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\porsche\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,
                                  )

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

new_df_flair_content = new_df_flair_content.fillna(0)
# new_df_flair_content[['Date',
#                        'OPEN',
```

```
#                        'HIGH',
#                        'LOW',
#                        'CLOSE',
#                        'VOLUME',
#                        'flair_sentiment_content_score']].astype(np.float64)

new_df_flair_content['Year'] = pd.DatetimeIndex(new_df_flair_content['Date']).year
new_df_flair_content['Month'] = pd.DatetimeIndex(new_df_flair_content['Date']).mont
h
new_df_flair_content['Day'] = pd.DatetimeIndex(new_df_flair_content['Date']).day
new_df_flair_content['Hour'] = pd.DatetimeIndex(new_df_flair_content['Date']).hour
new_df_flair_content['Minute'] = pd.DatetimeIndex(new_df_flair_content['Date']).min
ute
new_df_flair_content['Second'] = pd.DatetimeIndex(new_df_flair_content['Date']).sec
ond

new_df_flair_content = new_df_flair_content.drop(['Date'], axis=1)

# 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)

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


# normalize data
def minmax_scale_flair_content(df_x, series_y, normalizers_flair_content = None):
    features_to_minmax = ['Year',
                          'Month',
                          'Day',
                          'Hour',
                          'Minute',
                          'Second',
                          '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.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
                               )

def encode_cyclicals_flair_content(df_x):
    # "month","day","hour", "cdbw", "dayofweek"

    #DIRECTIONS = {"N": 1.0, "NE": 2.0, "E": 3.0, "SE": 4.0, "S": 5.0, "SW": 6.0, "
W": 7.0, "NW": 8.0, "cv": np.nan}

    df_x['month_sin'] = np.sin(2 * np.pi * df_x.Month / 12)
    df_x['month_cos'] = np.cos(2 * np.pi * df_x.Month / 12)
    df_x.drop('Month', axis=1, inplace=True)

    df_x['day_sin'] = np.sin(2 * np.pi * df_x.Day / 31)
    df_x['day_cos'] = np.cos(2 * np.pi * df_x.Day / 31)
    df_x.drop('Day', axis=1, inplace=True)

    df_x['hour_sin'] = np.sin(2 * np.pi * df_x.Hour / 24)
    df_x['hour_cos'] = np.cos(2 * np.pi * df_x.Hour / 24)
    df_x.drop('Hour', axis=1, inplace=True)

    df_x['min_sin'] = np.sin(2 * np.pi * df_x.Minute / 60)
    df_x['min_cos'] = np.cos(2 * np.pi * df_x.Minute / 60)
    df_x.drop('Minute', axis=1, inplace=True)

    df_x['sec_sin'] = np.sin(2 * np.pi * df_x.Second / 60)
    df_x['sec_cos'] = np.cos(2 * np.pi * df_x.Second / 60)
    df_x.drop('Second', axis=1, inplace=True)


    return df_x

X_train_norm_flair_content = encode_cyclicals_flair_content(X_train_norm_flair_cont
ent)
X_valid_norm_flair_content = encode_cyclicals_flair_content(X_valid_norm_flair_cont
ent)
X_test_norm_flair_content = encode_cyclicals_flair_content(X_test_norm_flair_conten
t)
```

```
# 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_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_flair_content = X_valid_rolled_flair_content.reshape(shape_flair_
content[0],
                                                                       shape_flair_
content[1]*shape_flair_content[2]
                                                                       )

# XGBoost needs it's custom data format to run quickly
dmatrix_train_flair_content = xgb.DMatrix(data=X_train_flattened_flair_content,
                                          label=y_train_rolled_flair_content
                                          )

dmatrix_valid_flair_content = xgb.DMatrix(data=X_valid_flattened_flair_content,
                                          label=y_valid_rolled_flair_content
                                          )

params_flair_content = {'objective': 'reg:squarederror', 'eval_metric': 'rmse', 'n_
estimators': 30, 'tree_method':'gpu_hist'}
#param['nthread'] = 4
evallist_flair_content = [(dmatrix_valid_flair_content, 'eval'), (dmatrix_train_fla
ir_content, 'train')]

#After some tests, it turned out to overfit after this point
num_round_flair_content = 12

xg_reg_flair_content = xgb.train(params_flair_content,
                                 dmatrix_train_flair_content,
```

```
                                 num_round_flair_content,
                                 evallist_flair_content
                                 )

xgb_predictions_flair_content = xg_reg_flair_content.predict(dmatrix_valid_flair_co
ntent)

rms_base_flair_content = sqrt(mean_squared_error(y_valid_rolled_flair_content, xgb_
predictions_flair_content))

print("Root mean squared error on valid:",rms_base_flair_content)
print("Root mean squared error on valid inverse transformed from normalization:",
      normalizers_flair_content["OPEN"].inverse_transform(np.array([rms_base_flair_
content]).reshape(1, -1)))

all_params_flair_content = {
    # 'min_child_weight': [1, 5, 10],
    # 'gamma': [0.5, 1, 1.5, 2, 5],
    # 'subsample': [0.6, 0.8, 1.0],
    # 'colsample_bytree': [0.6, 0.8, 1.0],
    # 'max_depth': [3, 4, 5],
    'n_estimators': [30, 100, 200, 500],
    'learning_rate': [0.01, 0.1, 0.2, 0.3],
    'objective': ['reg:squarederror'],
    'eval_metric': ['rmse'],
    'tree_method': ['gpu_hist'],
}

best_score_flair_content = 10000.0
run_flair_content = 1

evallist_flair_content = [(dmatrix_valid_flair_content, 'eval'), (dmatrix_train_fla
ir_content, 'train')]
for param_sample_flair_content in ParameterGrid(all_params_flair_content):
    print("----RUN ", run_flair_content)
    xg_reg_flair_content = xgb.train(param_sample_flair_content,
                                     dmatrix_train_flair_content,
                                     num_round_flair_content * 3,
                                     evallist_flair_content)

    xgb_predictions_flair_content = xg_reg_flair_content.predict(dmatrix_valid_flai
r_content)
    score_flair_content = sqrt(mean_squared_error(y_valid_rolled_flair_content, xgb
_predictions_flair_content))

    if score_flair_content < best_score_flair_content:
        best_score_flair_content = score_flair_content
        best_model_flair_content = xg_reg_flair_content
    run_flair_content += 1

print("Root mean squared error on valid:", best_score_flair_content)
print("Root mean squared error on valid inverse transformed from normalization:",
      normalizers_flair_content["OPEN"].inverse_transform(np.array([best_score_flai
r_content]).reshape(1, -1)))

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

xgboost_price_prediction_flair_content = normalizers_flair_content['OPEN'].inverse_
transform(np.array(xgb_predictions_flair_content).reshape(-1, 1))

print(xgboost_price_prediction_flair_content)

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

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

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

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

new_df_flair_header['Year'] = pd.DatetimeIndex(new_df_flair_header['Date']).year
new_df_flair_header['Month'] = pd.DatetimeIndex(new_df_flair_header['Date']).month
new_df_flair_header['Day'] = pd.DatetimeIndex(new_df_flair_header['Date']).day
new_df_flair_header['Hour'] = pd.DatetimeIndex(new_df_flair_header['Date']).hour
new_df_flair_header['Minute'] = pd.DatetimeIndex(new_df_flair_header['Date']).minut
e
new_df_flair_header['Second'] = pd.DatetimeIndex(new_df_flair_header['Date']).secon
d

new_df_flair_header = new_df_flair_header.drop(['Date'], axis=1)

# 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 = ['Year',
                          'Month',
                          'Day',
                          'Hour',
                          'Minute',
                          'Second',
                          'OPEN',
                          'HIGH',
```

```
                           'LOW',
                           'CLOSE',
                           'VOLUME',
                           'flair_sentiment_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
                              )

def encode_cyclicals_flair_header(df_x):
    # "month","day","hour", "cdbw", "dayofweek"

    #DIRECTIONS = {"N": 1.0, "NE": 2.0, "E": 3.0, "SE": 4.0, "S": 5.0, "SW": 6.0, "
W": 7.0, "NW": 8.0, "cv": np.nan}

    df_x['month_sin'] = np.sin(2 * np.pi * df_x.Month / 12)
    df_x['month_cos'] = np.cos(2 * np.pi * df_x.Month / 12)
    df_x.drop('Month', axis=1, inplace=True)

    df_x['day_sin'] = np.sin(2 * np.pi * df_x.Day / 31)
    df_x['day_cos'] = np.cos(2 * np.pi * df_x.Day / 31)
    df_x.drop('Day', axis=1, inplace=True)

    df_x['hour_sin'] = np.sin(2 * np.pi * df_x.Hour / 24)
    df_x['hour_cos'] = np.cos(2 * np.pi * df_x.Hour / 24)
    df_x.drop('Hour', axis=1, inplace=True)

    df_x['min_sin'] = np.sin(2 * np.pi * df_x.Minute / 60)
    df_x['min_cos'] = np.cos(2 * np.pi * df_x.Minute / 60)
    df_x.drop('Minute', axis=1, inplace=True)

    df_x['sec_sin'] = np.sin(2 * np.pi * df_x.Second / 60)
```

```
    df_x['sec_cos'] = np.cos(2 * np.pi * df_x.Second / 60)
    df_x.drop('Second', axis=1, inplace=True)


    return df_x

X_train_norm_flair_header = encode_cyclicals_flair_header(X_train_norm_flair_heade
r)
X_valid_norm_flair_header = encode_cyclicals_flair_header(X_valid_norm_flair_heade
r)
X_test_norm_flair_header = encode_cyclicals_flair_header(X_test_norm_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()]
                                         )

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_flair_header = X_valid_rolled_flair_header.reshape(shape_flair_he
ader[0],
                                                                     shape_flair_he
ader[1]*shape_flair_header[2]
                                                                     )

# XGBoost needs it's custom data format to run quickly
dmatrix_train_flair_header = xgb.DMatrix(data=X_train_flattened_flair_header,
                                         label=y_train_rolled_flair_header
                                         )

dmatrix_valid_flair_header = xgb.DMatrix(data=X_valid_flattened_flair_header,
                                         label=y_valid_rolled_flair_header
                                         )
```

```
params_flair_header = {'objective': 'reg:squarederror', 'eval_metric': 'rmse', 'n_e
stimators': 30, 'tree_method':'gpu_hist'}
#param['nthread'] = 4
evallist_flair_header = [(dmatrix_valid_flair_header, 'eval'), (dmatrix_train_flair
_header, 'train')]

#After some tests, it turned out to overfit after this point
num_round_flair_header = 12

xg_reg_flair_header = xgb.train(params_flair_header,
                                dmatrix_train_flair_header,
                                num_round_flair_header,
                                evallist_flair_header
                                )

xgb_predictions_flair_header = xg_reg_flair_header.predict(dmatrix_valid_flair_head
er)

rms_base_flair_header = sqrt(mean_squared_error(y_valid_rolled_flair_header, xgb_pr
edictions_flair_header))

print("Root mean squared error on valid:",rms_base_flair_header)
print("Root mean squared error on valid inverse transformed from normalization:",
      normalizers_flair_header["OPEN"].inverse_transform(np.array([rms_base_flair_h
eader]).reshape(1, -1)))

all_params_flair_header = {
    # 'min_child_weight': [1, 5, 10],
    # 'gamma': [0.5, 1, 1.5, 2, 5],
    # 'subsample': [0.6, 0.8, 1.0],
    # 'colsample_bytree': [0.6, 0.8, 1.0],
    # 'max_depth': [3, 4, 5],
    'n_estimators': [30, 100, 200, 500],
    'learning_rate': [0.01, 0.1, 0.2, 0.3],
    'objective': ['reg:squarederror'],
    'eval_metric': ['rmse'],
    'tree_method': ['gpu_hist'],
}

best_score_flair_header = 10000.0
run_flair_header = 1

evallist_flair_header = [(dmatrix_valid_flair_header, 'eval'), (dmatrix_train_flair
_header, 'train')]
for param_sample_flair_header in ParameterGrid(all_params_flair_header):
    print("----RUN ", run_flair_header)
    xg_reg_flair_header = xgb.train(param_sample_flair_header,
                                    dmatrix_train_flair_header,
                                    num_round_flair_header * 3,
                                    evallist_flair_header)

    xgb_predictions_flair_header = xg_reg_flair_header.predict(dmatrix_valid_flair_
header)
    score_flair_header = sqrt(mean_squared_error(y_valid_rolled_flair_header, xgb_p
redictions_flair_header))

    if score_flair_header < best_score_flair_header:
        best_score_flair_header = score_flair_header
        best_model_flair_header = xg_reg_flair_header
    run_flair_header += 1

print("Root mean squared error on valid:", best_score_flair_header)
print("Root mean squared error on valid inverse transformed from normalization:",
      normalizers_flair_header["OPEN"].inverse_transform(np.array([best_score_flair
```

```
_header]).reshape(1, -1)))

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

xgboost_price_prediction_flair_header = normalizers_flair_header['OPEN'].inverse_tr
ansform(np.array(xgb_predictions_flair_header).reshape(-1, 1))

print(xgboost_price_prediction_flair_header)

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

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

new_df_textblob_content = new_df_textblob_content.fillna(0)
# new_df_textblob_content[['Date',
#                            'OPEN',
#                            'HIGH',
#                            'LOW',
#                            'CLOSE',
#                            'VOLUME',
#                            'polarity_textblob_sentiment_content']].astype(np.float6
4)

new_df_textblob_content['Year'] = pd.DatetimeIndex(new_df_textblob_content['Date
']).year
new_df_textblob_content['Month'] = pd.DatetimeIndex(new_df_textblob_content['Date
']).month
new_df_textblob_content['Day'] = pd.DatetimeIndex(new_df_textblob_content['Date']).
day
new_df_textblob_content['Hour'] = pd.DatetimeIndex(new_df_textblob_content['Date
']).hour
new_df_textblob_content['Minute'] = pd.DatetimeIndex(new_df_textblob_content['Date
']).minute
new_df_textblob_content['Second'] = pd.DatetimeIndex(new_df_textblob_content['Date
']).second

new_df_textblob_content = new_df_textblob_content.drop(['Date'], axis=1)

# 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 = ['Year',
                          'Month',
                          'Day',
                          'Hour',
                          'Minute',
                          'Second',
                          'OPEN',
                          'HIGH',
                          'LOW',
                          'CLOSE',
                          'VOLUME',
                          'polarity_textblob_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
                                  )
```

```
def encode_cyclicals_textblob_content(df_x):
    # "month","day","hour", "cdbw", "dayofweek"

    #DIRECTIONS = {"N": 1.0, "NE": 2.0, "E": 3.0, "SE": 4.0, "S": 5.0, "SW": 6.0, "
W": 7.0, "NW": 8.0, "cv": np.nan}

    df_x['month_sin'] = np.sin(2 * np.pi * df_x.Month / 12)
    df_x['month_cos'] = np.cos(2 * np.pi * df_x.Month / 12)
    df_x.drop('Month', axis=1, inplace=True)

    df_x['day_sin'] = np.sin(2 * np.pi * df_x.Day / 31)
    df_x['day_cos'] = np.cos(2 * np.pi * df_x.Day / 31)
    df_x.drop('Day', axis=1, inplace=True)

    df_x['hour_sin'] = np.sin(2 * np.pi * df_x.Hour / 24)
    df_x['hour_cos'] = np.cos(2 * np.pi * df_x.Hour / 24)
    df_x.drop('Hour', axis=1, inplace=True)

    df_x['min_sin'] = np.sin(2 * np.pi * df_x.Minute / 60)
    df_x['min_cos'] = np.cos(2 * np.pi * df_x.Minute / 60)
    df_x.drop('Minute', axis=1, inplace=True)

    df_x['sec_sin'] = np.sin(2 * np.pi * df_x.Second / 60)
    df_x['sec_cos'] = np.cos(2 * np.pi * df_x.Second / 60)
    df_x.drop('Second', axis=1, inplace=True)


    return df_x

X_train_norm_textblob_content = encode_cyclicals_textblob_content(X_train_norm_text
blob_content)
X_valid_norm_textblob_content = encode_cyclicals_textblob_content(X_valid_norm_text
blob_content)
X_test_norm_textblob_content = encode_cyclicals_textblob_content(X_test_norm_textbl
ob_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
()]
                                             )

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_textblob_content = X_valid_rolled_textblob_content.reshape(shape_
textblob_content[0],
                                                                             shape_
textblob_content[1]*shape_textblob_content[2]
                                                                             )

# XGBoost needs it's custom data format to run quickly
dmatrix_train_textblob_content = xgb.DMatrix(data=X_train_flattened_textblob_conten
t,
                                             label=y_train_rolled_textblob_content
                                             )

dmatrix_valid_textblob_content = xgb.DMatrix(data=X_valid_flattened_textblob_conten
t,
                                             label=y_valid_rolled_textblob_content
                                             )

params_textblob_content = {'objective': 'reg:squarederror', 'eval_metric': 'rmse',
'n_estimators': 30, 'tree_method':'gpu_hist'}
#param['nthread'] = 4
evallist_textblob_content = [(dmatrix_valid_textblob_content, 'eval'), (dmatrix_tra
in_textblob_content, 'train')]

#After some tests, it turned out to overfit after this point
num_round_textblob_content = 12

xg_reg_textblob_content = xgb.train(params_textblob_content,
                                    dmatrix_train_textblob_content,
                                    num_round_textblob_content,
                                    evallist_textblob_content
                                    )

xgb_predictions_textblob_content = xg_reg_textblob_content.predict(dmatrix_valid_te
xtblob_content)

rms_base_textblob_content = sqrt(mean_squared_error(y_valid_rolled_textblob_conten
t, xgb_predictions_textblob_content))

print("Root mean squared error on valid:",rms_base_textblob_content)
print("Root mean squared error on valid inverse transformed from normalization:",
      normalizers_textblob_content["OPEN"].inverse_transform(np.array([rms_base_tex
tblob_content]).reshape(1, -1)))

all_params_textblob_content = {
    # 'min_child_weight': [1, 5, 10],
    # 'gamma': [0.5, 1, 1.5, 2, 5],
    # 'subsample': [0.6, 0.8, 1.0],
    # 'colsample_bytree': [0.6, 0.8, 1.0],
    # 'max_depth': [3, 4, 5],
```

```
    'n_estimators': [30, 100, 200, 500],
    'learning_rate': [0.01, 0.1, 0.2, 0.3],
    'objective': ['reg:squarederror'],
    'eval_metric': ['rmse'],
    'tree_method': ['gpu_hist'],
}

best_score_textblob_content = 10000.0
run_textblob_content = 1

evallist_textblob_content = [(dmatrix_valid_textblob_content, 'eval'), (dmatrix_tra
in_textblob_content, 'train')]
for param_sample_textblob_content in ParameterGrid(all_params_textblob_content):
    print("----RUN ", run_textblob_content)
    xg_reg_textblob_content = xgb.train(param_sample_textblob_content,
                                        dmatrix_train_textblob_content,
                                        num_round_textblob_content * 3,
                                        evallist_textblob_content
                                        )

    xgb_predictions_textblob_content = xg_reg_textblob_content.predict(dmatrix_vali
d_textblob_content)
    score_textblob_content = sqrt(mean_squared_error(y_valid_rolled_textblob_conten
t, xgb_predictions_textblob_content))

    if score_textblob_content < best_score_textblob_content:
        best_score_textblob_content = score_textblob_content
        best_model_textblob_content = xg_reg_textblob_content
    run_textblob_content += 1

print("Root mean squared error on valid:", best_score_textblob_content)
print("Root mean squared error on valid inverse transformed from normalization:",
      normalizers_textblob_content["OPEN"].inverse_transform(np.array([best_score_t
extblob_content]).reshape(1, -1)))

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

xgboost_price_prediction_textblob_content = normalizers_textblob_content['OPEN'].in
verse_transform(np.array(xgb_predictions_textblob_content).reshape(-1, 1))

print(xgboost_price_prediction_textblob_content)

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

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

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

```
new_df_textblob_header['Year'] = pd.DatetimeIndex(new_df_textblob_header['Date']).y
ear
new_df_textblob_header['Month'] = pd.DatetimeIndex(new_df_textblob_header['Date']).
month
new_df_textblob_header['Day'] = pd.DatetimeIndex(new_df_textblob_header['Date']).da
y
new_df_textblob_header['Hour'] = pd.DatetimeIndex(new_df_textblob_header['Date']).h
our
new_df_textblob_header['Minute'] = pd.DatetimeIndex(new_df_textblob_header['Date
']).minute
new_df_textblob_header['Second'] = pd.DatetimeIndex(new_df_textblob_header['Date
']).second

new_df_textblob_header = new_df_textblob_header.drop(['Date'], axis=1)

# 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 = ['Year',
                          'Month',
                          'Day',
                          'Hour',
                          'Minute',
                          'Second',
                          'OPEN',
                          'HIGH',
                          'LOW',
                          'CLOSE',
                          'VOLUME',
                          'polarity_textblob_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
                                 )

def encode_cyclicals_textblob_header(df_x):
    # "month","day","hour", "cdbw", "dayofweek"

    #DIRECTIONS = {"N": 1.0, "NE": 2.0, "E": 3.0, "SE": 4.0, "S": 5.0, "SW": 6.0, "
W": 7.0, "NW": 8.0, "cv": np.nan}

    df_x['month_sin'] = np.sin(2 * np.pi * df_x.Month / 12)
    df_x['month_cos'] = np.cos(2 * np.pi * df_x.Month / 12)
    df_x.drop('Month', axis=1, inplace=True)

    df_x['day_sin'] = np.sin(2 * np.pi * df_x.Day / 31)
    df_x['day_cos'] = np.cos(2 * np.pi * df_x.Day / 31)
    df_x.drop('Day', axis=1, inplace=True)

    df_x['hour_sin'] = np.sin(2 * np.pi * df_x.Hour / 24)
    df_x['hour_cos'] = np.cos(2 * np.pi * df_x.Hour / 24)
    df_x.drop('Hour', axis=1, inplace=True)

    df_x['min_sin'] = np.sin(2 * np.pi * df_x.Minute / 60)
    df_x['min_cos'] = np.cos(2 * np.pi * df_x.Minute / 60)
    df_x.drop('Minute', axis=1, inplace=True)

    df_x['sec_sin'] = np.sin(2 * np.pi * df_x.Second / 60)
    df_x['sec_cos'] = np.cos(2 * np.pi * df_x.Second / 60)
    df_x.drop('Second', axis=1, inplace=True)


    return df_x

X_train_norm_textblob_header = encode_cyclicals_textblob_header(X_train_norm_textbl
ob_header)
X_valid_norm_textblob_header = encode_cyclicals_textblob_header(X_valid_norm_textbl
ob_header)
X_test_norm_textblob_header = encode_cyclicals_textblob_header(X_test_norm_textblob
_header)
```

```
# 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()]
                                            )

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_textblob_header = X_valid_rolled_textblob_header.reshape(shape_te
xtblob_header[0],
                                                                           shape_te
xtblob_header[1]*shape_textblob_header[2]
                                                                           )

# XGBoost needs it's custom data format to run quickly
dmatrix_train_textblob_header = xgb.DMatrix(data=X_train_flattened_textblob_header,
                                            label=y_train_rolled_textblob_header
                                            )

dmatrix_valid_textblob_header = xgb.DMatrix(data=X_valid_flattened_textblob_header,
                                            label=y_valid_rolled_textblob_header
                                            )

params_textblob_header = {'objective': 'reg:squarederror', 'eval_metric': 'rmse', '
n_estimators': 30, 'tree_method':'gpu_hist'}
#param['nthread'] = 4
evallist_textblob_header = [(dmatrix_valid_textblob_header, 'eval'), (dmatrix_train
_textblob_header, 'train')]

#After some tests, it turned out to overfit after this point
num_round_textblob_header = 12
```

```
xg_reg_textblob_header = xgb.train(params_textblob_header,
                                   dmatrix_train_textblob_header,
                                   num_round_textblob_header,
                                   evallist_textblob_header
                                   )

xgb_predictions_textblob_header = xg_reg_textblob_header.predict(dmatrix_valid_text
blob_header)

rms_base_textblob_header = sqrt(mean_squared_error(y_valid_rolled_textblob_header,
xgb_predictions_textblob_header))

print("Root mean squared error on valid:",rms_base_textblob_header)
print("Root mean squared error on valid inverse transformed from normalization:",
      normalizers_textblob_header["OPEN"].inverse_transform(np.array([rms_base_text
blob_header]).reshape(1, -1)))

all_params_textblob_header = {
    # 'min_child_weight': [1, 5, 10],
    # 'gamma': [0.5, 1, 1.5, 2, 5],
    # 'subsample': [0.6, 0.8, 1.0],
    # 'colsample_bytree': [0.6, 0.8, 1.0],
    # 'max_depth': [3, 4, 5],
    'n_estimators': [30, 100, 200, 500],
    'learning_rate': [0.01, 0.1, 0.2, 0.3],
    'objective': ['reg:squarederror'],
    'eval_metric': ['rmse'],
    'tree_method': ['gpu_hist'],
}

best_score_textblob_header = 10000.0
run_textblob_header = 1

evallist_textblob_header = [(dmatrix_valid_textblob_header, 'eval'), (dmatrix_train
_textblob_header, 'train')]
for param_sample_textblob_header in ParameterGrid(all_params_textblob_header):
    print("----RUN ", run_textblob_header)
    xg_reg_textblob_header = xgb.train(param_sample_textblob_header,
                                       dmatrix_train_textblob_header,
                                       num_round_textblob_header * 3,
                                       evallist_textblob_header
                                       )

    xgb_predictions_textblob_header = xg_reg_textblob_header.predict(dmatrix_valid_
textblob_header)
    score_textblob_header = sqrt(mean_squared_error(y_valid_rolled_textblob_header,
xgb_predictions_textblob_header))

    if score_textblob_header < best_score_textblob_header:
        best_score_textblob_header = score_textblob_header
        best_model_textblob_header = xg_reg_textblob_header
    run_textblob_header += 1

print("Root mean squared error on valid:", best_score_textblob_header)
print("Root mean squared error on valid inverse transformed from normalization:",
      normalizers_textblob_header["OPEN"].inverse_transform(np.array([best_score_te
xtblob_header]).reshape(1, -1)))

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

xgboost_price_prediction_textblob_header = normalizers_textblob_header['OPEN'].inve
rse_transform(np.array(xgb_predictions_textblob_header).reshape(-1, 1))
```

```
print(xgboost_price_prediction_textblob_header)

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

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

new_df_vader_content = new_df_vader_content.fillna(0)
# new_df_vader_content[['Date',
#                       'OPEN',
#                       'HIGH',
#                       'LOW',
#                       'CLOSE',
#                       'VOLUME',
#                       'compound_vader_articel_content']].astype(np.float64)

new_df_vader_content['Year'] = pd.DatetimeIndex(new_df_vader_content['Date']).year
new_df_vader_content['Month'] = pd.DatetimeIndex(new_df_vader_content['Date']).mont
h
new_df_vader_content['Day'] = pd.DatetimeIndex(new_df_vader_content['Date']).day
new_df_vader_content['Hour'] = pd.DatetimeIndex(new_df_vader_content['Date']).hour
new_df_vader_content['Minute'] = pd.DatetimeIndex(new_df_vader_content['Date']).min
ute
new_df_vader_content['Second'] = pd.DatetimeIndex(new_df_vader_content['Date']).sec
ond

new_df_vader_content = new_df_vader_content.drop(['Date'], axis=1)

# 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)

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


# normalize data
def minmax_scale_vader_content(df_x, series_y, normalizers_vader_content = None):
    features_to_minmax = ['Year',
                          'Month',
                          'Day',
```

```
                           'Hour',
                           'Minute',
                           'Second',
                           '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
                               )

def encode_cyclicals_vader_content(df_x):
    # "month","day","hour", "cdbw", "dayofweek"

    #DIRECTIONS = {"N": 1.0, "NE": 2.0, "E": 3.0, "SE": 4.0, "S": 5.0, "SW": 6.0, "
W": 7.0, "NW": 8.0, "cv": np.nan}

    df_x['month_sin'] = np.sin(2 * np.pi * df_x.Month / 12)
    df_x['month_cos'] = np.cos(2 * np.pi * df_x.Month / 12)
    df_x.drop('Month', axis=1, inplace=True)

    df_x['day_sin'] = np.sin(2 * np.pi * df_x.Day / 31)
    df_x['day_cos'] = np.cos(2 * np.pi * df_x.Day / 31)
    df_x.drop('Day', axis=1, inplace=True)

    df_x['hour_sin'] = np.sin(2 * np.pi * df_x.Hour / 24)
    df_x['hour_cos'] = np.cos(2 * np.pi * df_x.Hour / 24)
    df_x.drop('Hour', axis=1, inplace=True)
```

```
    df_x['min_sin'] = np.sin(2 * np.pi * df_x.Minute / 60)
    df_x['min_cos'] = np.cos(2 * np.pi * df_x.Minute / 60)
    df_x.drop('Minute', axis=1, inplace=True)

    df_x['sec_sin'] = np.sin(2 * np.pi * df_x.Second / 60)
    df_x['sec_cos'] = np.cos(2 * np.pi * df_x.Second / 60)
    df_x.drop('Second', axis=1, inplace=True)


    return df_x

X_train_norm_vader_content = encode_cyclicals_vader_content(X_train_norm_vader_cont
ent)
X_valid_norm_vader_content = encode_cyclicals_vader_content(X_valid_norm_vader_cont
ent)
X_test_norm_vader_content = encode_cyclicals_vader_content(X_test_norm_vader_conten
t)

# 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_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_vader_content = X_valid_rolled_vader_content.reshape(shape_vader_
content[0],
                                                                       shape_vader_
content[1]*shape_vader_content[2]
                                                                       )

# XGBoost needs it's custom data format to run quickly
dmatrix_train_vader_content = xgb.DMatrix(data=X_train_flattened_vader_content,
```

```
                                          label=y_train_rolled_vader_content
                                          )

dmatrix_valid_vader_content = xgb.DMatrix(data=X_valid_flattened_vader_content,
                                          label=y_valid_rolled_vader_content
                                          )

params_vader_content = {'objective': 'reg:squarederror', 'eval_metric': 'rmse', 'n_
estimators': 30, 'tree_method':'gpu_hist'}
#param['nthread'] = 4
evallist_vader_content = [(dmatrix_valid_vader_content, 'eval'), (dmatrix_train_vad
er_content, 'train')]

#After some tests, it turned out to overfit after this point
num_round_vader_content = 12

xg_reg_vader_content = xgb.train(params_vader_content,
                                 dmatrix_train_vader_content,
                                 num_round_vader_content,
                                 evallist_vader_content
                                 )

xgb_predictions_vader_content = xg_reg_vader_content.predict(dmatrix_valid_vader_co
ntent)

rms_base_vader_content = sqrt(mean_squared_error(y_valid_rolled_vader_content, xgb_
predictions_vader_content))

print("Root mean squared error on valid:",rms_base_vader_content)
print("Root mean squared error on valid inverse transformed from normalization:",
      normalizers_vader_content["OPEN"].inverse_transform(np.array([rms_base_vader_
content]).reshape(1, -1)))

all_params_vader_content = {
    # 'min_child_weight': [1, 5, 10],
    # 'gamma': [0.5, 1, 1.5, 2, 5],
    # 'subsample': [0.6, 0.8, 1.0],
    # 'colsample_bytree': [0.6, 0.8, 1.0],
    # 'max_depth': [3, 4, 5],
    'n_estimators': [30, 100, 200, 500],
    'learning_rate': [0.01, 0.1, 0.2, 0.3],
    'objective': ['reg:squarederror'],
    'eval_metric': ['rmse'],
    'tree_method': ['gpu_hist'],
}

best_score_vader_content = 10000.0
run_vader_content = 1

evallist_vader_content = [(dmatrix_valid_vader_content, 'eval'), (dmatrix_train_vad
er_content, 'train')]
for param_sample_vader_content in ParameterGrid(all_params_vader_content):
    print("----RUN ", run_vader_content)
    xg_reg_vader_content = xgb.train(param_sample_vader_content,
                                     dmatrix_train_vader_content,
                                     num_round_vader_content * 3,
                                     evallist_vader_content)

    xgb_predictions_vader_content = xg_reg_vader_content.predict(dmatrix_valid_vade
r_content)
    score_vader_content = sqrt(mean_squared_error(y_valid_rolled_vader_content, xgb
_predictions_vader_content))

    if score_vader_content < best_score_vader_content:
```

```
        best_score_vader_content = score_vader_content
        best_model_vader_content = xg_reg_vader_content
    run_vader_content += 1

print("Root mean squared error on valid:", best_score_vader_content)
print("Root mean squared error on valid inverse transformed from normalization:",
      normalizers_vader_content["OPEN"].inverse_transform(np.array([best_score_vade
r_content]).reshape(1, -1)))

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

xgboost_price_prediction_vader_content = normalizers_vader_content['OPEN'].inverse_
transform(np.array(xgb_predictions_vader_content).reshape(-1, 1))

print(xgboost_price_prediction_vader_content)

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

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

new_df_vader_header = new_df_vader_header.fillna(0)
# new_df_vader_header[['Date',
#                      'OPEN',
#                      'HIGH',
#                      'LOW',
#                      'CLOSE',
#                      'VOLUME',
#                      'compound_vader_header']].astype(np.float64)

new_df_vader_header['Year'] = pd.DatetimeIndex(new_df_vader_header['Date']).year
new_df_vader_header['Month'] = pd.DatetimeIndex(new_df_vader_header['Date']).month
new_df_vader_header['Day'] = pd.DatetimeIndex(new_df_vader_header['Date']).day
new_df_vader_header['Hour'] = pd.DatetimeIndex(new_df_vader_header['Date']).hour
new_df_vader_header['Minute'] = pd.DatetimeIndex(new_df_vader_header['Date']).minut
e
new_df_vader_header['Second'] = pd.DatetimeIndex(new_df_vader_header['Date']).secon
d

new_df_vader_header = new_df_vader_header.drop(['Date'], axis=1)

# 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 = ['Year',
                          'Month',
                          'Day',
                          'Hour',
                          'Minute',
                          'Second',
                          '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
                              )

def encode_cyclicals_vader_header(df_x):
    # "month","day","hour", "cdbw", "dayofweek"

    #DIRECTIONS = {"N": 1.0, "NE": 2.0, "E": 3.0, "SE": 4.0, "S": 5.0, "SW": 6.0, "
W": 7.0, "NW": 8.0, "cv": np.nan}
```

```
    df_x['month_sin'] = np.sin(2 * np.pi * df_x.Month / 12)
    df_x['month_cos'] = np.cos(2 * np.pi * df_x.Month / 12)
    df_x.drop('Month', axis=1, inplace=True)

    df_x['day_sin'] = np.sin(2 * np.pi * df_x.Day / 31)
    df_x['day_cos'] = np.cos(2 * np.pi * df_x.Day / 31)
    df_x.drop('Day', axis=1, inplace=True)

    df_x['hour_sin'] = np.sin(2 * np.pi * df_x.Hour / 24)
    df_x['hour_cos'] = np.cos(2 * np.pi * df_x.Hour / 24)
    df_x.drop('Hour', axis=1, inplace=True)

    df_x['min_sin'] = np.sin(2 * np.pi * df_x.Minute / 60)
    df_x['min_cos'] = np.cos(2 * np.pi * df_x.Minute / 60)
    df_x.drop('Minute', axis=1, inplace=True)

    df_x['sec_sin'] = np.sin(2 * np.pi * df_x.Second / 60)
    df_x['sec_cos'] = np.cos(2 * np.pi * df_x.Second / 60)
    df_x.drop('Second', axis=1, inplace=True)

    return df_x

X_train_norm_vader_header = encode_cyclicals_vader_header(X_train_norm_vader_heade
r)
X_valid_norm_vader_header = encode_cyclicals_vader_header(X_valid_norm_vader_heade
r)
X_test_norm_vader_header = encode_cyclicals_vader_header(X_test_norm_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()]
                                         )

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_vader_header = X_valid_rolled_vader_header.reshape(shape_vader_he
ader[0],
                                                                      shape_vader_he
ader[1]*shape_vader_header[2]
                                                                      )

# XGBoost needs it's custom data format to run quickly
dmatrix_train_vader_header = xgb.DMatrix(data=X_train_flattened_vader_header,
                                         label=y_train_rolled_vader_header
                                         )

dmatrix_valid_vader_header = xgb.DMatrix(data=X_valid_flattened_vader_header,
                                         label=y_valid_rolled_vader_header
                                         )

params_vader_header = {'objective': 'reg:squarederror', 'eval_metric': 'rmse', 'n_e
stimators': 30, 'tree_method':'gpu_hist'}
#param['nthread'] = 4
evallist_vader_header = [(dmatrix_valid_vader_header, 'eval'), (dmatrix_train_vader
_header, 'train')]

#After some tests, it turned out to overfit after this point
num_round_vader_header = 12

xg_reg_vader_header = xgb.train(params_vader_header,
                                dmatrix_train_vader_header,
                                num_round_vader_header,
                                evallist_vader_header
                                )

xgb_predictions_vader_header = xg_reg_vader_header.predict(dmatrix_valid_vader_head
er)

rms_base_vader_header = sqrt(mean_squared_error(y_valid_rolled_vader_header, xgb_pr
edictions_vader_header))

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

all_params_vader_header = {
    # 'min_child_weight': [1, 5, 10],
    # 'gamma': [0.5, 1, 1.5, 2, 5],
    # 'subsample': [0.6, 0.8, 1.0],
    # 'colsample_bytree': [0.6, 0.8, 1.0],
    # 'max_depth': [3, 4, 5],
    'n_estimators': [30, 100, 200, 500],
    'learning_rate': [0.01, 0.1, 0.2, 0.3],
    'objective': ['reg:squarederror'],
    'eval_metric': ['rmse'],
    'tree_method': ['gpu_hist'],
}

best_score_vader_header = 10000.0
run_vader_header = 1

evallist_vader_header = [(dmatrix_valid_vader_header, 'eval'), (dmatrix_train_vader
_header, 'train')]
for param_sample_vader_header in ParameterGrid(all_params_vader_header):
    print("----RUN ", run_vader_header)
    xg_reg_vader_header = xgb.train(param_sample_vader_header,
```

```
                                    dmatrix_train_vader_header,
                                    num_round_vader_header * 3,
                                    evallist_vader_header)

    xgb_predictions_vader_header = xg_reg_vader_header.predict(dmatrix_valid_vader_
header)
    score_vader_header = sqrt(mean_squared_error(y_valid_rolled_vader_header, xgb_p
redictions_vader_header))

    if score_vader_header < best_score_vader_header:
        best_score_vader_header = score_vader_header
        best_model_vader_header = xg_reg_vader_header
    run_vader_header += 1

print("Root mean squared error on valid:", best_score_vader_header)
print("Root mean squared error on valid inverse transformed from normalization:",
      normalizers_vader_header["OPEN"].inverse_transform(np.array([best_score_vader
_header]).reshape(1, -1)))

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

xgboost_price_prediction_vader_header = normalizers_vader_header['OPEN'].inverse_tr
ansform(np.array(xgb_predictions_vader_header).reshape(-1, 1))

print(xgboost_price_prediction_vader_header)

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

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

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

new_df_without_semantics['Year'] = pd.DatetimeIndex(new_df_without_semantics['Date
']).year
new_df_without_semantics['Month'] = pd.DatetimeIndex(new_df_without_semantics['Date
']).month
new_df_without_semantics['Day'] = pd.DatetimeIndex(new_df_without_semantics['Date
']).day
new_df_without_semantics['Hour'] = pd.DatetimeIndex(new_df_without_semantics['Date
']).hour
new_df_without_semantics['Minute'] = pd.DatetimeIndex(new_df_without_semantics['Dat
e']).minute
new_df_without_semantics['Second'] = pd.DatetimeIndex(new_df_without_semantics['Dat
e']).second

new_df_without_semantics = new_df_without_semantics.drop(['Date'], axis=1)

# 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 = ['Year',
                          'Month',
                          'Day',
                          'Hour',
                          'Minute',
                          'Second',
                          '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
                                   )

def encode_cyclicals_without_semantics(df_x):
    # "month","day","hour", "cdbw", "dayofweek"

    #DIRECTIONS = {"N": 1.0, "NE": 2.0, "E": 3.0, "SE": 4.0, "S": 5.0, "SW": 6.0, "
W": 7.0, "NW": 8.0, "cv": np.nan}

    df_x['month_sin'] = np.sin(2 * np.pi * df_x.Month / 12)
    df_x['month_cos'] = np.cos(2 * np.pi * df_x.Month / 12)
    df_x.drop('Month', axis=1, inplace=True)

    df_x['day_sin'] = np.sin(2 * np.pi * df_x.Day / 31)
    df_x['day_cos'] = np.cos(2 * np.pi * df_x.Day / 31)
    df_x.drop('Day', axis=1, inplace=True)

    df_x['hour_sin'] = np.sin(2 * np.pi * df_x.Hour / 24)
    df_x['hour_cos'] = np.cos(2 * np.pi * df_x.Hour / 24)
    df_x.drop('Hour', axis=1, inplace=True)

    df_x['min_sin'] = np.sin(2 * np.pi * df_x.Minute / 60)
    df_x['min_cos'] = np.cos(2 * np.pi * df_x.Minute / 60)
    df_x.drop('Minute', axis=1, inplace=True)

    df_x['sec_sin'] = np.sin(2 * np.pi * df_x.Second / 60)
    df_x['sec_cos'] = np.cos(2 * np.pi * df_x.Second / 60)
    df_x.drop('Second', axis=1, inplace=True)


    return df_x

X_train_norm_without_semantics = encode_cyclicals_without_semantics(X_train_norm_wi
thout_semantics)
X_valid_norm_without_semantics = encode_cyclicals_without_semantics(X_valid_norm_wi
thout_semantics)
X_test_norm_without_semantics = encode_cyclicals_without_semantics(X_test_norm_with
out_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_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_without_semantics = X_valid_rolled_without_semantics.reshape(shap
e_without_semantics[0],
                                                                             shap
e_without_semantics[1]*shape_without_semantics[2]
                                                                             )

# XGBoost needs it's custom data format to run quickly
dmatrix_train_without_semantics = xgb.DMatrix(data=X_train_flattened_without_semant
ics,
                                              label=y_train_rolled_without_semantic
s
                                              )

dmatrix_valid_without_semantics = xgb.DMatrix(data=X_valid_flattened_without_semant
ics,
                                              label=y_valid_rolled_without_semantic
s
                                              )

params_without_semantics = {'objective': 'reg:squarederror', 'eval_metric': 'rmse',
'n_estimators': 30, 'tree_method':'gpu_hist'}
#param['nthread'] = 4
evallist_without_semantics = [(dmatrix_valid_without_semantics, 'eval'), (dmatrix_t
rain_without_semantics, 'train')]

#After some tests, it turned out to overfit after this point
num_round_without_semantics = 12

xg_reg_without_semantics = xgb.train(params_without_semantics,
                                     dmatrix_train_without_semantics,
                                     num_round_without_semantics,
                                     evallist_without_semantics
                                     )

xgb_predictions_without_semantics = xg_reg_without_semantics.predict(dmatrix_valid_
```

```
without_semantics)

rms_base_without_semantics = sqrt(mean_squared_error(y_valid_rolled_without_semanti
cs, xgb_predictions_without_semantics))

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

all_params_without_semantics = {
    # 'min_child_weight': [1, 5, 10],
    # 'gamma': [0.5, 1, 1.5, 2, 5],
    # 'subsample': [0.6, 0.8, 1.0],
    # 'colsample_bytree': [0.6, 0.8, 1.0],
    # 'max_depth': [3, 4, 5],
    'n_estimators': [30, 100, 200, 500],
    'learning_rate': [0.01, 0.1, 0.2, 0.3],
    'objective': ['reg:squarederror'],
    'eval_metric': ['rmse'],
    'tree_method': ['gpu_hist'],
}

best_score_without_semantics = 10000.0
run_without_semantics = 1

evallist_without_semantics = [(dmatrix_valid_without_semantics, 'eval'), (dmatrix_t
rain_without_semantics, 'train')]
for param_sample_without_semantics in ParameterGrid(all_params_without_semantics):
    print("----RUN ", run_without_semantics)
    xg_reg_without_semantics = xgb.train(param_sample_without_semantics,
                                         dmatrix_train_without_semantics,
                                         num_round_without_semantics * 3,
                                         evallist_without_semantics)

    xgb_predictions_without_semantics = xg_reg_without_semantics.predict(dmatrix_va
lid_without_semantics)
    score_without_semantics = sqrt(mean_squared_error(y_valid_rolled_without_semant
ics, xgb_predictions_without_semantics))

    if score_without_semantics < best_score_without_semantics:
        best_score_without_semantics = score_without_semantics
        best_model_without_semantics = xg_reg_without_semantics
    run_without_semantics += 1

print("Root mean squared error on valid:", best_score_without_semantics)
print("Root mean squared error on valid inverse transformed from normalization:",
      normalizers_without_semantics["OPEN"].inverse_transform(np.array([best_score_
without_semantics]).reshape(1, -1)))

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

xgboost_price_prediction_without_semantics = normalizers_without_semantics['OPEN'].
inverse_transform(np.array(xgb_predictions_without_semantics).reshape(-1, 1))

print(xgboost_price_prediction_without_semantics)

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


plt.figure(figsize=(10,5))
plt.plot(xgboost_price_prediction_flair_content, color='green', label='Predicted Po
```

```
rsche Stock Price with flair content analysis')
plt.plot(xgboost_price_prediction_flair_header, color='red', label='Predicted Porsc
he Stock Price with flair header analysis')
plt.plot(xgboost_price_prediction_textblob_content, color='orange', label='Predicte
d Porsche Stock Price with textblob content analysis')
plt.plot(xgboost_price_prediction_textblob_header, color='blue', label='Predicted P
orsche Stock Price with textblob header analysis')
plt.plot(xgboost_price_prediction_vader_content, color='cyan', label='Predicted Por
sche Stock Price with vader content analysis')
plt.plot(xgboost_price_prediction_vader_header, color='magenta', label='Predicted P
orsche Stock Price with vader header analysis')
plt.plot(xgboost_price_prediction_without_semantics, color='yellow', label='Predict
ed Porsche Stock Price without semantics analysis')
plt.title('Porsche Stock Price Prediction')
plt.xlabel('Time')
plt.ylabel('Porsche 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\xgboost\porsche\ho
urly\prediction_porsche_' + date_today + '.png',
            bbox_inches="tight",
            dpi=100,
            pad_inches=1.5)
plt.show()
print('Run is finished and plot is saved!')
```