In [ ]:
```
###necessary libaries###
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
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
from math import sqrt
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
import xgboost as xgb
from sklearn.model_selection import ParameterSampler, ParameterGrid

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\volkswagen\minutely\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)

new_df_flair_content = concatenate_dataframe[['OPEN',
                                              'HIGH',
                                              'LOW',
                                              'CLOSE',
                                              'VOLUME',
                                              'flair_sentiment_content_score']]

new_df_flair_content = new_df_flair_content.fillna(0)
# new_df[['OPEN', 'HIGH', 'LOW', 'CLOSE', 'VOLUME', 'compound_vader_articel_content
```

```
']].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_conte
nt*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)
#print(y_else)
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_sentimen
t_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.re
shape(-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 = 60
FORECAST_DISTANCE_flair_content = 30

segmenter_flair_content = SegmentXYForecast(width=TIME_WINDOW_flair_content,
                                            step=1,
                                            y_func=last,
                                            forecast=FORECAST_DISTANCE_flair_conten
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()]
                                          )

shape_flair_content = X_train_rolled_flair_content.shape
X_train_flattened_flair_content = X_train_rolled_flair_content.reshape(shape_flair_
content[0],
                                                                       shape_flair_
content[1]*shape_flair_content[2]
                                                                       )

X_train_flattened_flair_content.shape
shape_flair_content = X_valid_rolled_flair_content.shape
X_valid_flattened = X_valid_rolled_flair_content.reshape(shape_flair_content[0],
                                                         shape_flair_content[1]*sha
pe_flair_content[2]
                                                         )

# Random Forest
N_ESTIMATORS_flair_content = 30
RANDOM_STATE_flair_content = 452543634

RF_base_model_flair_content = RandomForestRegressor(random_state=RANDOM_STATE_flair
_content,
                                                    n_estimators=N_ESTIMATORS_flair
_content,
                                                    n_jobs=-1,
                                                    verbose=100
                                                    )

RF_base_model_flair_content.fit(X_train_flattened_flair_content, y_train_rolled_fla
ir_content)
print(' ')
print("-----------------------------------------------------------------")
print(' ')
RF_base_model_predictions_flair_content = RF_base_model_flair_content.predict(X_val
```

```
id_flattened)
print(' ')
print("----------------------------------------------------------------------")
print(' ')
rms_base_flair_content = sqrt(mean_squared_error(y_valid_rolled_flair_content,
                                                 RF_base_model_predictions_flair_co
ntent
                                                 )
                              )

print("Root mean squared error on valid:",rms_base_flair_content)
print("Root mean squared error on valid inverse transformed from normalization:",no
rmalizers_flair_content["OPEN"]
      .inverse_transform(np.array([rms_base_flair_content]).reshape(-1, 1)))

print(' ')
print("----------------------------------------------------------------------")
print(' ')
RF_base_model_predictions_flair_content = normalizers_flair_content['OPEN']\
                                          .inverse_transform(np.array(RF_base_model
_predictions_flair_content).reshape(-1, 1))
print(' ')
print("----------------------------------------------------------------------")
print(' ')

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

new_df_flair_header = concatenate_dataframe[['OPEN',
                                             'HIGH',
                                             'LOW',
                                             'CLOSE',
                                             'VOLUME',
                                             'flair_sentiment_header_score']]

new_df_flair_header = new_df_flair_header.fillna(0)
# new_df[['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)
#print(y_else)
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 = 60
FORECAST_DISTANCE_flair_header = 30

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()]
                                         )

shape_flair_header = X_train_rolled_flair_header.shape
X_train_flattened_flair_header = X_train_rolled_flair_header.reshape(shape_flair_he
ader[0],
                                                                     shape_flair_he
ader[1]*shape_flair_header[2]
                                                                     )

X_train_flattened_flair_header.shape
shape_flair_header = X_valid_rolled_flair_header.shape
X_valid_flattened = X_valid_rolled_flair_header.reshape(shape_flair_header[0],
                                                        shape_flair_header[1]*shape
_flair_header[2]
                                                        )

# Random Forest
N_ESTIMATORS_flair_header = 30
RANDOM_STATE_flair_header = 452543634

RF_base_model_flair_header = RandomForestRegressor(random_state=RANDOM_STATE_flair_
header,
                                                   n_estimators=N_ESTIMATORS_flair_
header,
                                                   n_jobs=-1,
                                                   verbose=100
                                                   )

RF_base_model_flair_header.fit(X_train_flattened_flair_header, y_train_rolled_flair
_header)
print(' ')
print("--------------------------------------------------------------------")
print(' ')
RF_base_model_predictions_flair_header = RF_base_model_flair_header.predict(X_valid
_flattened)
print(' ')
print("--------------------------------------------------------------------")
print(' ')
rms_base_flair_header = sqrt(mean_squared_error(y_valid_rolled_flair_header,
                                                RF_base_model_predictions_flair_hea
der
                                                )
                             )

print("Root mean squared error on valid:",rms_base_flair_header)
print("Root mean squared error on valid inverse transformed from normalization:",no
rmalizers_flair_header["OPEN"]
      .inverse_transform(np.array([rms_base_flair_header]).reshape(-1, 1)))

print(' ')
print("--------------------------------------------------------------------")
print(' ')
RF_base_model_predictions_flair_header = normalizers_flair_header['OPEN']\
                                         .inverse_transform(np.array(RF_base_model
_predictions_flair_header).reshape(-1, 1))
print(' ')
print("--------------------------------------------------------------------")
print(' ')

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

```
print(' ')

new_df_textblob_content = concatenate_dataframe[['OPEN',
                                                 'HIGH',
                                                 'LOW',
                                                 'CLOSE',
                                                 'VOLUME',
                                                 'polarity_textblob_sentiment_conte
nt']]

new_df_textblob_content = new_df_textblob_content.fillna(0)
# new_df[['OPEN', 'HIGH', 'LOW', 'CLOSE', 'VOLUME', 'polarity_textblob_sentiment_co
ntent']].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)
#print(y_else)
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 = 60
FORECAST_DISTANCE_textblob_content = 30

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
()]
                                             )

shape_textblob_content = X_train_rolled_textblob_content.shape
X_train_flattened_textblob_content = X_train_rolled_textblob_content.reshape(shape_
textblob_content[0],
                                                                             shape_
textblob_content[1]*shape_textblob_content[2]
                                                                             )

X_train_flattened_textblob_content.shape
shape_textblob_content = X_valid_rolled_textblob_content.shape
```

```
X_valid_flattened = X_valid_rolled_textblob_content.reshape(shape_textblob_content
[0],
                                                            shape_textblob_content
[1]*shape_textblob_content[2]
                                                            )

# Random Forest
N_ESTIMATORS_textblob_content = 30
RANDOM_STATE_textblob_content = 452543634

RF_base_model_textblob_content = RandomForestRegressor(random_state=RANDOM_STATE_te
xtblob_content,
                                                       n_estimators=N_ESTIMATORS_te
xtblob_content,
                                                       n_jobs=-1,
                                                       verbose=100
                                                       )

RF_base_model_textblob_content.fit(X_train_flattened_textblob_content, y_train_roll
ed_textblob_content)
print(' ')
print("-------------------------------------------------------------------")
print(' ')
RF_base_model_predictions_textblob_content = RF_base_model_textblob_content.predict
(X_valid_flattened)
print(' ')
print("-------------------------------------------------------------------")
print(' ')
rms_base_textblob_content = sqrt(mean_squared_error(y_valid_rolled_textblob_conten
t,
                                                    RF_base_model_predictions_textb
lob_content
                                                    )
                                 )

print("Root mean squared error on valid:",rms_base_textblob_content)
print("Root mean squared error on valid inverse transformed from normalization:",no
rmalizers_textblob_content["OPEN"]
      .inverse_transform(np.array([rms_base_textblob_content]).reshape(-1, 1)))

print(' ')
print("-------------------------------------------------------------------")
print(' ')
RF_base_model_predictions_textblob_content = normalizers_textblob_content['OPEN']\
                                             .inverse_transform(np.array(RF_base_model
_predictions_textblob_content).reshape(-1, 1))
print(' ')
print("-------------------------------------------------------------------")
print(' ')

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

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[['OPEN', 'HIGH', 'LOW', 'CLOSE', 'VOLUME', 'polarity_textblob_sentiment_he
```

```
ader']].astype(np.float64)
# print(new_df)

# 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)
#print(y_else)
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 = 60
FORECAST_DISTANCE_textblob_header = 30

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()]
                                            )

shape_textblob_header = X_train_rolled_textblob_header.shape
X_train_flattened_textblob_header = X_train_rolled_textblob_header.reshape(shape_te
xtblob_header[0],
                                                                           shape_te
xtblob_header[1]*shape_textblob_header[2]
                                                                           )

X_train_flattened_textblob_header.shape
shape_textblob_header = X_valid_rolled_textblob_header.shape
X_valid_flattened = X_valid_rolled_textblob_header.reshape(shape_textblob_header
[0],
                                                           shape_textblob_header[1]
*shape_textblob_header[2]
                                                           )

# Random Forest
N_ESTIMATORS_textblob_header = 30
RANDOM_STATE_textblob_header = 452543634

RF_base_model_textblob_header = RandomForestRegressor(random_state=RANDOM_STATE_tex
tblob_header,
                                                      n_estimators=N_ESTIMATORS_tex
tblob_header,
                                                      n_jobs=-1,
                                                      verbose=100
                                                      )
```

```
RF_base_model_textblob_header.fit(X_train_flattened_textblob_header, y_train_rolled
_textblob_header)
print(' ')
print("-----------------------------------------------------------------")
print(' ')
RF_base_model_predictions_textblob_header = RF_base_model_textblob_header.predict(X
_valid_flattened)
print(' ')
print("-----------------------------------------------------------------")
print(' ')
rms_base_textblob_header = sqrt(mean_squared_error(y_valid_rolled_textblob_header,
                                                   RF_base_model_predictions_textbl
ob_header
                                                   )
                                )

print("Root mean squared error on valid:",rms_base_textblob_header)
print("Root mean squared error on valid inverse transformed from normalization:",no
rmalizers_textblob_header["OPEN"]
      .inverse_transform(np.array([rms_base_textblob_header]).reshape(-1, 1)))

print(' ')
print("-----------------------------------------------------------------")
print(' ')
RF_base_model_predictions_textblob_header = normalizers_textblob_header['OPEN']\
                                            .inverse_transform(np.array(RF_base_model
_predictions_textblob_header).reshape(-1, 1))
print(' ')
print("-----------------------------------------------------------------")
print(' ')

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

new_df_vader_content = concatenate_dataframe[['OPEN',
                                              'HIGH',
                                              'LOW',
                                              'CLOSE',
                                              'VOLUME',
                                              'compound_vader_articel_content']]

new_df_vader_content = new_df_vader_content.fillna(0)
# new_df[['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_conte
nt*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)
#print(y_else)
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.re
shape(-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 = 60
FORECAST_DISTANCE_vader_content = 30

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

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()]
                                          )

shape_vader_content = X_train_rolled_vader_content.shape
X_train_flattened_vader_content = X_train_rolled_vader_content.reshape(shape_vader_
content[0],
                                                                       shape_vader_
content[1]*shape_vader_content[2]
                                                                       )

X_train_flattened_vader_content.shape
shape_vader_content = X_valid_rolled_vader_content.shape
X_valid_flattened = X_valid_rolled_vader_content.reshape(shape_vader_content[0],
                                                         shape_vader_content[1]*sha
pe_vader_content[2]
                                                         )

# Random Forest
N_ESTIMATORS_vader_content = 30
RANDOM_STATE_vader_content = 452543634

RF_base_model_vader_content = RandomForestRegressor(random_state=RANDOM_STATE_vader
_content,
                                                    n_estimators=N_ESTIMATORS_vader
_content,
                                                    n_jobs=-1,
                                                    verbose=100
                                                    )

RF_base_model_vader_content.fit(X_train_flattened_vader_content, y_train_rolled_vad
er_content)
print(' ')
print("-----------------------------------------------------------------")
print(' ')
RF_base_model_predictions_vader_content = RF_base_model_vader_content.predict(X_val
id_flattened)
print(' ')
print("-----------------------------------------------------------------")
print(' ')
rms_base_vader_content = sqrt(mean_squared_error(y_valid_rolled_vader_content,
                                                 RF_base_model_predictions_vader_co
ntent
                                                 )
                              )

print("Root mean squared error on valid:",rms_base_vader_content)
print("Root mean squared error on valid inverse transformed from normalization:",no
rmalizers_vader_content["OPEN"]
      .inverse_transform(np.array([rms_base_textblob_content]).reshape(-1, 1)))

print(' ')
print("-----------------------------------------------------------------")
print(' ')
RF_base_model_predictions_vader_content = normalizers_vader_content['OPEN']\
                                          .inverse_transform(np.array(RF_base_model
```

```
_predictions_vader_content).reshape(-1, 1))
print(' ')
print("------------------------------------------------------------------")
print(' ')

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

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[['OPEN', 'HIGH', 'LOW', 'CLOSE', 'VOLUME', 'compound_vader_header']].astyp
e(np.float64)
# print(new_df)

# 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)
#print(y_else)
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.res
hape(-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 = 60
FORECAST_DISTANCE_vader_header = 30

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()]
                                         )

shape_vader_header = X_train_rolled_vader_header.shape
X_train_flattened_vader_header = X_train_rolled_vader_header.reshape(shape_vader_he
ader[0],
                                                                     shape_vader_he
ader[1]*shape_vader_header[2]
                                                                     )

X_train_flattened_vader_header.shape
shape_vader_header = X_valid_rolled_vader_header.shape
X_valid_flattened = X_valid_rolled_vader_header.reshape(shape_vader_header[0],
                                                        shape_vader_header[1]*shape
_vader_header[2]
                                                        )

# Random Forest
```

```
N_ESTIMATORS_vader_header = 30
RANDOM_STATE_vader_header = 452543634

RF_base_model_vader_header = RandomForestRegressor(random_state=RANDOM_STATE_vader_
header,
                                                   n_estimators=N_ESTIMATORS_vader_
header,
                                                   n_jobs=-1,
                                                   verbose=100
                                                   )

RF_base_model_vader_header.fit(X_train_flattened_vader_header, y_train_rolled_vader
_header)
print(' ')
print("-----------------------------------------------------------------")
print(' ')
RF_base_model_predictions_vader_header = RF_base_model_vader_header.predict(X_valid
_flattened)
print(' ')
print("-----------------------------------------------------------------")
print(' ')
rms_base_vader_header = sqrt(mean_squared_error(y_valid_rolled_vader_header,
                                                RF_base_model_predictions_vader_hea
der
                                                )
                             )

print("Root mean squared error on valid:", rms_base_vader_header)
print("Root mean squared error on valid inverse transformed from normalization:", n
ormalizers_vader_header["OPEN"]
      .inverse_transform(np.array([rms_base_vader_header]).reshape(-1, 1)))

print(' ')
print("-----------------------------------------------------------------")
print(' ')
RF_base_model_predictions_vader_header = normalizers_vader_header['OPEN']\
                                         .inverse_transform(np.array(RF_base_model
_predictions_vader_header).reshape(-1, 1))
print(' ')
print("-----------------------------------------------------------------")
print(' ')

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

new_df_without_semantics = concatenate_dataframe[['OPEN',
                                                  'HIGH',
                                                  'LOW',
                                                  'CLOSE',
                                                  'VOLUME']]


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

# 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)
#print(y_else)
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.resh
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 = 60
FORECAST_DISTANCE_without_semantics = 30
```

```
segmenter_without_semantics = SegmentXYForecast(width=TIME_WINDOW_without_semantic
s,
                                                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.value
s],
                                              [y_train_norm_without_semantics.flatt
en()]
                                              )

X_valid_rolled_without_semantics, \
y_valid_rolled_without_semantics, \
_ = segmenter_without_semantics.fit_transform([X_valid_norm_without_semantics.value
s],
                                              [y_valid_norm_without_semantics.flatt
en()]
                                              )

X_test_rolled_without_semantics, \
y_test_rolled_without_semantics, \
_ = segmenter_without_semantics.fit_transform([X_test_norm_without_semantics.value
s],
                                              [y_test_norm_without_semantics.flatte
n()]
                                              )

shape_without_semantics = X_train_rolled_without_semantics.shape
X_train_flattened_without_semantics = X_train_rolled_without_semantics.reshape(shap
e_without_semantics[0],
                                                                               shap
e_without_semantics[1]*shape_without_semantics[2]
                                                                               )

X_train_flattened_without_semantics.shape
shape_without_semantics = X_valid_rolled_without_semantics.shape
X_valid_flattened = X_valid_rolled_without_semantics.reshape(shape_without_semantic
s[0],
                                                             shape_without_semantic
s[1]*shape_without_semantics[2]
                                                             )

# Random Forest
N_ESTIMATORS_without_semantics = 30
RANDOM_STATE_without_semantics = 452543634

RF_base_model_without_semantics = RandomForestRegressor(random_state=RANDOM_STATE_w
ithout_semantics,
                                                        n_estimators=N_ESTIMATORS_w
ithout_semantics,
                                                        n_jobs=-1,
                                                        verbose=100
                                                        )

RF_base_model_without_semantics.fit(X_train_flattened_without_semantics, y_train_ro
lled_without_semantics)
print(' ')
print("-----------------------------------------------------------------")
```

```
print(' ')
RF_base_model_predictions_without_semantics = RF_base_model_without_semantics.predi
ct(X_valid_flattened)
print(' ')
print("-----------------------------------------------------------------")
print(' ')
rms_base_without_semantics = sqrt(mean_squared_error(y_valid_rolled_without_semanti
cs,
                                                     RF_base_model_predictions_with
out_semantics
                                                     )
                                  )

print("Root mean squared error on valid:", rms_base_without_semantics)
print("Root mean squared error on valid inverse transformed from normalization:", n
ormalizers_without_semantics["OPEN"]
      .inverse_transform(np.array([rms_base_without_semantics]).reshape(-1, 1)))

print(' ')
print("-----------------------------------------------------------------")
print(' ')
RF_base_model_predictions_without_semantics = normalizers_without_semantics['OPEN
']\
                                            .inverse_transform(np.array(RF_base_model
_predictions_without_semantics).reshape(-1, 1))
print(' ')
print("-----------------------------------------------------------------")
print(' ')

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

plt.figure(figsize=(10,5))
plt.plot(RF_base_model_predictions_flair_content, color='green', label='Predicted V
olkswagen Stock Price with flair content analysis')
plt.plot(RF_base_model_predictions_flair_header, color='red', label='Predicted Volk
swagen Stock Price with flair header analysis')
plt.plot(RF_base_model_predictions_textblob_content, color='orange', label='Predict
ed Volkswagen Stock Price with textblob content analysis')
plt.plot(RF_base_model_predictions_textblob_header, color='blue', label='Predicted
Volkswagen Stock Price with textblob header analysis')
plt.plot(RF_base_model_predictions_vader_content, color='cyan', label='Predicted Vo
lkswagen Stock Price with vader content analysis')
plt.plot(RF_base_model_predictions_vader_header, color='magenta', label='Predicted
Volkswagen Stock Price with vader header analysis')
plt.plot(RF_base_model_predictions_without_semantics, color='yellow', label='Predic
ted Volkswagen Stock Price without semantics analysis')
plt.title('Volkswagen Stock Price Prediction')
plt.xlabel('Time')
plt.ylabel('Volkswagen 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\RandomForest_base_
model\volkswagen\minutely\prediction_volkswagen_with_semantics_' + date_today + '.p
ng',
            bbox_inches="tight",
            dpi=100,
            pad_inches=1.5)
plt.show()
print('Run is finished and plot is saved!')
```