hist_vader_content.history['val_loss']))
print(' ')
print("-----")
print(' ')
# predicting stock prices
predicted_stock_price_vader_content = model_vader_content.predict(X_test_rolled_vad
er_content)

predicted_stock_price_vader_content = normalizers_vader_content['OPEN']\
.inverse_transform(predicted_stock_price_vade
r_content).reshape(-1, 1)
print(' ')
print("Root mean squared error on valid:", rms_LSTM_vader_content)
print(' ')
print("-----")
print(' ')
print("Root mean squared error on valid inverse transformed from normalization:",
      normalizers_vader_content['OPEN'].inverse_transform(np.array([rms_LSTM_vader_
content]).reshape(1, -1)))
print(' ')
print("-----")
print(' ')
print(predicted_stock_price_vader_content)

### analysis with vader header
new_df_vader_header = concatenate_dataframe[['OPEN',
                                             'HIGH',
                                             'LOW',
                                             'CLOSE',
                                             'VOLUME',
                                             'compound_vader_header']]

new_df_vader_header = new_df_vader_header.fillna(0)
new_df_vader_header[['OPEN', 'HIGH', 'LOW', 'CLOSE', 'VOLUME', 'compound_vader_head
er']].astype(np.float64)
print(new_df_vader_header)

# train, valid, test split

```



```
valid_test_size_split_vader_header = 0.1

X_train_vader_header, \
X_else_vader_header, \
y_train_vader_header, \
y_else_vader_header = train_test_split(new_df_vader_header,
                                       new_df_vader_header['OPEN'],
                                       test_size=valid_test_size_split_vader_header
                                       *2,
                                       shuffle=False)

X_valid_vader_header, \
X_test_vader_header, \
y_valid_vader_header, \
y_test_vader_header = train_test_split(X_else_vader_header,
                                       y_else_vader_header,
                                       test_size=0.5,
                                       shuffle=False)

warnings.filterwarnings(action='ignore', category=DataConversionWarning)

# normalize data
def minmax_scale_vader_header(df_x, series_y, normalizers_vader_header = None):
    features_to_minmax = ['OPEN', 'HIGH', 'LOW', 'CLOSE', 'VOLUME', 'compound_vader_header']

    if not normalizers_vader_header:
        normalizers_vader_header = {}

    for feat in features_to_minmax:
        if feat not in normalizers_vader_header:
            normalizers_vader_header[feat] = MinMaxScaler()
            normalizers_vader_header[feat].fit(df_x[feat].values.reshape(-1, 1))

        df_x[feat] = normalizers_vader_header[feat].transform(df_x[feat].values.reshape(-1, 1))

    series_y = normalizers_vader_header['OPEN'].transform(series_y.values.reshape(-1, 1))

    return df_x, series_y, normalizers_vader_header

X_train_norm_vader_header, \
y_train_norm_vader_header, \
normalizers_vader_header = minmax_scale_vader_header(X_train_vader_header,
                                                    y_train_vader_header
                                                    )

X_valid_norm_vader_header, \
y_valid_norm_vader_header, \
_ = minmax_scale_vader_header(X_valid_vader_header,
                              y_valid_vader_header,
                              normalizers_vader_header=normalizers_vader_header
                              )

X_test_norm_vader_header, \
y_test_norm_vader_header, \
_ = minmax_scale_vader_header(X_test_vader_header,
                              y_test_vader_header,
                              normalizers_vader_header=normalizers_vader_header
                              )

# Creating target (y) and "windows" (X) for modeling
```

```

TIME_WINDOW_vader_header = 45
FORECAST_DISTANCE_vader_header = 9

segmenter_vader_header = SegmentXYForecast(width=TIME_WINDOW_vader_header,
                                             step=1,
                                             y_func=last,
                                             forecast=FORECAST_DISTANCE_vader_header
                                             )

X_train_rolled_vader_header, \
y_train_rolled_vader_header, \
_ = segmenter_vader_header.fit_transform([X_train_norm_vader_header.values],
                                         [y_train_norm_vader_header.flatten()])

X_valid_rolled_vader_header, \
y_valid_rolled_vader_header, \
_ = segmenter_vader_header.fit_transform([X_valid_norm_vader_header.values],
                                         [y_valid_norm_vader_header.flatten()])

X_test_rolled_vader_header, \
y_test_rolled_vader_header, \
_ = segmenter_vader_header.fit_transform([X_test_norm_vader_header.values],
                                         [y_test_norm_vader_header.flatten()])

# LSTM Model
first_lstm_size_vader_header = 75
second_lstm_size_vader_header = 40
dropout_vader_header = 0.1
EPOCHS_vader_header = 50
BATCH_SIZE_vader_header = 32
column_count_vader_header = len(X_train_norm_vader_header.columns)
# model with use of Funcational API of Keras
# input layer
input_layer_vader_header = Input(shape=(TIME_WINDOW_vader_header, column_count_vader_header))
# first LSTM layer
first_lstm_vader_header = LSTM(first_lstm_size_vader_header,
                                return_sequences=True,
                                dropout=dropout_vader_header)(input_layer_vader_header)
# second LSTM layer
second_lstm_vader_header = LSTM(second_lstm_size_vader_header,
                                return_sequences=False,
                                dropout=dropout_vader_header)(first_lstm_vader_header)
# output layer
output_layer_vader_header = Dense(1)(second_lstm_vader_header)
# creating Model
model_vader_header = Model(inputs=input_layer_vader_header, outputs=output_layer_vader_header)
# compile model
model_vader_header.compile(optimizer='adam', loss='mean_absolute_error')
# model summary
model_vader_header.summary()
print(' ')
print("-----")
print(' ')
# fitting model
hist_vader_header = model_vader_header.fit(x=X_train_rolled_vader_header,
                                           y=y_train_rolled_vader_header,
                                           batch_size=BATCH_SIZE_vader_header,

```

```

validation_data=(X_valid_rolled_vader_header,
                  y_valid_rolled_vader_header),
epochs=EPOCHS_vader_header,
verbose=1,
shuffle=False
)

print(' ')
print("-----")
print(' ')

plt.plot(hist_vader_header.history['loss'], label='train_vader_header')
plt.plot(hist_vader_header.history['val_loss'], label='test_vader_header')
plt.legend()
plt.show()
print(' ')
print("-----")
print(' ')
rms_LSTM_vader_header = math.sqrt(min(hist_vader_header.history['val_loss']))
print(' ')
print("-----")
print(' ')
# predicting stock prices
predicted_stock_price_vader_header = model_vader_header.predict(X_test_rolled_vader_header)

predicted_stock_price_vader_header = normalizers_vader_header['OPEN']\
    .inverse_transform(predicted_stock_price_vader_header).reshape(-1, 1)
print(' ')
print("Root mean squared error on valid:", rms_LSTM_vader_header)
print(' ')
print("-----")
print(' ')
print("Root mean squared error on valid inverse transformed from normalization:",
      normalizers_vader_header["OPEN"].inverse_transform(np.array([rms_LSTM_vader_header]).reshape(1, -1)))
print(' ')
print("-----")
print(' ')
print(predicted_stock_price_vader_header)

### analysis with without semantics
new_df_without_semantics = concatenate_dataframe[['OPEN',
                                                  'HIGH',
                                                  'LOW',
                                                  'CLOSE',
                                                  'VOLUME',]]

new_df_without_semantics = new_df_without_semantics.fillna(0)
new_df_without_semantics[['OPEN', 'HIGH', 'LOW', 'CLOSE', 'VOLUME']].astype(np.float64)
print(new_df_without_semantics)

# train, valid, test split
valid_test_size_split_without_semantics = 0.1

X_train_without_semantics, \
X_else_without_semantics, \
y_train_without_semantics, \
y_else_without_semantics = train_test_split(new_df_without_semantics,
                                             new_df_without_semantics['OPEN'],

```

```

        test_size=valid_test_size_split_without
        _semantics*2,
        shuffle=False)

X_valid_without_semantics, \
X_test_without_semantics, \
y_valid_without_semantics, \
y_test_without_semantics = train_test_split(X_else_without_semantics,
                                             y_else_without_semantics,
                                             test_size=0.5,
                                             shuffle=False)

warnings.filterwarnings(action='ignore', category=DataConversionWarning)

# normalize data
def minmax_scale_without_semantics(df_x, series_y, normalizers_without_semantics =
None):
    features_to_minmax = ['OPEN', 'HIGH', 'LOW', 'CLOSE', 'VOLUME']

    if not normalizers_without_semantics:
        normalizers_without_semantics = {}

    for feat in features_to_minmax:
        if feat not in normalizers_without_semantics:
            normalizers_without_semantics[feat] = MinMaxScaler()
            normalizers_without_semantics[feat].fit(df_x[feat].values.reshape(-1,
1))

        df_x[feat] = normalizers_without_semantics[feat].transform(df_x[feat].value
s.reshape(-1, 1))

        series_y = normalizers_without_semantics['OPEN'].transform(series_y.values.res
ape(-1, 1))

    return df_x, series_y, normalizers_without_semantics

X_train_norm_without_semantics, \
y_train_norm_without_semantics, \
normalizers_without_semantics = minmax_scale_without_semantics(X_train_without_sema
ntics,
                                                                y_train_without_sema
ntics
                                                                )

X_valid_norm_without_semantics, \
y_valid_norm_without_semantics, \
_ = minmax_scale_without_semantics(X_valid_without_semantics,
                                   y_valid_without_semantics,
                                   normalizers_without_semantics=normalizers_withou
t_semantics
                                   )

X_test_norm_without_semantics, \
y_test_norm_without_semantics, \
_ = minmax_scale_without_semantics(X_test_without_semantics,
                                   y_test_without_semantics,
                                   normalizers_without_semantics=normalizers_withou
t_semantics
                                   )

# Creating target (y) and "windows" (X) for modeling
TIME_WINDOW_without_semantics = 45
FORECAST_DISTANCE_without_semantics = 9

```

```

segmenter_without_semantics = SegmentXYForecast(width=TIME_WINDOW_without_semantics,
step=1,
y_func=last,
forecast=FORECAST_DISTANCE_without_semantics,
)

X_train_rolled_without_semantics, \
y_train_rolled_without_semantics, \
_ = segmenter_without_semantics.fit_transform([X_train_norm_without_semantics.values],
[y_train_norm_without_semantics.flatten()])

X_valid_rolled_without_semantics, \
y_valid_rolled_without_semantics, \
_ = segmenter_without_semantics.fit_transform([X_valid_norm_without_semantics.values],
[y_valid_norm_without_semantics.flatten()])

X_test_rolled_without_semantics, \
y_test_rolled_without_semantics, \
_ = segmenter_without_semantics.fit_transform([X_test_norm_without_semantics.values],
[y_test_norm_without_semantics.flatten()])

# LSTM Model
first_lstm_size_without_semantics = 75
second_lstm_size_without_semantics = 40
dropout_without_semantics = 0.1
EPOCHS_without_semantics = 50
BATCH_SIZE_without_semantics = 32
column_count_without_semantics = len(X_train_norm_without_semantics.columns)
# model with use of Functional API of Keras
# input layer
input_layer_without_semantics = Input(shape=(TIME_WINDOW_without_semantics, column_count_without_semantics))
# first LSTM layer
first_lstm_without_semantics = LSTM(first_lstm_size_without_semantics,
return_sequences=True,
dropout=dropout_without_semantics)(input_layer_without_semantics)
# second LSTM layer
second_lstm_without_semantics = LSTM(second_lstm_size_without_semantics,
return_sequences=False,
dropout=dropout_without_semantics)(first_lstm_without_semantics)
# output layer
output_layer_without_semantics = Dense(1)(second_lstm_without_semantics)
# creating Model
model_without_semantics = Model(inputs=input_layer_without_semantics, outputs=output_layer_without_semantics)
# compile model
model_without_semantics.compile(optimizer='adam', loss='mean_absolute_error')
# model summary
model_without_semantics.summary()
print(' ')

```

```

print("-----")
print(' ')
# fitting model
hist_without_semantics = model_without_semantics.fit(x=X_train_rolled_without_semantics,
                                                    y=y_train_rolled_without_semantics,
                                                    batch_size=BATCH_SIZE_without_semantics,
                                                    validation_data=(X_valid_rolled_without_semantics,
                                                                    y_valid_rolled_without_semantics),
                                                    epochs=EPOCHS_without_semantics,
                                                    verbose=1,
                                                    shuffle=False
                                                    )

print(' ')
print("-----")
print(' ')

plt.plot(hist_without_semantics.history['loss'], label='train_without_semantics')
plt.plot(hist_without_semantics.history['val_loss'], label='test_without_semantics')
plt.legend()
plt.show()
print(' ')
print("-----")
print(' ')
rms_LSTM_without_semantics = math.sqrt(min(hist_without_semantics.history['val_loss']))
print(' ')
print("-----")
print(' ')
# predicting stock prices
predicted_stock_price_without_semantics = model_without_semantics.predict(X_test_rolled_without_semantics)

predicted_stock_price_without_semantics = normalizers_without_semantics['OPEN'].inverse_transform(predicted_stock_price_without_semantics).reshape(-1, 1)
print(' ')
print("Root mean squared error on valid:", rms_LSTM_without_semantics)
print(' ')
print("-----")
print(' ')
print("Root mean squared error on valid inverse transformed from normalization:",
      normalizers_without_semantics["OPEN"].inverse_transform(np.array([rms_LSTM_without_semantics])).reshape(1, -1))
print(' ')
print("-----")
print(' ')
print(predicted_stock_price_without_semantics)

plt.figure(figsize=(10,5))
#plt.plot(X_test, color='black', label='BMW Stock Price')
plt.plot(predicted_stock_price_flair_content, color='green', label='Predicted BMW Stock Price with flair content analysis')
plt.plot(predicted_stock_price_flair_header, color='red', label='Predicted BMW Stock Price with flair header analysis')
plt.plot(predicted_stock_price_textblob_header, color='yellow', label='Predicted BMW Stock Price with textblob header analysis')

```

```
plt.plot(predicted_stock_price_textblob_content, color='blue', label='Predicted BMW  
Stock Price with textblob content analysis')  
plt.plot(predicted_stock_price_vader_content, color='cyan', label='Predicted BMW St  
ock Price with vader content analysis')  
plt.plot(predicted_stock_price_vader_header, color='magenta', label='Predicted BMW  
Stock Price with vader header analysis')  
plt.plot(predicted_stock_price_without_semantics, color='orange', label='Predicted  
BMW Stock Price without semantics analysis')  
#plt.rcParams['figure.facecolor'] = 'salmon'  
plt.title('BMW Stock Price Prediction')  
plt.xlabel('Time')  
plt.ylabel('BMW Stock Price')  
plt.legend(loc='upper center', bbox_to_anchor=(0.5, -0.005), borderaxespad=8)  
date_today = str(datetime.now().strftime("%Y%m%d"))  
plt.savefig(r'C:\Users\victo\Master_Thesis\stockprice_prediction\LSTM\bmw\hourly\pr  
ediction_bmw_with_all_' + date_today + '.png',  
            bbox_inches="tight",  
            dpi=100,  
            pad_inches=1.5)  
  
plt.show()  
  
print('Run is finished and plot is saved!')
```