Certificate in Quantitative Finance : Final Project

DEEP LEARNING for Time Series Prediction

A report by Debashish GOGOI

SUMMARY

Time series prediction of stock price movements poses a big challenge for most finance practitioners due to the highly stochastic nature of price changes. Traditional statistical time series approaches like ARIMA have been in use for long but it's difficult to implement non-linearity if/when present in the data. On the other hand, advances in Deep Learning have opened a plethora of new possibilities for time series predictions especially when autoregressive methods fail or are of limited use. This project employs a special type of Deep Neural Nets called the Recurrent Neural Networks (LSTMs) to predict the direction of price movements for five European stocks from five different industries, where a classification approach for 5-Day forward return predictions is used. Three different LSTM architectures were used and compared for accuracies. A comparison of LSTMs with tree based and linear classifiers has also been carried out, and further possibilities of turning the project into a regression problem were checked.

The sections below are organized as follows:

In [338…

```
# Imports

import pandas as pd
import numpy as np
import datetime as dt

!pip install -q yfinance==0.1.63
import yfinance as yf

#!pip install -q talib
import talib

from sklearn import metrics
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn import preprocessing,impute
from sklearn import decomposition, tree, cluster
from sklearn.pipeline import Pipeline

from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.graphics import tsaplots
from scipy.spatial.distance import cdist

import matplotlib.pyplot as plt
import seaborn as sns

#!pip install -q minisom
from minisom import MiniSom

import itertools

import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras.models import Model, load_model
from tensorflow.keras import optimizers
from tensorflow.keras import callbacks
from tensorflow.keras import utils
from tensorflow.keras import backend as K
from tensorflow.keras.utils import plot_model


!pip install -q tqdm==4.62.1
import tqdm
from tqdm.keras import TqdmCallback

import random
```

```
random.seed(22)
tf.random.set_seed(22)
np.random.seed(22)

import tabulate
import pickle

import warnings
warnings.filterwarnings('ignore')
```

Data Extraction

In [339…

```
# Data extracts

#Features: Spreads, Xassets, Fin Ratios (extracted from Bloomberg and Capital IQ!)

spreads=pd.read_csv(r'../inputs/spreads.csv');spreads['Date']=pd.to_datetime(spreads['Date'])
xassets=pd.read_excel(r'../inputs/xassets.xlsx')

#Imputing missing values:
imputer=impute.SimpleImputer()
xassets=pd.DataFrame(imputer.fit_transform(xassets.set_index('Date')),columns=xassets.set_index('Date').columns,index=xassets.set_

dtickers=['BATSLN','DT','RENAUL','SIEGR','TTEFP'] #Debt tickers used to extract values from Capital IQ, Bloomberg and iBoxx
fin_ratios={}

for ticker in dtickers:
    fin_ratios[ticker]=pd.read_excel(r'../inputs/fin_ratios.xlsx',sheet_name=ticker).replace(0,np.nan).replace('NM',np.nan).bfill(
    fin_ratios[ticker]['Date']=fin_ratios[ticker].iloc[:,0].apply(lambda x:dt.datetime.strptime(str(int(x[2:3])*3)+'-'+str(x[-4:])
    fin_ratios[ticker]=fin_ratios[ticker].iloc[:,1:].set_index('Date')
    fin_ratios[ticker].columns=[c for c in map(lambda x:x.split('iq_')[1],fin_ratios[ticker].columns.tolist())]
```

In [340…

```
# Extracting Stock Prices from Yahoo Finance!

ytickers=['BATS.L','DTE.DE','RNO.PA','SIE.DE','TTE.PA'] #5 European stocks from different industries
stocks={}

for ticker,ticker_ in zip(ytickers,dtickers):
    stocks[ticker]=yf.download(progress=False,start='2004-01-01',end='2021-08-10',tickers=ticker)
    stocks[ticker]['dret']=stocks[ticker].Close.apply(lambda x:np.log(x))-stocks[ticker].Close.shift(1).apply(lambda x:np.log(x))
    stocks[ticker]=stocks[ticker].dropna()

    #combining prices with spreads,finratios and xassets:
    stocks[ticker]=stocks[ticker].join(xassets).join(spreads[spreads.Ticker==ticker_].drop('Ticker',axis=1).set_index('Date')).joi
    stocks[ticker]=stocks[ticker].bfill().ffill()#missing fin ratios
    stocks[ticker]=stocks[ticker].dropna(how='all',axis=1)
```

EDA: Timeseries Decomposition

In [341…

```
#Seasonality
#Trend
#Noise

class TS_EDA:

    def __init__(self,df,ticker):
        self.df=df
        self.ticker=ticker

    def seasonality(self,figsize=(30,6)):
        ts=self.df.copy().Close
        ts=ts.asfreq('D').ffill()
        decomp=seasonal_decompose(ts,model='multiplicative')

        fig,ax=plt.subplots(1,4,figsize=figsize)

        print('\n')
        print('Ticker: '+self.ticker)

        ax[0].plot(ts,color='k')
        ax[1].plot(decomp.seasonal,color='brown')
        ax[2].plot(decomp.trend,color='brown')
        ax[3].plot(decomp.resid,color='brown')

        ax[0].set_title('Close Price')
        ax[1].set_title('Seasonality')
        ax[2].set_title('Trend')
        ax[3].set_title('Noise')

        plt.show()

    def feature_plots(self,figsize=(30,8)):
        print('\n')
        print('Stock: '+self.ticker)
        fig,ax=plt.subplots(1,2,figsize=figsize,gridspec_kw={'width_ratios':[1.5,2]})
```

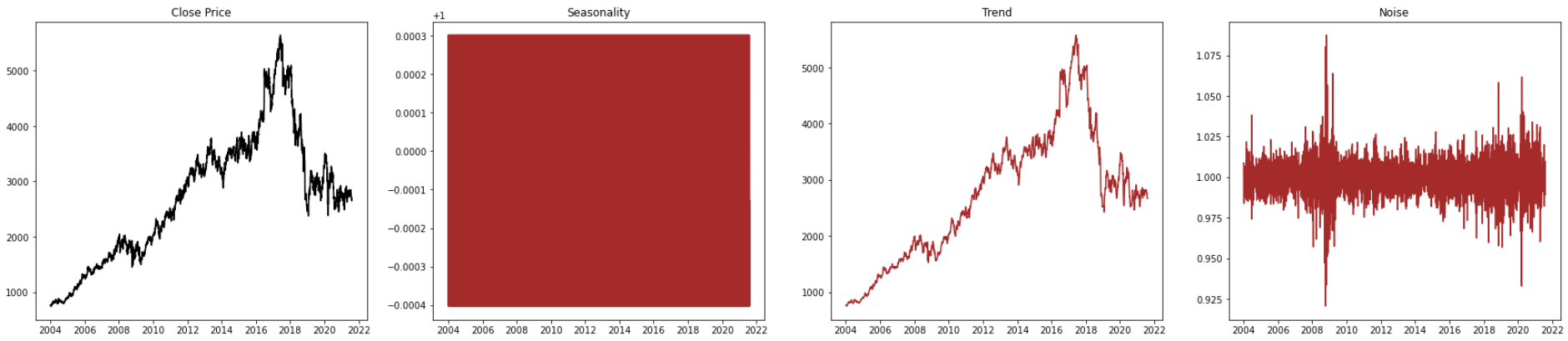
```
        #fig,ax=plt.subplots(1,2,figsize=figsize)

        sns.heatmap(self.df.corr().iloc[:,:114],ax=ax[0],cmap='viridis')
        ax[0].set_title('Correlation heatmap')

        corr1=self.df.drop(['Open','High','Low','Adj Close'],axis=1).iloc[:,:114].corr()['Close'].sort_values(ascending=True)[-21:
        corr2=self.df.drop(['Open','High','Low','Adj Close'],axis=1).iloc[:,:114].corr()['Close'].sort_values(ascending=False)[-20
        ax[1].bar(corr1.append(corr2).index,height=corr1.append(corr2),color=(corr1.append(corr2)>0).map({True:'darkgreen',False:'
        ax[1].set_ylabel('Correlation coefficients')
        ax[1].set_xlabel('Features')
        ax[1].set_xticklabels(corr1.append(corr2).index,rotation=90)
        ax[1].set_title('Top 20 features positively and negatively correlated with the Stock Price')
        ax[1].set_xmargin(0)
        ax[1].set_ymargin(0)
        plt.show()


    def ac_plots(self,figsize=(25,6)):
        fig,ax=plt.subplots(1,2,figsize=figsize)
        ac=tsaplots.plot_acf(self.df.Close.values,lags=50,ax=ax[0],title=self.ticker+': Autocorrelation')
        pac=tsaplots.plot_pacf(self.df.Close.values,lags=50,ax=ax[1],title=self.ticker+': Partial Autocorrelation')
        ax[0].set_xlabel('lags')
        ax[1].set_xlabel('lags')
        ax[0].set_ylabel('correlation coeff.')
        ax[1].set_ylabel('correlation coeff.')
        plt.show()
```

Seasonality

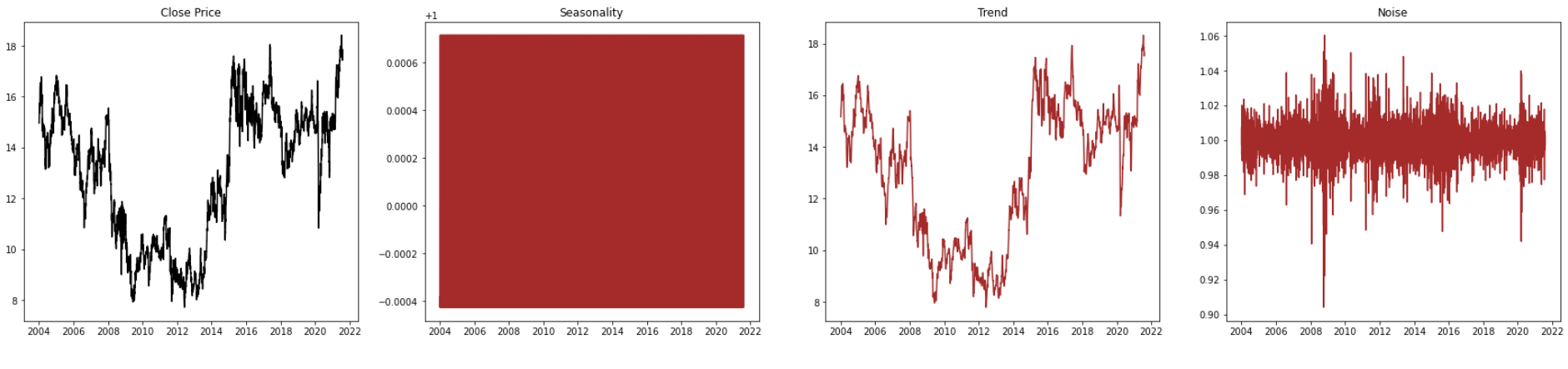
In [342…

```
TS_EDA(stocks['BATS.L'].copy(),'British American Tobacco').seasonality()
TS_EDA(stocks['DTE.DE'].copy(),'Deutsche Telekom').seasonality()
TS_EDA(stocks['RNO.PA'].copy(),'Renault').seasonality()
TS_EDA(stocks['SIE.DE'].copy(),'Siemens').seasonality()
TS_EDA(stocks['TTE.PA'].copy(),'Total Energies').seasonality()
```

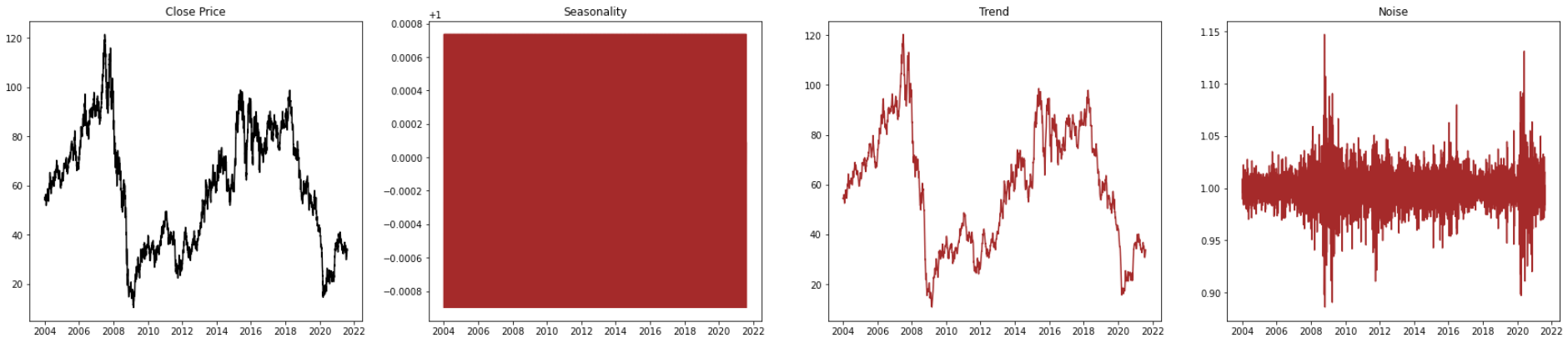
Ticker: British American Tobacco



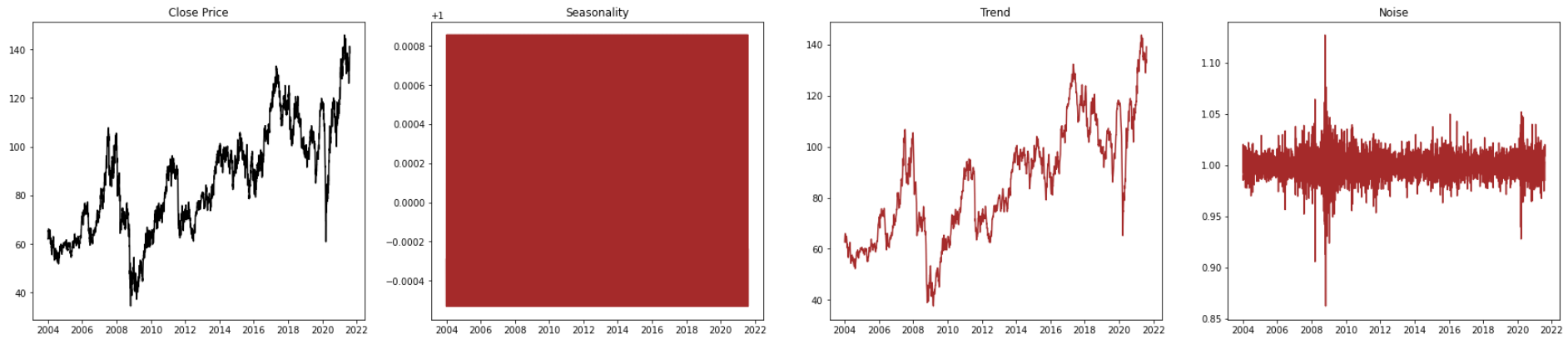
Ticker: Deutsche Telekom



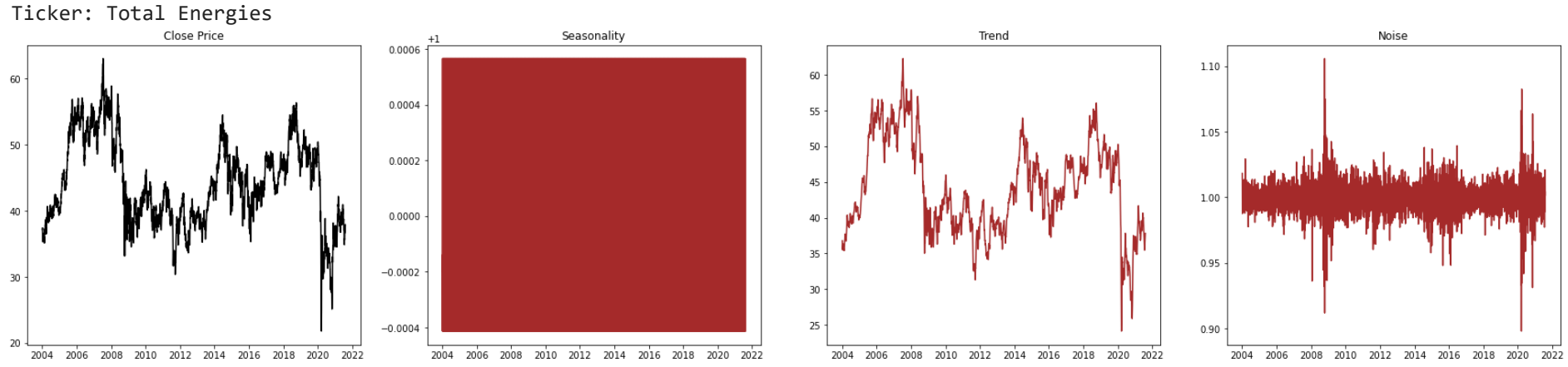
Ticker: Renault



Ticker: Siemens



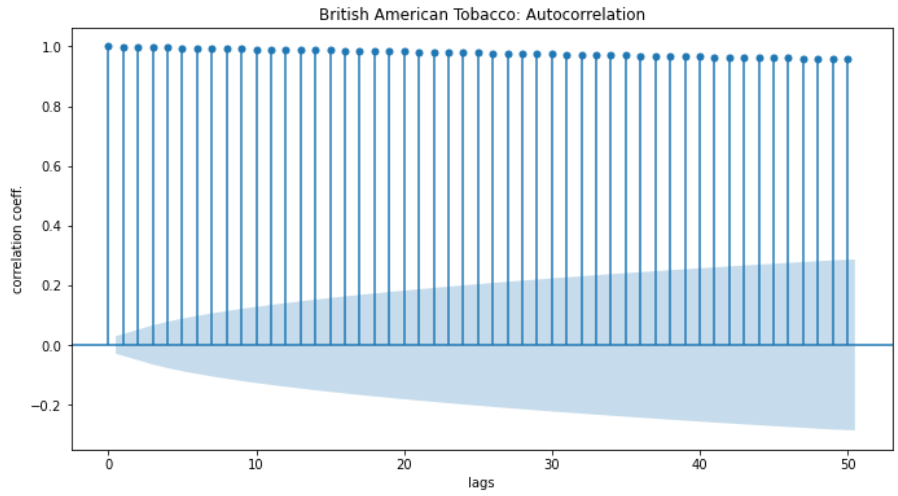
Ticker: Total Energies

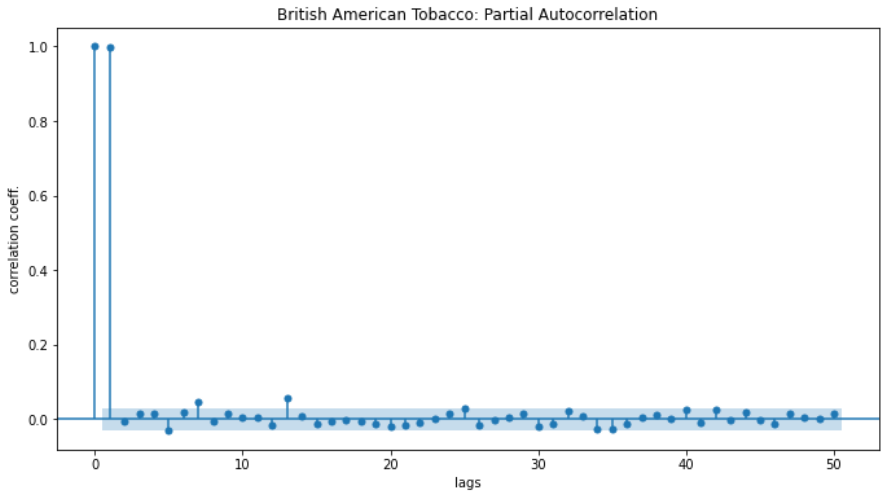


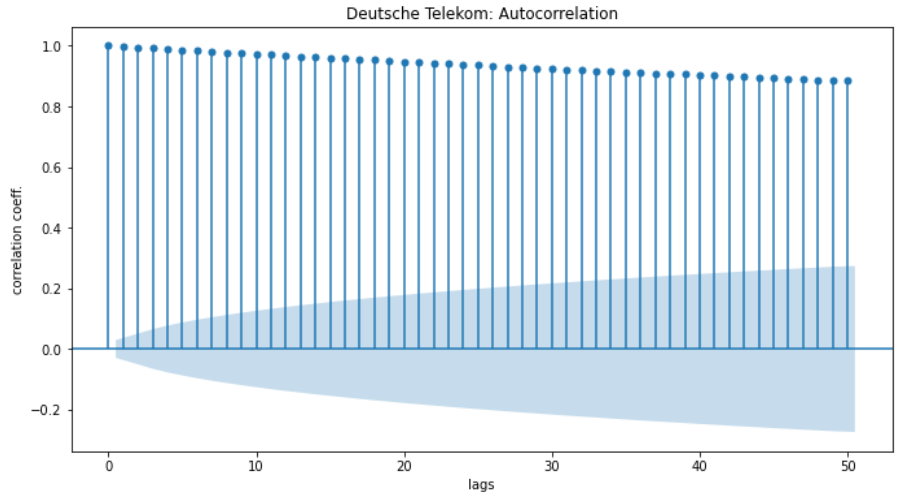
Autocorrelations

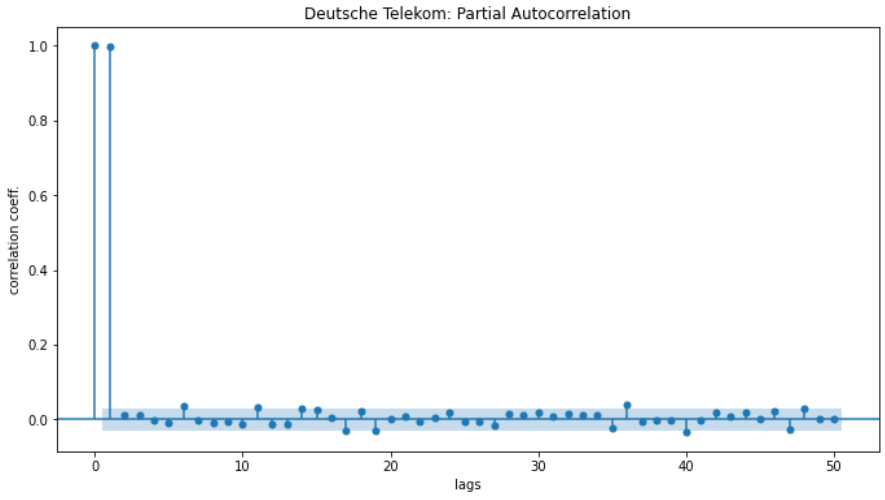
In [343…

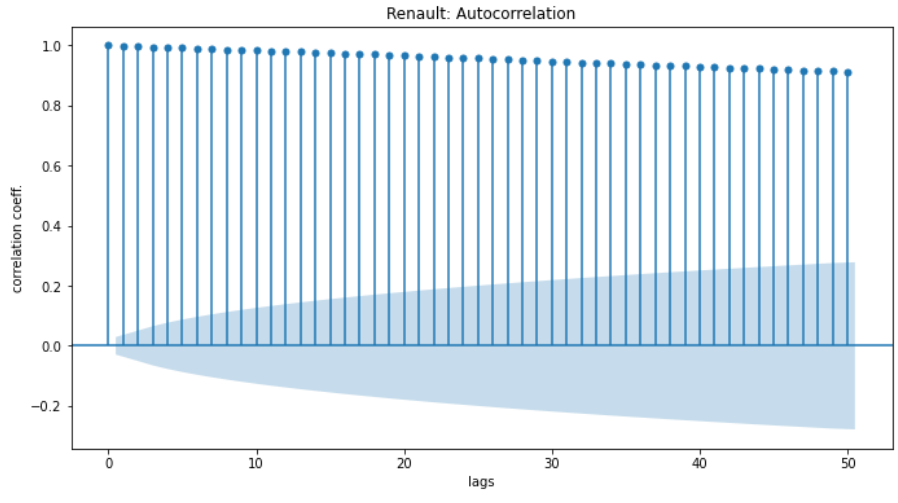
```
TS_EDA(stocks['BATS.L'].copy(),'British American Tobacco').ac_plots()
TS_EDA(stocks['DTE.DE'].copy(),'Deutsche Telekom').ac_plots()
TS_EDA(stocks['RNO.PA'].copy(),'Renault').ac_plots()
TS_EDA(stocks['SIE.DE'].copy(),'Siemens').ac_plots()
TS_EDA(stocks['TTE.PA'].copy(),'Total Energies').ac_plots()
```

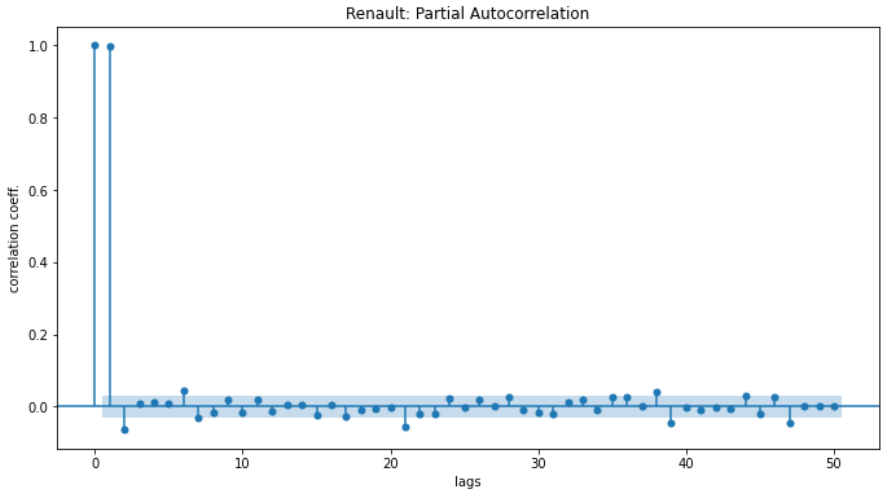


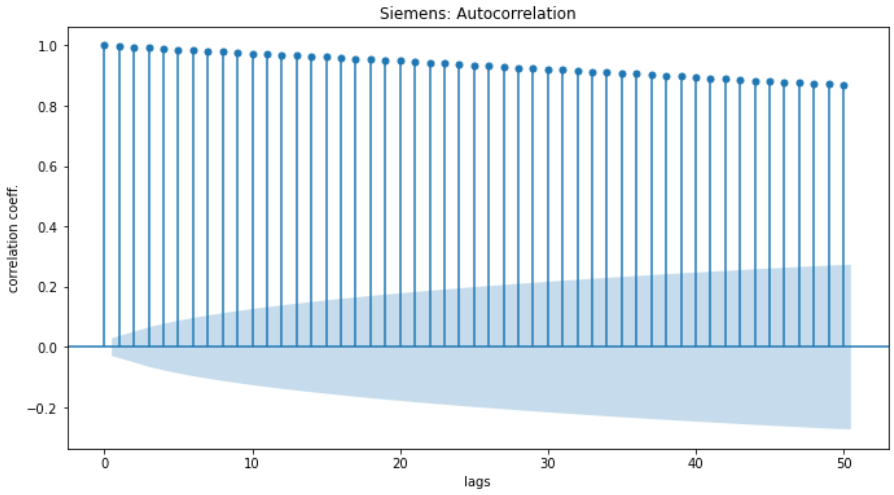


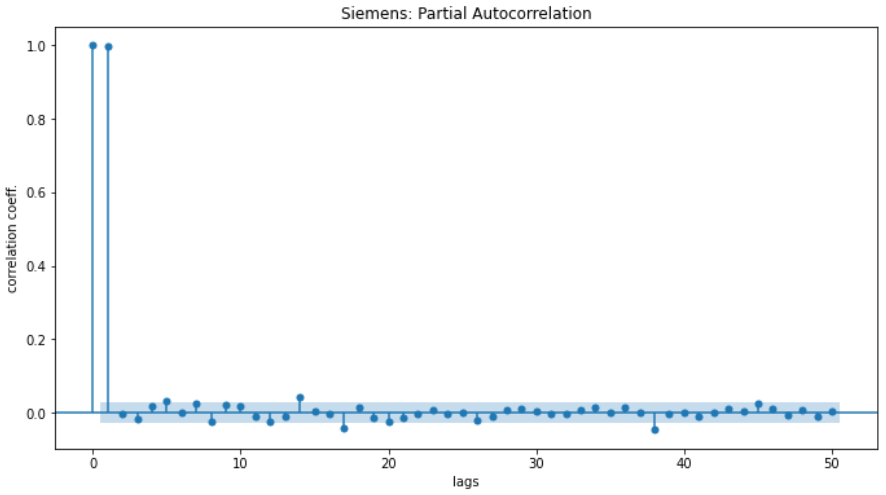


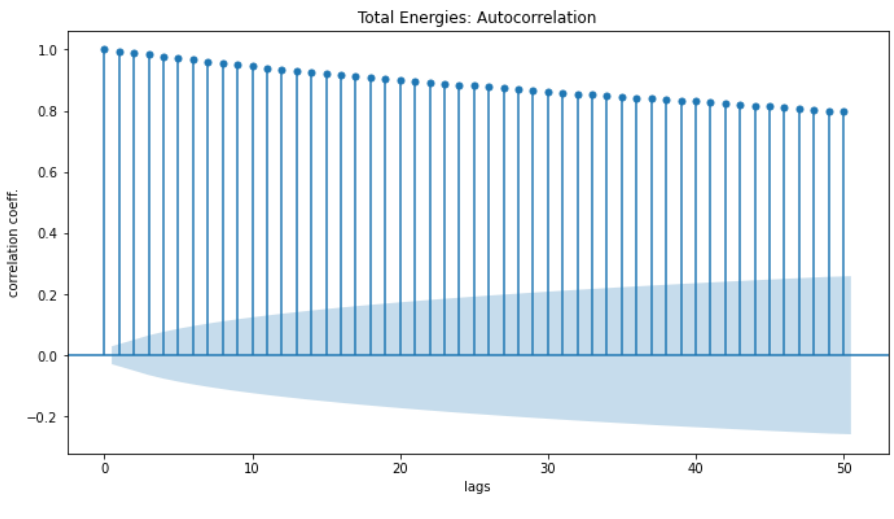


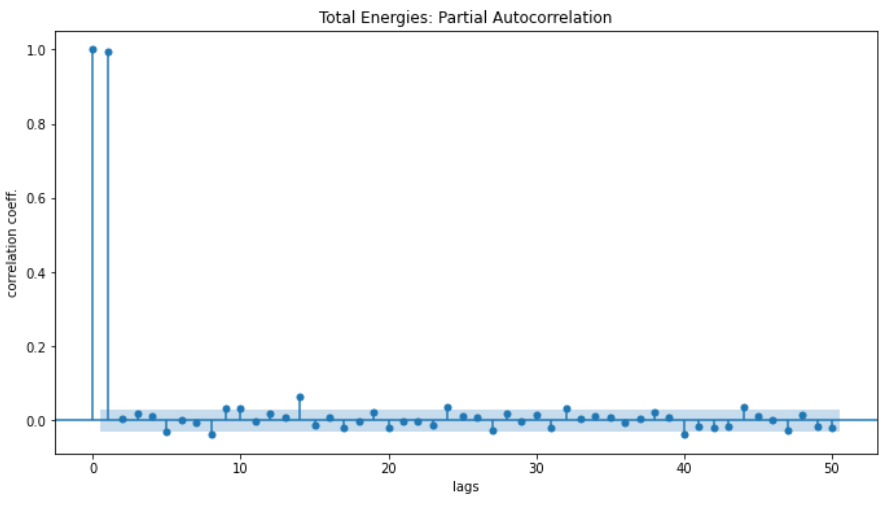










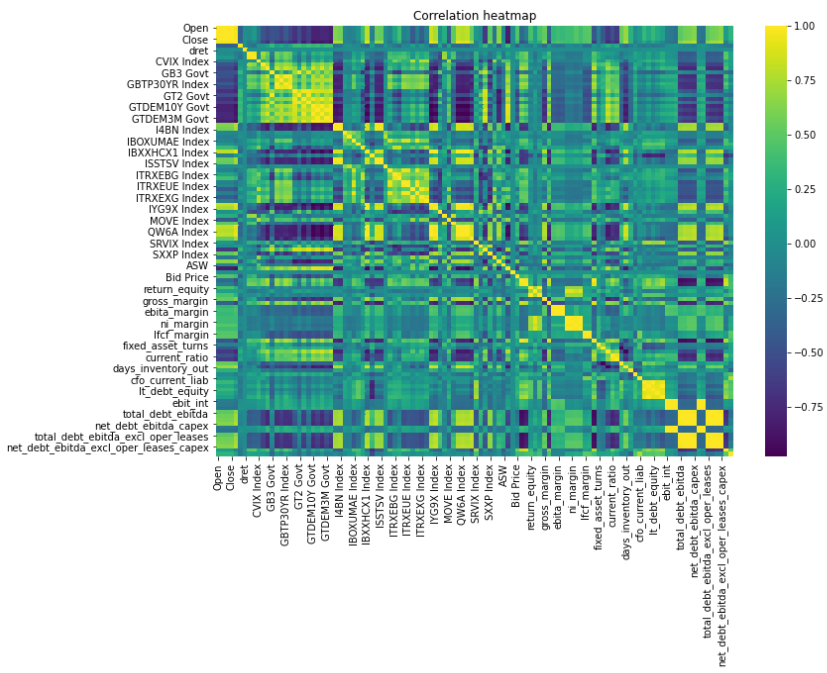


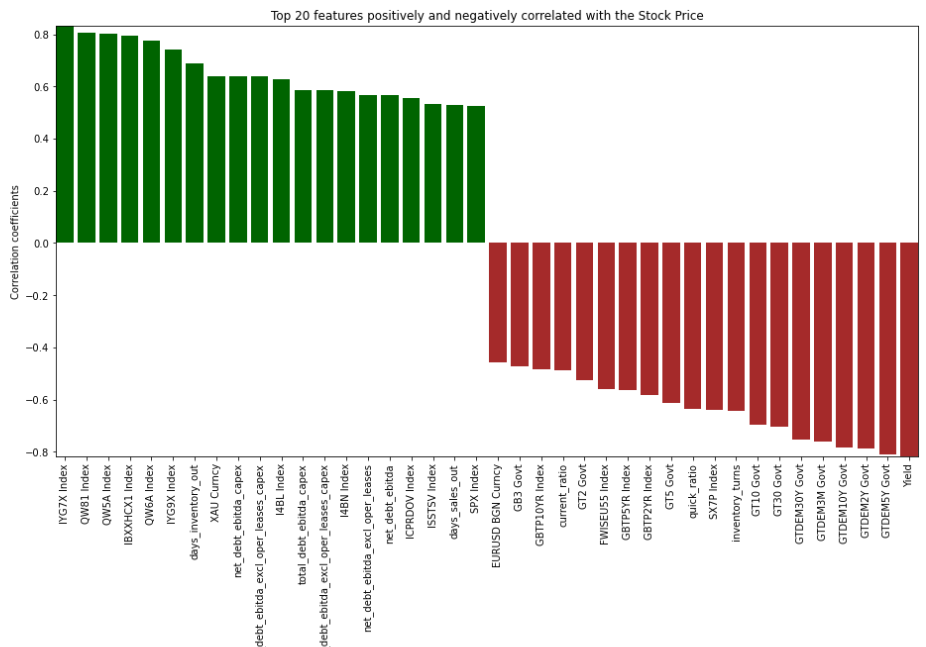
Correlations

In [344…

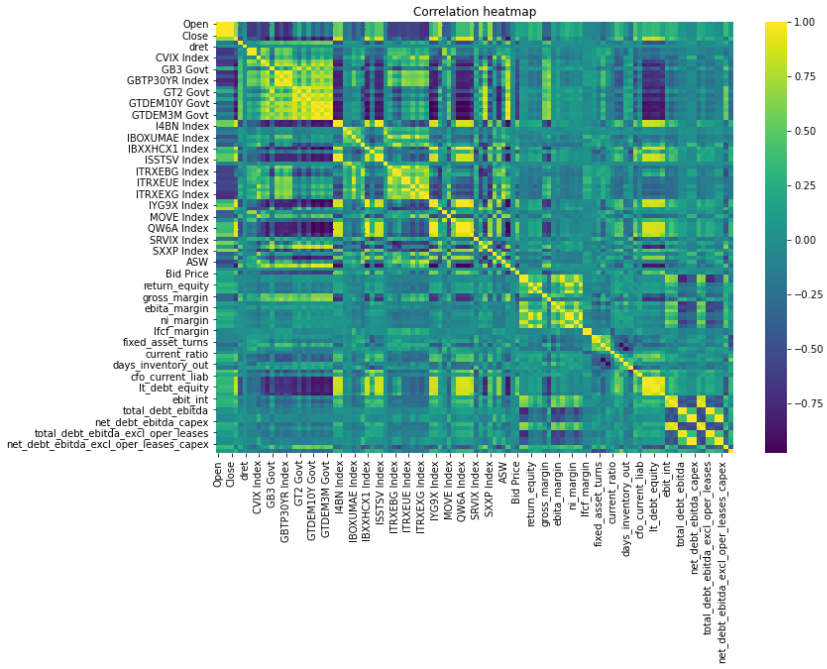
```
TS_EDA(stocks['BATS.L'].copy(),'British American Tobacco').feature_plots()
TS_EDA(stocks['DTE.DE'].copy(),'Deutsche Telekom').feature_plots()
TS_EDA(stocks['RNO.PA'].copy(),'Renault').feature_plots()
TS_EDA(stocks['SIE.DE'].copy(),'Siemens').feature_plots()
TS_EDA(stocks['TTE.PA'].copy(),'Total Energies').feature_plots()
```

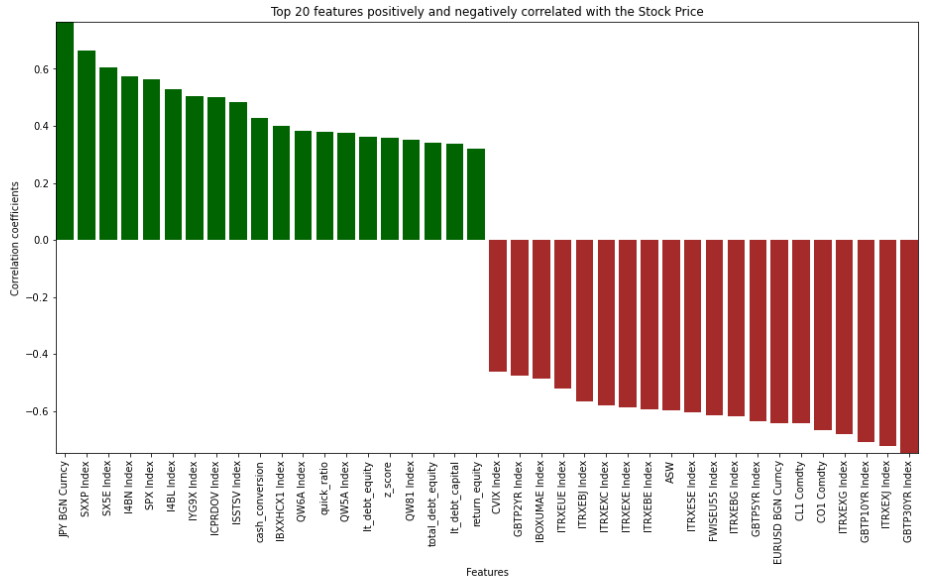
Stock: British American Tobacco



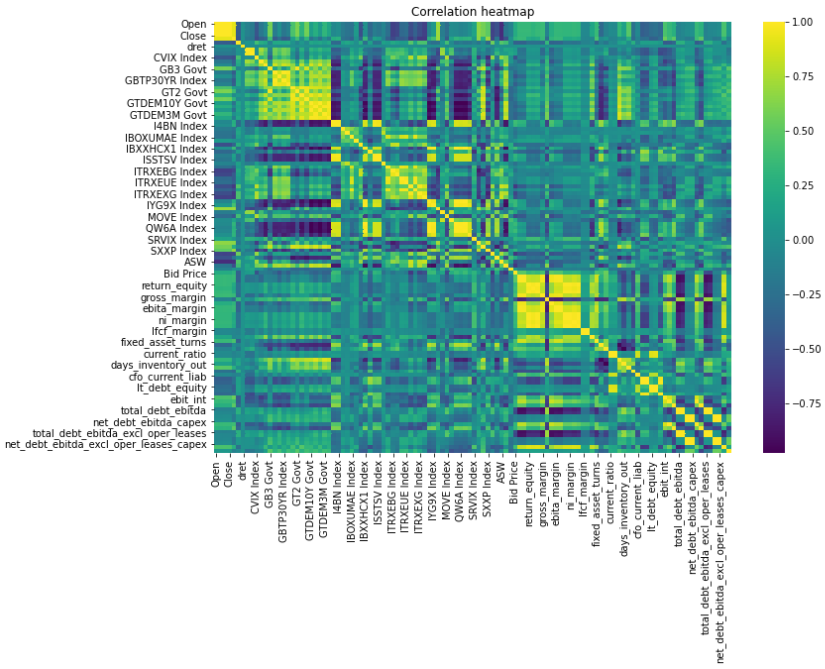


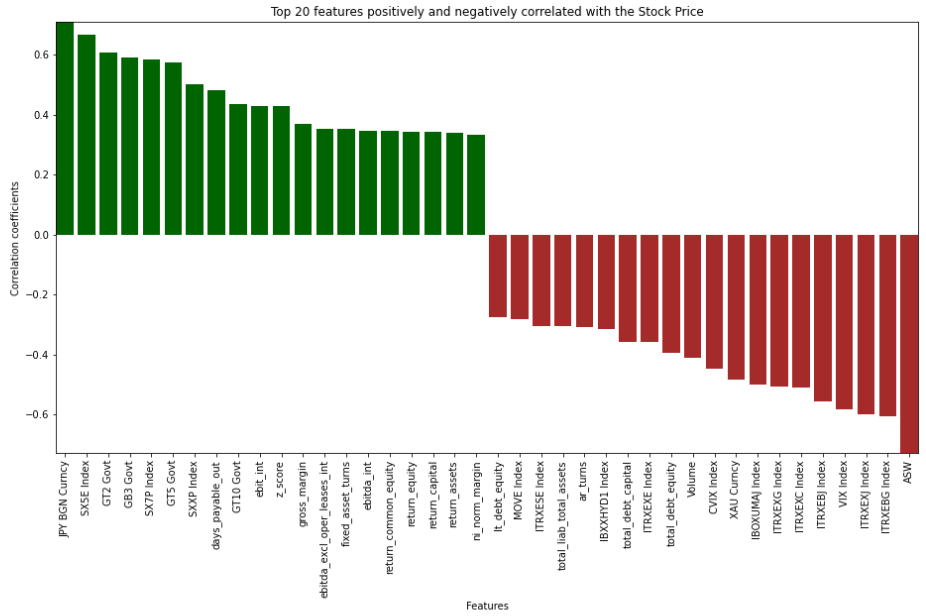
Stock: Deutsche Telekom



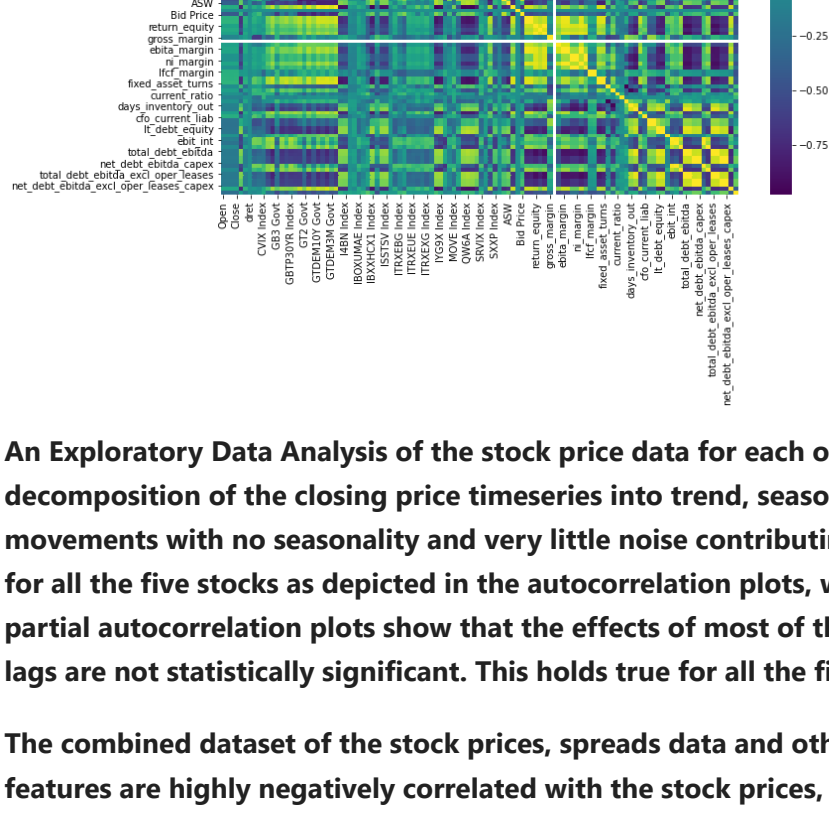


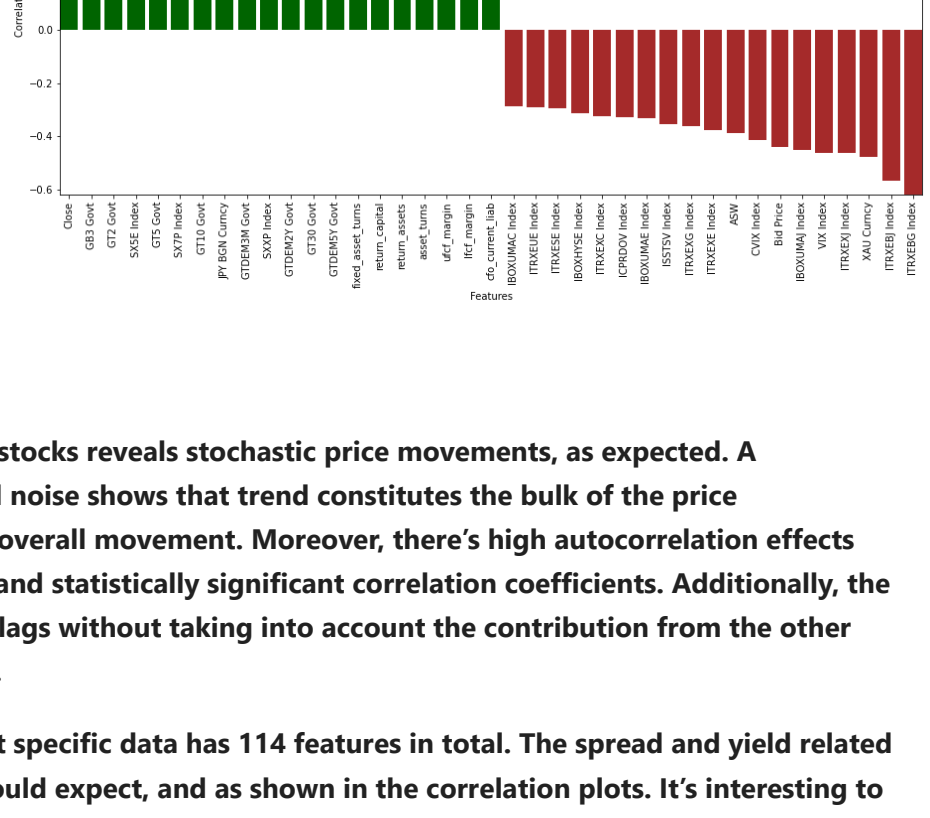
Stock: Renault



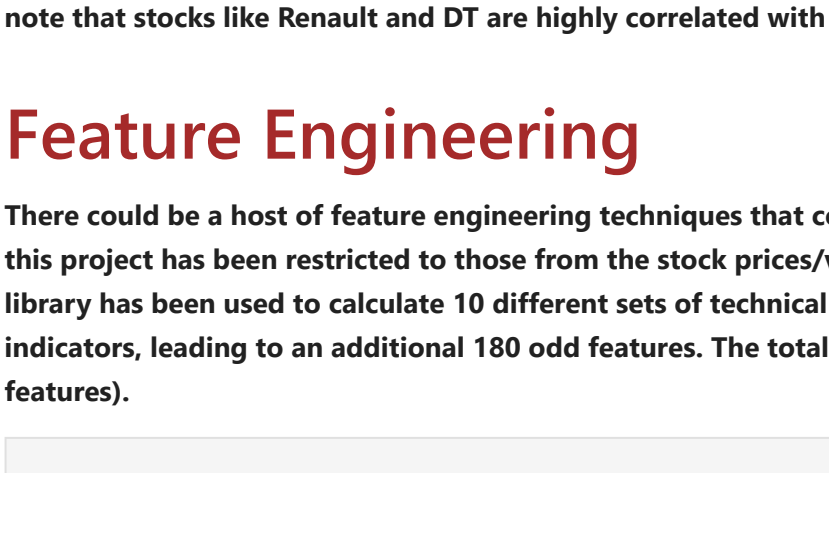


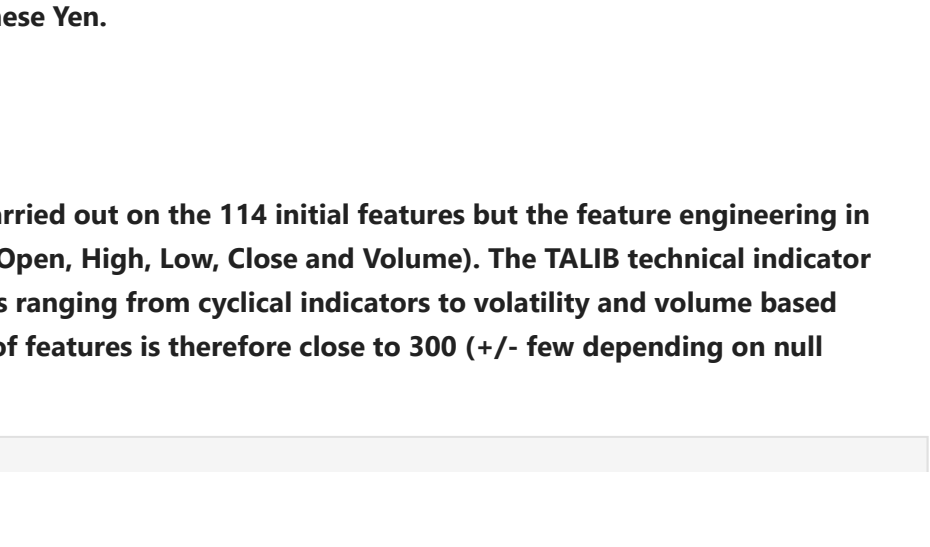
Stock: Siemens





Stock: Total Energies





An Exploratory Data Analysis of the stock price data for each of the five stocks reveals stochastic price movements, as expected. A decomposition of the closing price timeseries into trend, seasonality and noise shows that trend constitutes the bulk of the price movements with no seasonality and very little noise contributing to the overall movement. Moreover, there's high autocorrelation effects for all the five stocks as depicted in the autocorrelation plots, with high and statistically significant correlation coefficients. Additionally, the partial autocorrelation plots show that the effects of most of the higher lags without taking into account the contribution from the other lags are not statistically significant. This holds true for all the five stocks.

The combined dataset of the stock prices, spreads data and other market specific data has 114 features in total. The spread and yield related features are highly negatively correlated with the stock prices, as one would expect, and as shown in the correlation plots. It's interesting to note that stocks like Renault and DT are highly correlated with the Japanese Yen.

Feature Engineering

There could be a host of feature engineering techniques that could be carried out on the 114 initial features but the feature engineering in this project has been restricted to those from the stock prices/volumes (Open, High, Low, Close and Volume). The TALIB technical indicator library has been used to calculate 10 different sets of technical indicators ranging from cyclical indicators to volatility and volume based indicators, leading to an additional 180 odd features. The total number of features is therefore close to 300 (+/- few depending on null features).

In [345…

```
# Technical Indicators from TALIB

def TA(df):
    # Cyclical Indicators
    df['HT_DCPERIOD']=talib.HT_DCPERIOD(df.Close)
    df['HT_DCPHASE']=talib.HT_DCPHASE(df.Close)
    df['HT_PHASOR1'],df['HT_PHASOR2']=talib.HT_PHASOR(df.Close)
    df['HT_SINE1'],df['HT_SINE2']=talib.HT_SINE(df.Close)
    df['HT_TRENDMODE']=talib.HT_TRENDMODE(df.Close)

    # Math Operators
    df['ADD']=talib.ADD(df.High,df.Low)
    df['DIV']=talib.DIV(df.High,df.Low)
    df['MAX']=talib.MAX(df.Close,timeperiod=30)
    df['MAXINDEX']=talib.MAXINDEX(df.Close,timeperiod=30)
    df['MIN']=talib.MIN(df.Close,timeperiod=30)
    df['MININDEX']=talib.MININDEX(df.Close,timeperiod=30)
    df['MULT']=talib.MULT(df.High,df.Low)
    df['SUB']=talib.SUB(df.High,df.Low)
    df['SUM']=talib.SUM(df.Close,timeperiod=30)

    # Math Transform
    df['ACOS']=talib.ACOS(df.Close)
    df['ASIN']=talib.ASIN(df.Close)
    df['ATAN']=talib.ATAN(df.Close)
    df['CEIL']=talib.CEIL(df.Close)
    df['COS']=talib.COS(df.Close)
    df['COSH']=talib.COSH(df.Close)
    df['EXP']=talib.EXP(df.Close)
    df['FLOOR']=talib.FLOOR(df.Close)
    df['LN']=talib.LN(df.Close)
    df['LOG10']=talib.LOG10(df.Close)
    df['SIN']=talib.SIN(df.Close)
    df['SINH']=talib.SINH(df.Close)
    df['SQRT']=talib.SQRT(df.Close)
    df['TAN']=talib.TAN(df.Close)
    df['TANH']=talib.TANH(df.Close)

    # Momentum Indicators
    df['ADX']=talib.ADX(df.High,df.Low,df.Close,timeperiod=14)
    df['ADXR']=talib.ADXR(df.High,df.Low,df.Close,timeperiod=14)
    df['APO']=talib.APO(df.Close,fastperiod=12,slowperiod=26,matype=0)
    df['AROONDN'],df['AROONUP']=talib.AROON(df.High,df.Low,timeperiod=14)
    df['AROONOSC']=talib.AROONOSC(df.High,df.Low,timeperiod=14)
    df['BOP']=talib.BOP(df.Open,df.High,df.Low,df.Close)
    df['CCI']=talib.CCI(df.High,df.Low,df.Close,timeperiod=14)
    df['CMO']=talib.CMO(df.Close,timeperiod=14)
    df['DX']=talib.DX(df.High,df.Low,df.Close,timeperiod=14)
    df['MACD'],df['MACDSIGNAL'],df['MACDHIST']=talib.MACDEXT(df.Close,fastperiod=12,fastmatype=0,slowperiod=26,slowmatype=0,signal
    df['MFI']=talib.MFI(df.High,df.Low,df.Close,df.Volume,timeperiod=14)
    df['MINUS_DI']=talib.MINUS_DI(df.High,df.Low,df.Close,timeperiod=14)
    df['MINUS_DM']=talib.MINUS_DM(df.High,df.Low,timeperiod=14)
    df['MOM10']=talib.MOM(df.Close,timeperiod=10)
    df['MOM30']=talib.MOM(df.Close,timeperiod=30)
    df['MOM50']=talib.MOM(df.Close,timeperiod=50)
    df['PLUS_DI']=talib.PLUS_DI(df.High,df.Low,df.Close,timeperiod=14)
    df['PLUS_DM']=talib.PLUS_DM(df.High,df.Low,timeperiod=14)
    df['PPO']=talib.PPO(df.Close,fastperiod=12,slowperiod=26,matype=0)
    df['ROC']=talib.ROC(df.Close,timeperiod=10)
    df['ROCP']=talib.ROCP(df.Close,timeperiod=10)
    df['ROCR']=talib.ROCR(df.Close,timeperiod=10)
    df['ROCR100']=talib.ROCR100(df.Close,timeperiod=10)
    df['RSI']=talib.RSI(df.Close,timeperiod=14)
    df['SLOWK'],df['SLOWD']=talib.STOCH(df.High,df.Low,df.Close,fastk_period=5,slowk_period=3,slowk_matype=0,slowd_period=3,slowd_
    df['FASTK'],df['FASTD']=talib.STOCHF(df.High,df.Low,df.Close,fastk_period=5,fastd_period=3,fastd_matype=0)
    df['FASTK_'],df['FASTD_']=talib.STOCHRSI(df.Close,timeperiod=14,fastk_period=5,fastd_period=3,fastd_matype=0)
    df['TRIX']=talib.TRIX(df.Close,timeperiod=30)
    df['ULTOSC']=talib.ULTOSC(df.High,df.Low,df.Close,timeperiod1=7,timeperiod2=14,timeperiod3=28)
    df['WILLR']=talib.WILLR(df.High,df.Low,df.Close,timeperiod=14)

    # Overlap Studies
    df['BBU'],df['BBM'],df['BBL']=talib.BBANDS(df.Close,timeperiod=5,nbdevup=2,nbdevdn=2,matype=0)
    df['DEMA10']=talib.DEMA(df.Close,timeperiod=10)
    df['DEMA30']=talib.DEMA(df.Close,timeperiod=30)
    df['EMA5']=talib.EMA(df.Close,timeperiod=5)
    df['EMA10']=talib.EMA(df.Close,timeperiod=10)
    df['EMA30']=talib.EMA(df.Close,timeperiod=30)
    df['HT_TRENDLINE']=talib.HT_TRENDLINE(df.Close)
    df['KAMA10']=talib.KAMA(df.Close,timeperiod=10)
    df['KAMA30']=talib.KAMA(df.Close,timeperiod=30)
    df['MA5']=talib.MA(df.Close,timeperiod=5,matype=0)
    df['MA10']=talib.MA(df.Close,timeperiod=10,matype=0)
    df['MA30']=talib.MA(df.Close,timeperiod=30,matype=0)
    df['MAMA'],df['FAMA']=talib.MAMA(df.Close)
    df['MIDPOINT']=talib.MIDPOINT(df.Close,timeperiod=14)
    df['MIDPRICE']=talib.MIDPRICE(df.High,df.Low,timeperiod=14)
    df['SAR']=talib.SAR(df.High,df.Low,acceleration=0,maximum=0)
    df['SMA5']=talib.SMA(df.Close,timeperiod=5)
    df['SMA10']=talib.SMA(df.Close,timeperiod=10)
    df['SMA30']=talib.SMA(df.Close,timeperiod=30)
    df['T3']=talib.T3(df.Close,timeperiod=5,vfactor=0)
    df['TEMA10']=talib.TEMA(df.Close,timeperiod=10)
    df['TEMA30']=talib.TEMA(df.Close,timeperiod=30)
    df['TRIMA10']=talib.TRIMA(df.Close,timeperiod=10)
```

```
df['TRIMA30']=talib.TRIMA(df.Close,timeperiod=30)
df['WMA10']=talib.WMA(df.Close,timeperiod=10)
df['WMA30']=talib.WMA(df.Close,timeperiod=30)

# Pattern Recognition
df['CDL2CROWS']=talib.CDL2CROWS(df.Open,df.High,df.Low,df.Close)
df['CDL3BLACKCROWS']=talib.CDL3BLACKCROWS(df.Open,df.High,df.Low,df.Close)
df['CDL3INSIDE']=talib.CDL3INSIDE(df.Open,df.High,df.Low,df.Close)
df['CDL3LINESTRIKE']=talib.CDL3LINESTRIKE(df.Open,df.High,df.Low,df.Close)
df['CDL3OUTSIDE']=talib.CDL3OUTSIDE(df.Open,df.High,df.Low,df.Close)
df['CDL3STARSINSOUTH']=talib.CDL3STARSINSOUTH(df.Open,df.High,df.Low,df.Close)
df['CDL3WHITESOLDIERS']=talib.CDL3WHITESOLDIERS(df.Open,df.High,df.Low,df.Close)
df['CDLABANDONEDBABY']=talib.CDLABANDONEDBABY(df.Open,df.High,df.Low,df.Close,penetration=0)
df['CDLADVANCEBLOCK']=talib.CDLADVANCEBLOCK(df.Open,df.High,df.Low,df.Close)
df['CDLBELTHOLD']=talib.CDLBELTHOLD(df.Open,df.High,df.Low,df.Close)
df['CDLBREAKAWAY']=talib.CDLBREAKAWAY(df.Open,df.High,df.Low,df.Close)
df['CDLCLOSINGMARUBOZU']=talib.CDLCLOSINGMARUBOZU(df.Open,df.High,df.Low,df.Close)
df['CDLCONCEALBABYSWALL']=talib.CDLCONCEALBABYSWALL(df.Open,df.High,df.Low,df.Close)
df['CDLCOUNTERATTACK']=talib.CDLCOUNTERATTACK(df.Open,df.High,df.Low,df.Close)
df['CDLDARKCLOUDCOVER']=talib.CDLDARKCLOUDCOVER(df.Open,df.High,df.Low,df.Close,penetration=0)
df['CDLDOJI']=talib.CDLDOJI(df.Open,df.High,df.Low,df.Close)
df['CDLDOJISTAR']=talib.CDLDOJISTAR(df.Open,df.High,df.Low,df.Close)
df['CDLDRAGONFLYDOJI']=talib.CDLDRAGONFLYDOJI(df.Open,df.High,df.Low,df.Close)
df['CDLENGULFING']=talib.CDLENGULFING(df.Open,df.High,df.Low,df.Close)
df['CDLEVENINGDOJISTAR']=talib.CDLEVENINGDOJISTAR(df.Open,df.High,df.Low,df.Close,penetration=0)
df['CDLEVENINGSTAR']=talib.CDLEVENINGSTAR(df.Open,df.High,df.Low,df.Close,penetration=0)
df['CDLGAPSIDESIDEWHITE']=talib.CDLGAPSIDESIDEWHITE(df.Open,df.High,df.Low,df.Close)
df['CDLGRAVESTONEDOJI']=talib.CDLGRAVESTONEDOJI(df.Open,df.High,df.Low,df.Close)
df['CDLHAMMER']=talib.CDLHAMMER(df.Open,df.High,df.Low,df.Close)
df['CDLHANGINGMAN']=talib.CDLHANGINGMAN(df.Open,df.High,df.Low,df.Close)
df['CDLHARAMI']=talib.CDLHARAMI(df.Open,df.High,df.Low,df.Close)
df['CDLHARAMICROSS']=talib.CDLHARAMICROSS(df.Open,df.High,df.Low,df.Close)
df['CDLHIGHWAVE']=talib.CDLHIGHWAVE(df.Open,df.High,df.Low,df.Close)
df['CDLHIKKAKE']=talib.CDLHIKKAKE(df.Open,df.High,df.Low,df.Close)
df['CDLHIKKAKEMOD']=talib.CDLHIKKAKEMOD(df.Open,df.High,df.Low,df.Close)
df['CDLHOMINGPIGEON']=talib.CDLHOMINGPIGEON(df.Open,df.High,df.Low,df.Close)
df['CDLIDENTICAL3CROWS']=talib.CDLIDENTICAL3CROWS(df.Open,df.High,df.Low,df.Close)
df['CDLINNECK']=talib.CDLINNECK(df.Open,df.High,df.Low,df.Close)
df['CDLINVERTEDHAMMER']=talib.CDLINVERTEDHAMMER(df.Open,df.High,df.Low,df.Close)
df['CDLKICKING']=talib.CDLKICKING(df.Open,df.High,df.Low,df.Close)
df['CDLKICKINGBYLENGTH']=talib.CDLKICKINGBYLENGTH(df.Open,df.High,df.Low,df.Close)
df['CDLLADDERBOTTOM']=talib.CDLLADDERBOTTOM(df.Open,df.High,df.Low,df.Close)
df['CDLLONGLEGGEDDOJI']=talib.CDLLONGLEGGEDDOJI(df.Open,df.High,df.Low,df.Close)
df['CDLLONGLINE']=talib.CDLLONGLINE(df.Open,df.High,df.Low,df.Close)
df['CDLMARUBOZU']=talib.CDLMARUBOZU(df.Open,df.High,df.Low,df.Close)
df['CDLMATCHINGLOW']=talib.CDLMATCHINGLOW(df.Open,df.High,df.Low,df.Close)
df['CDLMATHOLD']=talib.CDLMATHOLD(df.Open,df.High,df.Low,df.Close,penetration=0)
df['CDLMORNINGDOJISTAR']=talib.CDLMORNINGDOJISTAR(df.Open,df.High,df.Low,df.Close,penetration=0)
df['CDLMORNINGSTAR']=talib.CDLMORNINGSTAR(df.Open,df.High,df.Low,df.Close,penetration=0)
df['CDLONNECK']=talib.CDLONNECK(df.Open,df.High,df.Low,df.Close)
df['CDLPIERCING']=talib.CDLPIERCING(df.Open,df.High,df.Low,df.Close)
df['CDLRICKSHAWMAN']=talib.CDLRICKSHAWMAN(df.Open,df.High,df.Low,df.Close)
df['CDLRISEFALL3METHODS']=talib.CDLRISEFALL3METHODS(df.Open,df.High,df.Low,df.Close)
df['CDLSEPARATINGLINES']=talib.CDLSEPARATINGLINES(df.Open,df.High,df.Low,df.Close)
df['CDLSHOOTINGSTAR']=talib.CDLSHOOTINGSTAR(df.Open,df.High,df.Low,df.Close)
df['CDLSHORTLINE']=talib.CDLSHORTLINE(df.Open,df.High,df.Low,df.Close)
df['CDLSPINNINGTOP']=talib.CDLSPINNINGTOP(df.Open,df.High,df.Low,df.Close)
df['CDLSTICKSANDWICH']=talib.CDLSTICKSANDWICH(df.Open,df.High,df.Low,df.Close)
df['CDLTAKURI']=talib.CDLTAKURI(df.Open,df.High,df.Low,df.Close)
df['CDLTASUKIGAP']=talib.CDLTASUKIGAP(df.Open,df.High,df.Low,df.Close)
df['CDLTHRUSTING']=talib.CDLTHRUSTING(df.Open,df.High,df.Low,df.Close)
df['CDLTRISTAR']=talib.CDLTRISTAR(df.Open,df.High,df.Low,df.Close)
df['CDLUNIQUE3RIVER']=talib.CDLUNIQUE3RIVER(df.Open,df.High,df.Low,df.Close)
df['CDLUPSIDEGAP2CROWS']=talib.CDLUPSIDEGAP2CROWS(df.Open,df.High,df.Low,df.Close)
df['CDLXSIDEGAP3METHODS']=talib.CDLXSIDEGAP3METHODS(df.Open,df.High,df.Low,df.Close)

# Price Transform
df['AVGPRICE']=talib.AVGPRICE(df.Open,df.High,df.Low,df.Close)
df['MEDPRICE']=talib.MEDPRICE(df.High,df.Low)
df['TYPPRICE']=talib.TYPPRICE(df.High,df.Low,df.Close)
df['WCLPRICE']=talib.WCLPRICE(df.High,df.Low,df.Close)

# Statistic Functions
df['BETA']=talib.BETA(df.High,df.Low,timeperiod=5)
df['CORREL10']=talib.CORREL(df.High,df.Low,timeperiod=10)
df['CORREL30']=talib.CORREL(df.High,df.Low,timeperiod=30)
df['LINEARREG']=talib.LINEARREG(df.Close,timeperiod=14)
df['LINEARREG_ANGLE']=talib.LINEARREG_ANGLE(df.Close,timeperiod=14)
df['LINEARREG_INTERCEPT']=talib.LINEARREG_INTERCEPT(df.Close,timeperiod=14)
df['LINEARREG_SLOPE']=talib.LINEARREG_SLOPE(df.Close,timeperiod=14)
df['STDDEV']=talib.STDDEV(df.Close,timeperiod=5,nbdev=1)
df['TSF']=talib.TSF(df.Close,timeperiod=14)
df['VAR']=talib.VAR(df.Close,timeperiod=5,nbdev=1)

# Volatility Indicators
df['ATR5']=talib.ATR(df.High,df.Low,df.Close,timeperiod=5)
df['ATR10']=talib.ATR(df.High,df.Low,df.Close,timeperiod=10)
df['ATR20']=talib.ATR(df.High,df.Low,df.Close,timeperiod=20)
df['ATR30']=talib.ATR(df.High,df.Low,df.Close,timeperiod=30)
df['ATR50']=talib.ATR(df.High,df.Low,df.Close,timeperiod=50)
df['ATR100']=talib.ATR(df.High,df.Low,df.Close,timeperiod=100)
df['NATR5']=talib.NATR(df.High,df.Low,df.Close,timeperiod=5)
df['NATR10']=talib.NATR(df.High,df.Low,df.Close,timeperiod=10)
df['NATR20']=talib.NATR(df.High,df.Low,df.Close,timeperiod=20)
```

```
    df['NATR30']=talib.NATR(df.High,df.Low,df.Close,timeperiod=30)
    df['NATR50']=talib.NATR(df.High,df.Low,df.Close,timeperiod=50)
    df['NATR100']=talib.NATR(df.High,df.Low,df.Close,timeperiod=100)
    df['TRANGE']=talib.TRANGE(df.High,df.Low,df.Close)

    # Volume Indicators
    df['AD']=talib.AD(df.High,df.Low,df.Close,df.Volume)
    df['ADOSC']=talib.ADOSC(df.High,df.Low,df.Close,df.Volume,fastperiod=3,slowperiod=10)
    df['OBV']=talib.OBV(df.Close,df.Volume)

    return df
```

In [346…

```
for ticker in ytickers:
    stocks[ticker]=TA(stocks[ticker].copy())

    stocks[ticker]=stocks[ticker].replace(np.inf,np.nan).dropna(how='all',axis=1)

    imputer=impute.SimpleImputer().fit(stocks[ticker])
    stocks[ticker]=pd.DataFrame(imputer.transform(stocks[ticker]),columns=stocks[ticker].columns,index=stocks[ticker].index)
```

In [347…

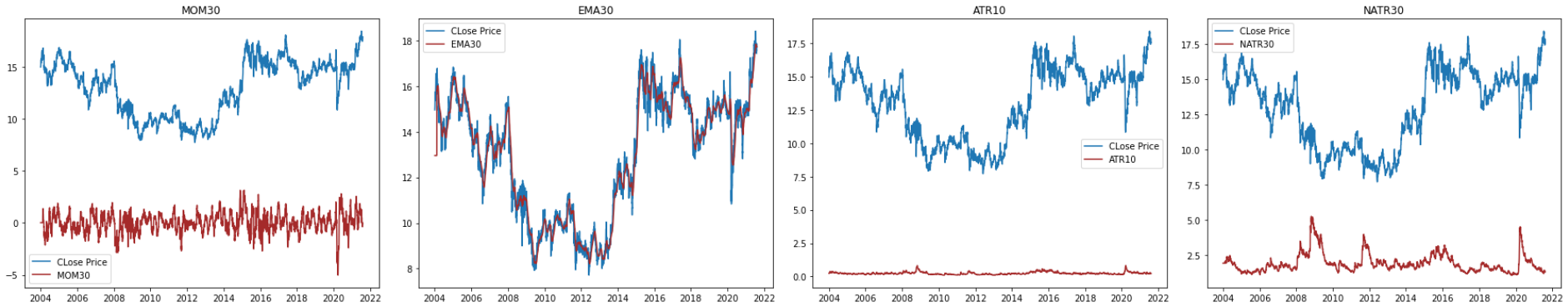
```
class TALIB_EDA:

    def __init__(self,df,ticker,features,nr,nc):
        self.df=df.copy()
        self.ticker=ticker
        self.features=features
        self.nr=nr
        self.nc=nc

    def _plot(self,figsize=(25,6)):
        fig,ax=plt.subplots(self.nr,self.nc,figsize=figsize)
        print('\n')
        print('Stock: '+self.ticker)

        k=0

        if self.nr>1:
            for i in range(self.nr):
                for j in range(self.nc):
                    if k<len(self.features):
                        ax[i][j].set_title(self.features[k])
                        ax[i][j].plot(self.df.index,self.df.Close,label='CLose Price')
                        ax[i][j].plot(self.df.index,self.df[self.features[k]],color='brown',label=self.features[k])
                        ax[i][j].legend()
                        k+=1
        else:
            for j in range(self.nc):
                if k<len(self.features):
                    ax[j].set_title(self.features[k])
                    ax[j].plot(self.df.index,self.df.Close,label='CLose Price')
                    ax[j].plot(self.df.index,self.df[self.features[k]],color='brown',label=self.features[k])
                    ax[j].legend()
                    k+=1

        plt.tight_layout()
        plt.show()
```

In [348…

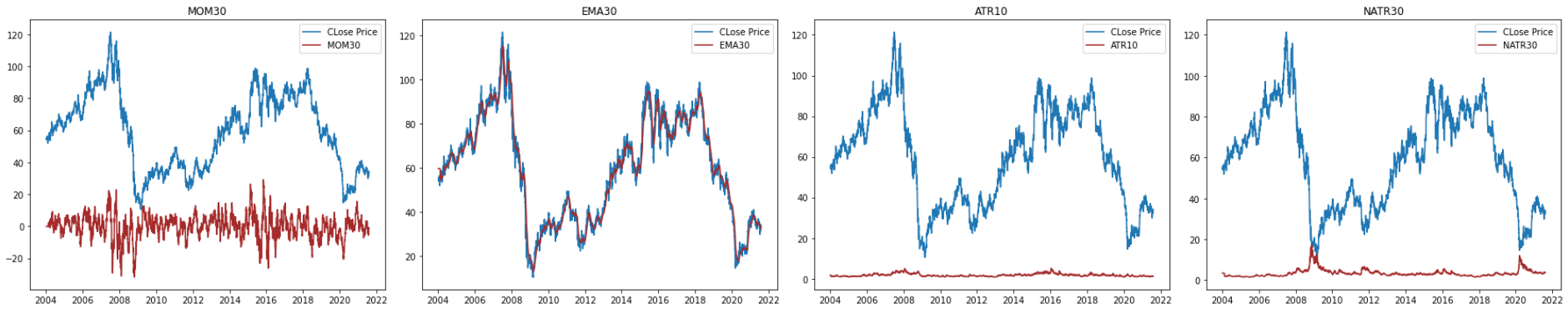
```
#Plots of a few Technical Indicators for each stock

TALIB_EDA(stocks['BATS.L'].copy(),'British American Tobacco',['MOM30','EMA30','ATR10','NATR30'],1,4)._plot(figsize=(25,5))
TALIB_EDA(stocks['DTE.DE'].copy(),'Deutsche Telekom',['MOM30','EMA30','ATR10','NATR30'],1,4)._plot(figsize=(25,5))
TALIB_EDA(stocks['RNO.PA'].copy(),'Renault',['MOM30','EMA30','ATR10','NATR30'],1,4)._plot(figsize=(25,5))
TALIB_EDA(stocks['SIE.DE'].copy(),'Siemens',['MOM30','EMA30','ATR10','NATR30'],1,4)._plot(figsize=(25,5))
TALIB_EDA(stocks['TTE.PA'].copy(),'Total Energies',['MOM30','EMA30','ATR10','NATR30'],1,4)._plot(figsize=(25,5))
```

```
Stock: British American Tobacco
```

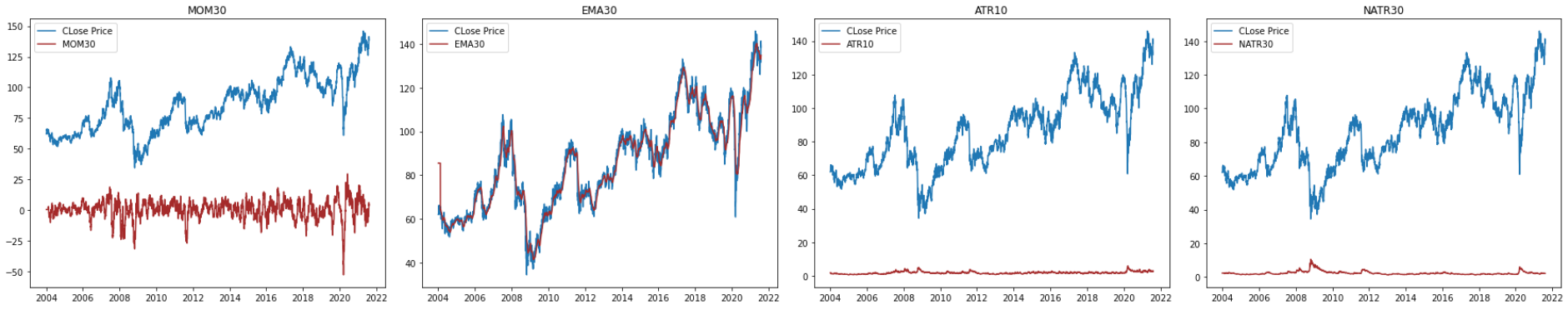

```
Stock: Deutsche Telekom
```



Stock: Renault



Stock: Siemens



Stock: Total Energies

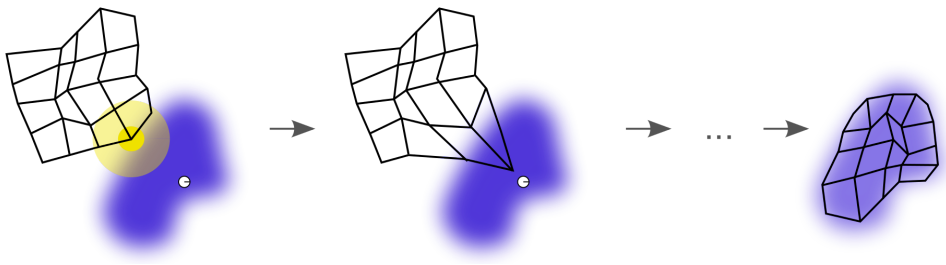
It might not be feasible to check each of the 180 technical indicators and do EDA on each. A larger set of features might also lead to what is called the 'Curse of Dimensionality', which necessitates the need for Feature Selection techniques.

Feature Selection

Self Organising Maps

SOM is an unsupervised deep learning technique which projects a high dimensional dataset into a low dimensional space without changing the topographical structure of the data. It's based on the idea of competitive learning as opposed to the error-based learning of the supervised neural network structures. The SOM algorithm is as follows:

- a. Select an NxN map and randomize the node weight vectors in the map.
- b. Randomly pick an input sample and traverse each node in the map.
- c. Find a Best Matching Unit (BMU) neuron that produces the least (Euclidean) distance between the input vector and the weight vector.
- d. Update the weights of the nodes nearest to the BMU.
- e. Repeat until threshold criteria is met (number of iterations/convergence).



By Mcld - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=10373592

SOM: Mathematical Notations

Distance of inputs from the weight vectors: $d_{ij} = \sum (w_{ij} - x_i)^2$

Weights are updated as: $w_{ij}(t+1) = w_{ij}(t) + \theta(t) * \alpha(t) * (x_i(t) - w_{ij}(t))$

where, x is the input vector, α is the learning rate and θ is the neighbourhood function

The SOM algorithm was used to select a total of around 50 features from the initial 300. For selecting the features, the weight vectors across the 2 dimensional map of each feature within each feature set was compared and a process of recursive feature elimination was carried out when similar mapping was observed. For instance, the features 'Open' and 'Close' have very similar mappings and only out of the two was kept.

In [349…

```
#Custom SOM class based on the Minisom library

class _SOM:
    def __init__(self,df,units):
        self.df=df.copy()
        self.X=preprocessing.scale(df.copy())
        self.units=units

        som=MiniSom(self.units,self.units,len(self.X[0]),neighborhood_function='gaussian',random_seed=2021)
        som.random_weights_init(self.X)
        som.train_random(self.X,100,verbose=True)

        self.W=som.get_weights()

    @property
    def _get_weights(self):
        return self.W

    def _features(self,cols,threshold=0.5):

        """
        Features are selected based on the similarity mapping of the weight vectors for each feature.
        The norm of the differences between two weight vectors is taken as a proxy for similarity.
        A threshold of 0.5 for the norm difference is taken as default, below which one feature out of the two is eliminated.

        """

        combs=[c for c in itertools.combinations(cols,2)]
        features=cols.copy()

        norms=np.array([np.linalg.norm(self.W[:,:,cols.index(c[0])]-self.W[:,:,cols.index(c[1])]) for c in combs])
        _max=max(norms)
        _min=min(norms)
        norms=list([(n-_min)/(_max-_min) for n in (norms)])

        for i,c in enumerate(combs):

            if norms[i]<threshold and c[1] in features:
                features.remove(c[1])

        return features
```

In [374…

```
features_som={}

for ticker in ytickers:

    print('\nTicker: ',ticker)

    df=stocks[ticker].copy()
    cols=df.columns.tolist()

    sm=_SOM(df.copy(),units=100)

    features1=sm._features(cols=cols[:7]) #Stock price/Volume
    features2=sm._features(cols=cols[7:63]) #Market data
    features3=sm._features(cols=cols[63:67]) #Bond specific data (spreads/yields)
    features4=sm._features(cols=cols[67:114]) #Fin ratios
    features5=sm._features(cols=cols[114:]) #Technical indicators from TALIB

    features_som[ticker]=features1+features2+features3+features4+features5

    if 'dret' not in features_som[ticker]:
        features_som[ticker]=features_som[ticker]+['dret']

print('\n')
```

```
Ticker:  BATS.L
 [ 100 / 100 ] 100% - 0:00:00 left
 quantization error: 0.7497842229086239

Ticker:  DTE.DE
 [ 100 / 100 ] 100% - 0:00:00 left
 quantization error: 0.7732178747017653

Ticker:  RNO.PA
 [ 100 / 100 ] 100% - 0:00:00 left
 quantization error: 0.76440515371113

Ticker:  SIE.DE
 [ 100 / 100 ] 100% - 0:00:00 left
```

```
 quantization error: 0.7787193611421842

Ticker:  TTE.PA
 [ 100 / 100 ] 100% - 0:00:00 left
 quantization error: 0.7638896188635578
```

In [375…

```
som_features=pd.DataFrame(features_som.items(),columns=['Stock','No. of SOM Selected features'])
som_features['No. of SOM Selected features']=som_features['No. of SOM Selected features'].apply(lambda x:len(x))
som_features['Total features']=som_features['Stock'].apply(lambda x:len(stocks[x].columns))
som_features.style.background_gradient(cmap=sns.color_palette('RdYlGn_r',as_cmap=True),axis=None)
```

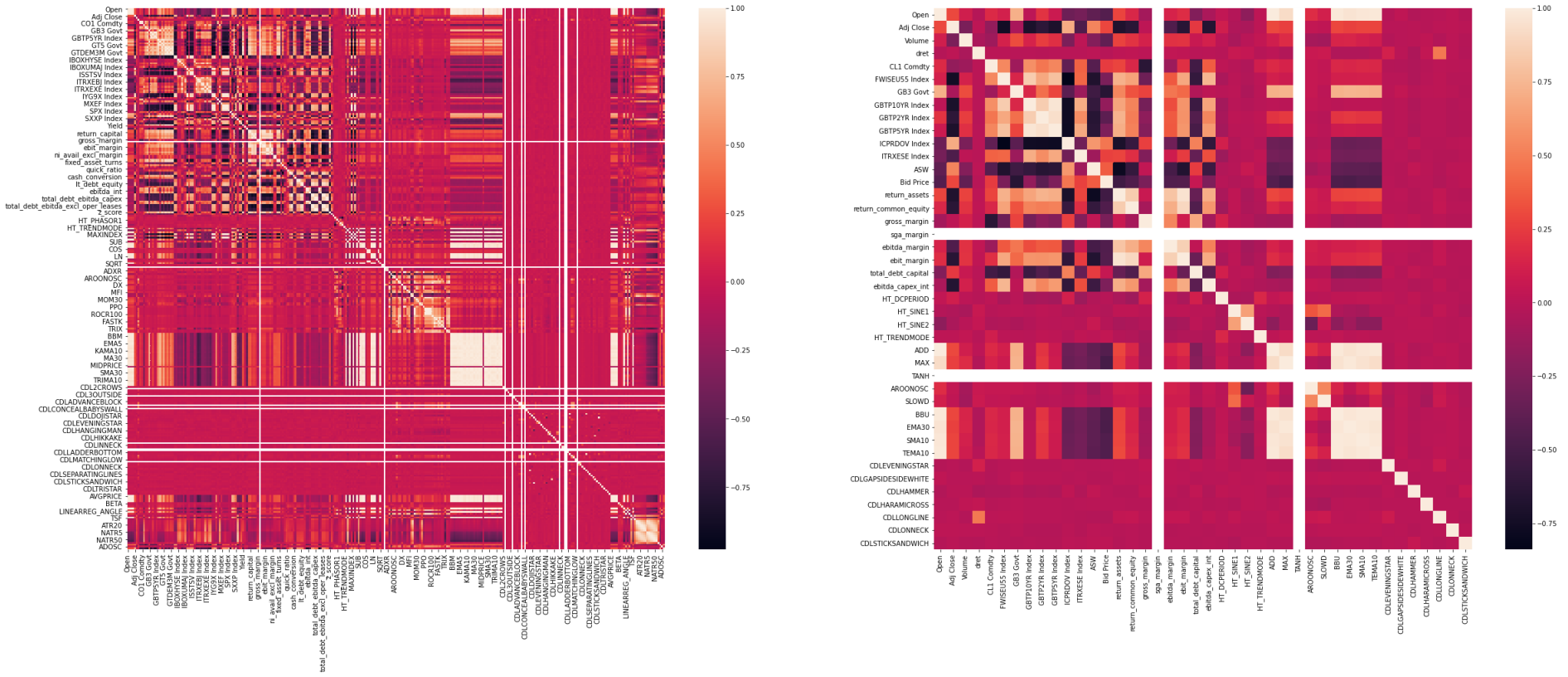
Out[375…

	Stock	No. of SOM Selected features	Total features
0	BATS.L	52	295
1	DTE.DE	50	298
2	RNO.PA	48	298
3	SIE.DE	48	298
4	TTE.PA	42	298

In [352…

```
fig,ax=plt.subplots(1,2,figsize=(40,15))

print('\nCorrealtion Heatmaps before and after feature selection\n')

ticker=='DTE.DE'
sns.heatmap(stocks[ticker].corr(),ax=ax[0])
sns.heatmap(stocks[ticker][features_som[ticker]].corr(),ax=ax[1])

plt.show()
```

```
Correaltion Heatmaps before and after feature selection
```

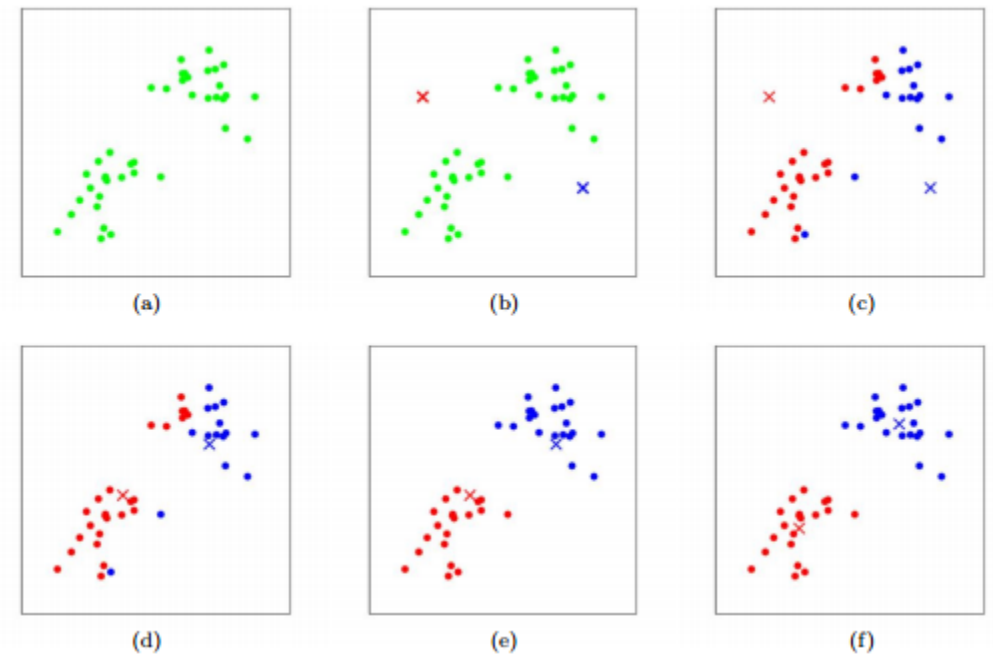


K-Means Clustering

The K-Means Clustering technique is a very simple and one of the most widely used unsupervised machine learning techniques used to group similarly structured data samples into clusters. The idea is to select a total ok k centroids which are used to define the k clusters. A point belongs to a particular cluster if its distance to the centroid of that cluster is the least compared to any other centroids.

The KMC algorithm is as follows:

- Randomly select cluster centroids, $\mu_1, \mu_2, \mu_3, \ldots, \mu_k \in \mathbb{R}^n$
- Repeat until convergence:
  - $c^{(i)} := \arg\min_j ||x^{(i)} - \mu_j||^2$
  - $\mu_j := \frac{\sum_{i=1}^{m} \mathbf{1}\{c^{(i)}=j\} x_i}{\sum_{i=1}^{m} \mathbf{1}\{c^{(i)}=j\}}$



From: https://stanford.edu/~cpiech/cs221/handouts/kmeans.html

Similar to the SOM feature selection technique, the KMC technique was used to reduce the 300 features into 40-50 based on the elbow plots. Similar features which might have similar effects on the target variables are grouped into the same clusters and the feature with the smallest distance to the cluster centroid is chosen as the only feature from a particular cluster. The KMC feature selection was carried out separately for each of the five feature sets leading to five different elbow plots as shown below.

In [353…

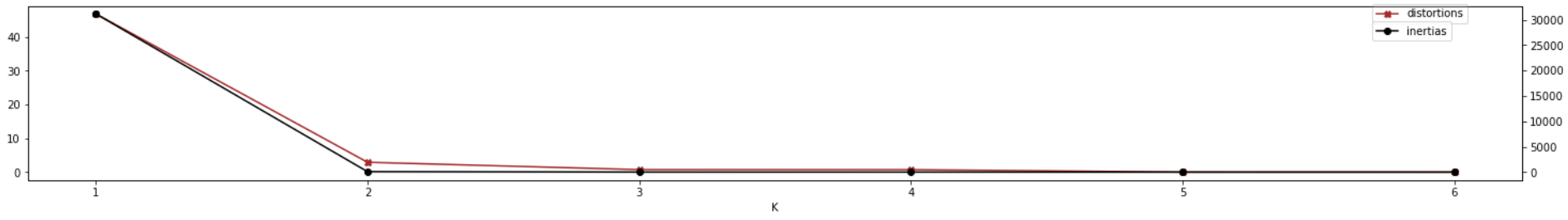
```
#Custom KMeans class based on the sklearn library

class _KMeans:
    def __init__(self,df,k):
        self.df=df
        self.X=preprocessing.scale(df.T.fillna(999))
        self.k=k
        self.cols=df.columns.tolist()

    def _fit(self):
        self.kmeans=cluster.KMeans(n_clusters=self.k,init='random',max_iter=100,random_state=21)
        self.kmeans.fit(self.X)

    @property
    def _inertia(self):
        return self.kmeans.inertia_

    @property
    def _distortion(self):
        return sum(np.min(cdist(self.X,self.kmeans.cluster_centers_,metric='euclidean'),axis=1))/self.X.shape[0]

    def _elbow_plot(self,distortions,inertias,Ks,figsize=(25,3)):
        fig,ax=plt.subplots(figsize=figsize)
        ax.plot(Ks,distortions,color='brown',marker='X',label='distortions')
        ax_=ax.twinx()
        ax_.plot(Ks,inertias,color='k',marker='o',label='inertias')
        ax.set_xlabel('K')
        ax.legend(loc=(0.9,0.9))
        ax_.legend(loc=(0.9,0.8))
        plt.show()
        return fig

    @property
    def _features(self):

        """
        Features are selected based on the distance of a particular feature to the cluster centroid.
        The one with the least distance is chosen from a particular cluster.

        """
        self._fit()
        feat=pd.DataFrame({'feature':self.df.T.index.tolist()})
        feat['cluster']=self.kmeans.labels_
        feat['dist']=np.min(cdist(self.X,self.kmeans.cluster_centers_,metric='euclidean'),axis=1)
        feat=feat.sort_values(by='dist',ascending=False).drop_duplicates(subset='cluster',keep='last')
        return feat.feature.tolist()


def km_plots(ticker):

    df=stocks[ticker].copy()
    cols=df.columns.tolist()

    f1=cols[:7] #Stock price/Volume
    f2=cols[7:63] #Market data
    f3=cols[63:67] #Bond specific data (spreads/yields)
    f4=cols[67:114] #Fin ratios
    f5=cols[114:] #Technical indicators from TALIB

    print('Stock :',ticker)
```

```
    for i,f in enumerate([f1,f2,f3,f4,f5]):

        distortions=[]
        inertias=[]

        Ks=np.arange(1,len(f)) if len(f)<30 else np.arange(1,30)

        print('\n')
        print('Feature set ',i+1)

        for k in (Ks):

            km=_KMeans(df[f].copy(),k)
            km._fit()
            distortions.append(km._distortion)
            inertias.append(km._inertia)

        km._elbow_plot(distortions,inertias,Ks)

def get_ks(ticker,threshold):

    t=[]

    df=stocks[ticker].copy()
    cols=df.columns.tolist()

    f1=cols[:7] #Stock price/Volume
    f2=cols[7:63] #Market data
    f3=cols[63:67] #Bond specific data (spreads/yields)
    f4=cols[67:114] #Fin ratios
    f5=cols[114:] #Technical indicators from TALIB

    for i,f in enumerate([f1,f2,f3,f4,f5]):

        distortions=[]
        inertias=[]

        deltas=[]

        Ks=np.arange(1,len(f)) if len(f)<30 else np.arange(1,30)

        for k in (Ks):

            km=_KMeans(df[f].copy(),k)
            km._fit()
            distortions.append(km._distortion)
            inertias.append(km._inertia)

        deltas=list(-np.diff(distortions))
        #print(deltas)
        try:
            t.append(deltas.index(max([x for x in deltas if x<threshold]))) #selecting the K for which the decrease is below the s
        except:
            t.append(Ks[-1])
    return t
```

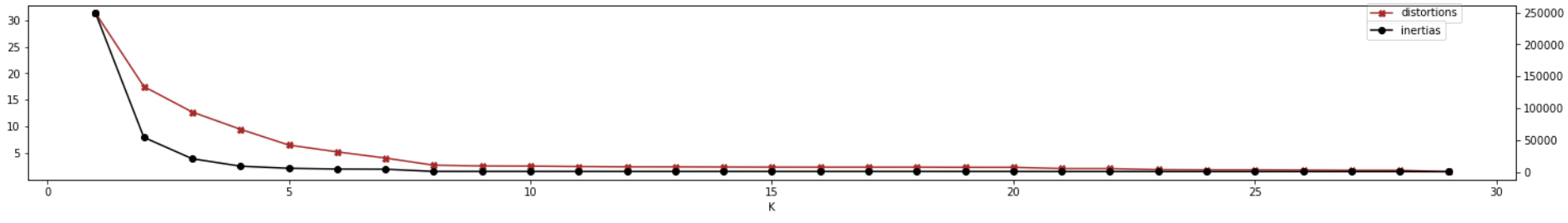
In [354…

```
#Visualizing the elbow plots for select stocks

km_plots('BATS.L')
```

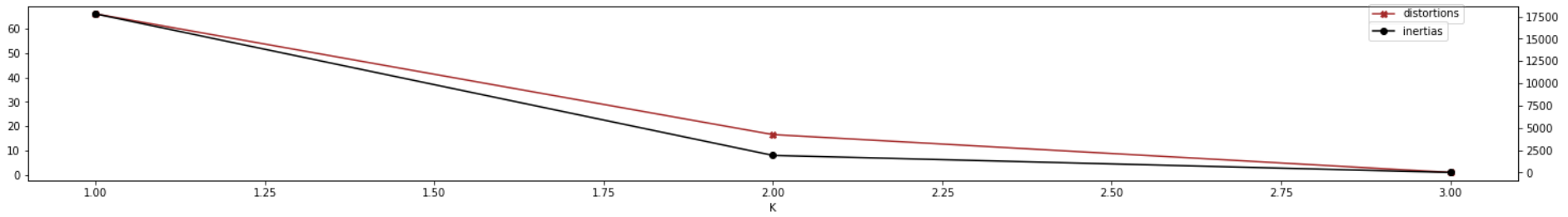
```
Stock : BATS.L


Feature set  1
```



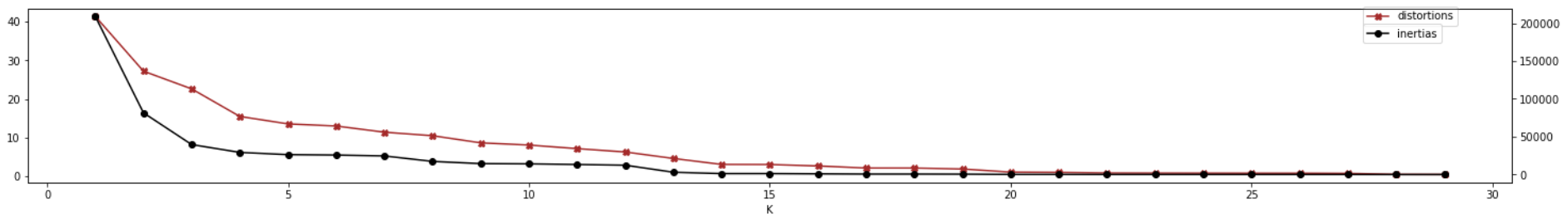

```
Feature set  2
```



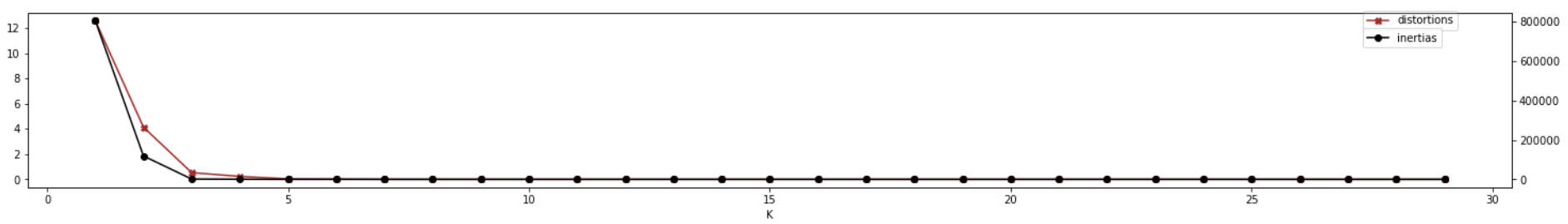

```
Feature set  3
```



Feature set 4



Feature set 5



For each of the 5 stocks, features within the feature set 5 (technical indicators) are closely related to one another, leading to very few clusters.

In [355...

```
features_km={}

for ticker in ytickers:

    k=get_ks(ticker,0.01)#No of clusters to be chosen for each feature set

    df=stocks[ticker].copy()
    cols=df.columns.tolist()

    features1=_KMeans(df[cols[:7]].copy(),k[0])._features #Stock price/Volume
    features2=_KMeans(df[cols[7:63]].copy(),k[1])._features #Market data
    features3=_KMeans(df[cols[63:67]].copy(),k[2])._features #Bond specific data (spreads/yields)
    features4=_KMeans(df[cols[67:114]].copy(),k[3])._features #Fin ratios
    features5=_KMeans(df[cols[114:]].copy(),k[4])._features #Technical indicators from TALIB

    features_km[ticker]=features1+features2+features3+features4+features5

    if 'dret' not in features_km[ticker]:
        features_km[ticker]=features_km[ticker]+['dret']

print('Done!\n')
```

```
Done!
```

In [356...

```
km_features=pd.DataFrame(features_km.items(),columns=['Stock','No. of KMC Selected features'])
km_features['No. of KMC Selected features']=km_features['No. of KMC Selected features'].apply(lambda x:len(x))
km_features['Total features']=km_features['Stock'].apply(lambda x:len(stocks[x].columns))
km_features.style.background_gradient(cmap=sns.color_palette('RdYlGn_r',as_cmap=True),axis=None)
```

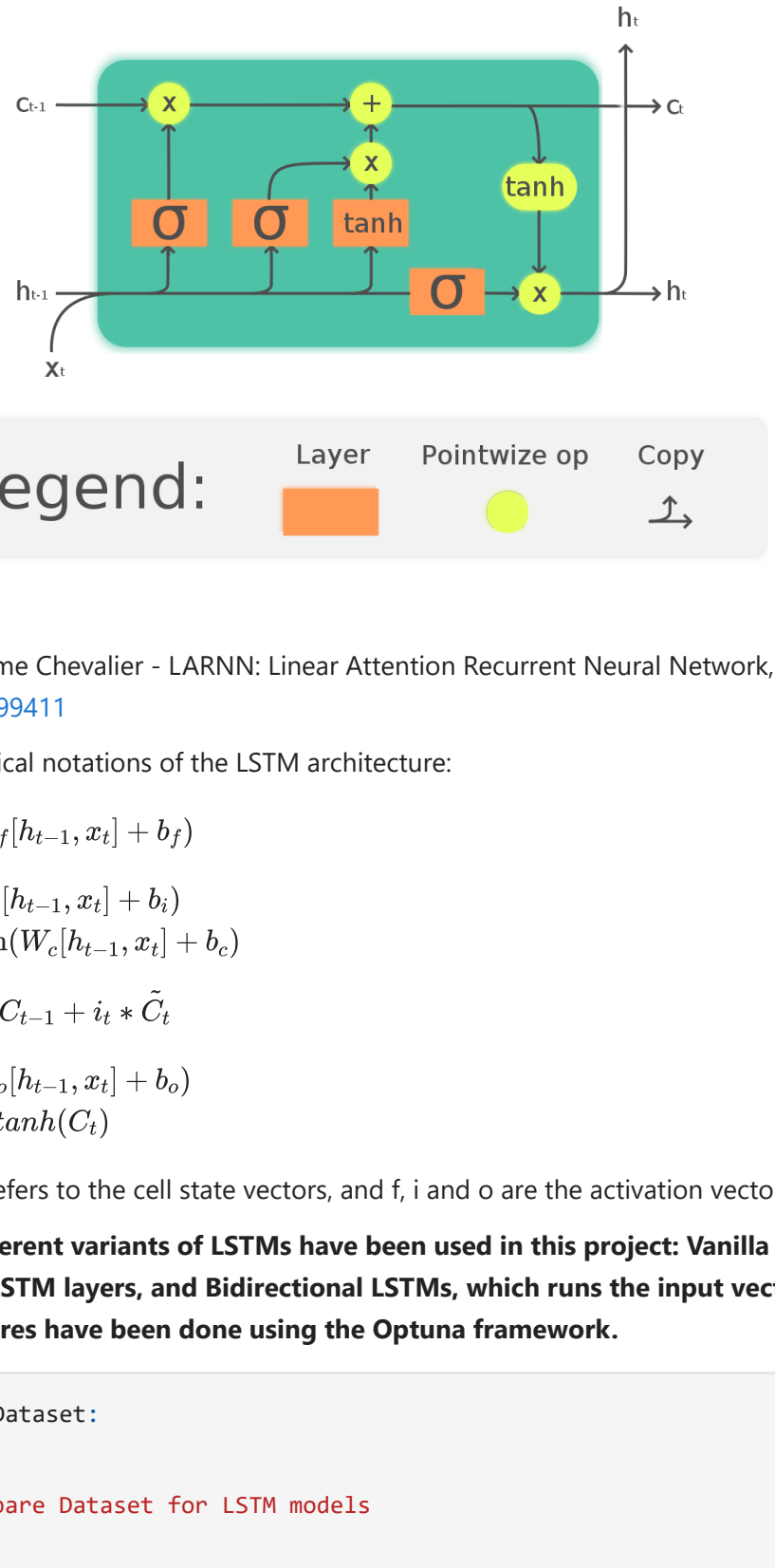
Out[356...

	Stock	No. of KMC Selected features	Total features
0	BATS.L	43	295
1	DTE.DE	45	298
2	RNO.PA	51	298
3	SIE.DE	38	298
4	TTE.PA	46	298

Deep Neural Nets

LSTMs

The LSTMs are a special category of Recurrent Neural Networks (RNNs), which are designed to preserve long sequence information, and could be especially useful in time series problems. RNNs often have the limitation of the vanishing gradient problem which leads to very slow convergence. LSTMs solve this issue by preserving information over arbitrary time intervals and regulating the flow of information in and out of the cell. This is done with the help of three gates in the LSTM cell: the input gate, the output gate and the forget gate.



By Guillaume Chevalier - LARNN: Linear Attention Recurrent Neural Network, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=99599411

Mathematical notations of the LSTM architecture:

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$$
$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_c[h_{t-1}, x_t] + b_c)$$

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$$
$$h_t = o_t * tanh(C_t)$$

where, C refers to the cell state vectors, and f, i and o are the activation vectors of the forget gate, input gate and ouput gate respectively.

Three different variants of LSTMs have been used in this project: Vanilla LSTMs (as described above), Stacked LSTMs, which comprises of multiple LSTM layers, and Bidirectional LSTMs, which runs the input vectors in two directions. Hyperparameter tuning for each of the architectures have been done using the Optuna framework.

In [357…

```
class _Dataset:

    """
    Prepare Dataset for LSTM models
    """

    def __init__(self,df,timestep=10,forward=10):
        self.timestep=timestep
        self.forward=forward
        self.df=df.copy()

    def prepare(self,convert=True,prob='classification'):

        df_=self.df.copy()
        #df_=df_.dropna()

        Xtrain,Xtest=train_test_split(df_,test_size=0.3,shuffle=False)

        imputer=impute.SimpleImputer().fit(Xtrain)
        Xtrain=pd.DataFrame(imputer.transform(Xtrain),columns=df_.columns,index=Xtrain.index)
        Xtest=pd.DataFrame(imputer.transform(Xtest),columns=df_.columns,index=Xtest.index)

        scaler=preprocessing.StandardScaler().fit(Xtrain)
        Xtrain_scaled=pd.DataFrame(scaler.transform(Xtrain),columns=df_.columns,index=Xtrain.index)
        Xtest_scaled=pd.DataFrame(scaler.transform(Xtest),columns=df_.columns,index=Xtest.index)

        temp=Xtrain_scaled.append(Xtest_scaled).copy()
        final=pd.DataFrame(columns=temp.columns,index=temp.index)

        print('Preparing dataframes...')

        if prob=='classification':
            final['temp1']=''
            final['temp2']=pd.DataFrame(scaler.inverse_transform(temp),columns=temp.columns,index=temp.index).dret.values

            for col in (temp.columns):
                    for i in range(len(temp)):
                        final.loc[:,col][i]=(temp[col].values[i-self.timestep:i])
                        #final.loc[:,'temp1'][i]=(final['temp2'].values[i-self.forward:i])

            for i in range(len(temp)):
                final.loc[:,'temp1'][i]=(final['temp2'].values[i-self.forward:i])

            _max=max(self.forward,self.timestep)
            final=final.iloc[_max:]
```

```
            #Defining the target
            final['target']=final.temp1.shift(-self.forward)
            final=final.dropna()
            final['target']=final.target.apply(lambda x:1 if np.prod([i for i in map(lambda y:(1+y),x)])-1>0.025 else 0).values
            final=final.drop(['temp1','temp2'],axis=1)

        else:#Regression

            for col in (temp.columns):
                    for i in range(len(temp)):
                        final.loc[:,col][i]=(temp[col].values[i-self.timestep:i])

            _max=max(self.forward,self.timestep)
            final=final.iloc[_max:]

            #Defining the target
            final['target']=final.Close.shift(-self.forward)
            final=final.dropna()

        Xtrain,Xtest,ytrain,ytest=train_test_split(final.drop('target',axis=1),final.target,test_size=0.3,shuffle=False)

        if convert==False:
            return Xtrain,Xtest,ytrain,ytest

        else:
            print('\n')
            print('Converting dataframes to arrays...')

            if prob=='classification':
                X1=np.ndarray(shape=(Xtrain.shape[0],Xtrain.shape[1],self.timestep))
                X2=np.ndarray(shape=(Xtest.shape[0],Xtest.shape[1],self.timestep))

                y1=np.array(ytrain).reshape(-1,1)
                y2=np.array(ytest).reshape(-1,1)

                for i in (range(X1.shape[0])):
                    for j in range(X1.shape[1]):
                        for k in range(self.timestep):
                            X1[i,j,k]=Xtrain.iloc[i,j][k]

                for i in (range(X2.shape[0])):
                    for j in range(X2.shape[1]):
                        for k in range(self.timestep):
                            X2[i,j,k]=Xtest.iloc[i,j][k]

            else:#Regression

                X1=np.ndarray(shape=(Xtrain.shape[0],Xtrain.shape[1],self.timestep))
                X2=np.ndarray(shape=(Xtest.shape[0],Xtest.shape[1],self.timestep))

                y1=np.ndarray(shape=(ytrain.shape[0],self.forward))
                y2=np.ndarray(shape=(ytest.shape[0],self.forward))

                for i in (range(X1.shape[0])):
                    for j in range(X1.shape[1]):
                        for k in range(self.timestep):
                            X1[i,j,k]=Xtrain.iloc[i,j][k]

                    for k in range(self.forward):
                        y1[i,k]=ytrain.iloc[i][k]

                for i in (range(X2.shape[0])):
                    for j in range(X2.shape[1]):
                        for k in range(self.timestep):
                            X2[i,j,k]=Xtest.iloc[i,j][k]

                for k in range(self.forward):
                    y2[i,k]=ytrain.iloc[i][k]

            return X1,X2,y1,y2
```

In [358…

```
#LSTM class

class _LSTM:

    """
    A custom class for different LSTM model variants

    """

    def __init__(self,data,timestep=10,type=None):
        self.timestep=timestep
        self.type=type
        self.X1,self.X2,self.y1,self.y2=data


    def _fit(self,verbose=False,summary=False,epochs=100):

        x=layers.Input(shape=(self.X1.shape[1],self.timestep))

        if self.type==None or self.type=='vanilla lstm':
            y=layers.LSTM(50,activation='tanh',return_sequences=False,input_shape=(self.timestep,self.X1.shape[1]),name='VaLSTM')(
        elif self.type=='stacked lstm':
```

```
            y=layers.LSTM(50,activation='tanh',return_sequences=True,input_shape=(self.timestep,self.X1.shape[1]),name='LSTM1')(x)
            y=layers.LSTM(50,activation='tanh',name='LSTM2')(y)
        elif self.type=='bidirectional lstm':
            y=layers.Bidirectional(layers.LSTM(50,activation='tanh',input_shape=(self.timestep,self.X1.shape[1]),name='BiLSTM1'))(

        y=layers.Dense(200,activation='relu',name='Dense1')(y)
        y=layers.Dropout(0.2)(y)
        y=layers.BatchNormalization()(y)

        y=layers.Dense(100,activation='relu',name='Dense2')(y)
        y=layers.Dropout(0.2)(y)
        y=layers.BatchNormalization()(y)

        y=layers.Dense(1,activation='sigmoid',name='Output')(y)

        self.model=Model(inputs=x,outputs=y)

        self.model.compile(loss='binary_crossentropy',optimizer=optimizers.Adam(learning_rate=0.001),metrics=['AUC','accuracy'])

        if summary==True:self.model.summary()

        print('\n')
        print('Fitting model...')
        self.model.fit(self.X1,self.y1,epochs=epochs,verbose=0,shuffle=False,validation_split=0.25)

        K.clear_session()

    def _fit_regression(self,verbose=False,summary=False,forward=5,epochs=100):

        x=layers.Input(shape=(self.X1.shape[1],self.timestep))
        if self.type==None or self.type=='vanilla lstm':
            y=layers.LSTM(50,activation='tanh',return_sequences=False,input_shape=(self.timestep,self.X1.shape[1]),name='VaLSTM')(
        elif self.type=='stacked lstm':
            y=layers.LSTM(50,activation='tanh',return_sequences=True,input_shape=(self.timestep,self.X1.shape[1]),name='LSTM1')(x)
            y=layers.LSTM(50,activation='tanh',name='LSTM2')(y)
        elif self.type=='bidirectional lstm':
            y=layers.Bidirectional(layers.LSTM(50,activation='tanh',input_shape=(self.timestep,self.X1.shape[1]),name='BiLSTM1'))(

        y=layers.Dense(200,activation='relu',name='Dense1')(y)
        y=layers.Dropout(0.2)(y)
        y=layers.BatchNormalization()(y)

        y=layers.Dense(100,activation='relu',name='Dense2')(y)
        y=layers.Dropout(0.2)(y)
        y=layers.BatchNormalization()(y)

        y=layers.Dense(forward,name='Output')(y)

        self.model=Model(inputs=x,outputs=y)

        self.model.compile(loss='mse',optimizer=optimizers.Adam(learning_rate=0.001),metrics=['mean_squared_error'])

        if summary==True:self.model.summary()

        print('\n')
        print('Fitting model...')
        self.model.fit(self.X1,self.y1,epochs=epochs,verbose=0,shuffle=False,validation_split=0.25)

        K.clear_session()

    def _predict(self,X):
        preds=self.model.predict(X)
        return preds

    def _metrics(self,act,pred):

        self.fpr,self.tpr,self.th=metrics.roc_curve(act,pred)
        self.auc=metrics.auc(self.fpr,self.tpr)
        self.accuracy=metrics.accuracy_score(act,np.round(pred))

        return self.auc,self.accuracy

    def _plot_model(self):
        plot_model(self.model,show_shapes=True,show_layer_names=True)

    def _plots(self,ticker,act,pred,figsize=(20,5)):
        fig,ax=plt.subplots(1,2,figsize=figsize)
        ax[0].plot(self.fpr,self.tpr)
        ax[0].set_xlabel('FPR')
        ax[0].set_ylabel('TPR')
        ax[0].set_title(ticker+': ROC Curve')
        auc,acc=self._metrics(act,pred)
        ax[0].annotate('AUC: '+ str(np.round(auc,4)),xy=(0.85,0.1))

        ax[0].plot(np.linspace(0,1),np.linspace(0,1))

        cm=metrics.confusion_matrix(act,np.round(pred))
        #print(metrics.confusion_matrix(act,np.round(pred)))
        im=ax[1].imshow(cm,cmap='viridis')
        ax[1].set_xlabel("Predicted labels")
        ax[1].set_ylabel("True labels")
        ax[1].set_xticks([],[])
        ax[1].set_yticks([],[])
```

```
        ax[1].set_title(ticker+': Confusion matrix')

        fig.colorbar(im,ax=ax[1],orientation='vertical')

        plt.show()

    def _reg_preds(self):

        #Train set
        t1=self.y1
        t2=self.model.predict(self.X1)

        #Vaid set
        t3=self.y2
        t4=self.model.predict(self.X2)

        act=list((t1)[0])
        pred=list(t2[0])

        act_=list((t3)[0])
        pred_=list(t4[0])

        for i in range(1,len(self.y1)):
            act.append(t1[i][-1])
            pred.append(t2[i][-1])

        for i in range(1,len(self.y2)):
            act_.append(t3[i][-1])
            pred_.append(t4[i][-1])

        fig,ax=plt.subplots(1,2,figsize=(25,6))

        ax[0].plot(act,label='Actual train set')
        ax[0].plot(pred,label='Predictions',color='darkgreen')
        ax[0].set_title('Train set (Normalized prices), '+'RMSE: '+str(np.round(np.sqrt(metrics.mean_squared_error(act,pred)),4)))
        ax[0].legend()

        ax[1].plot(act_,label='Actual valid set',color='orange')
        ax[1].plot(pred_,label='Predictions',color='darkgreen')
        ax[1].set_title('Valid set (Normalized prices), '+'RMSE: '+str(np.round(np.sqrt(metrics.mean_squared_error(act_,pred_)),4)))
        ax[1].legend()


#Function to calculate Metrics for classification
def check_metrics(ticker,data,lstm_model):
    #print('\nStock: ',ticker)
    act=data[3]
    pred=lstm_model._predict(X=data[1])
    auc,acc=lstm_model._metrics(act,pred)
    lstm_model._plots(ticker,act,pred)
```

Dataset prep with SOM features

The datasets for each stock are prepared for inputs into the LSTM models as shown below. A timestep of 10 is used initially for predicting the forward 5 day componded return.

In [359…

```
data={}

for i,ticker in enumerate(ytickers):

    print('\nPreparing dataset for: ', ticker)
    data[ticker]=_Dataset(stocks[ticker][features_som[ticker]],timestep=10,forward=5).prepare(convert=True)

"""
Alternative: Load already pre-prepared datasets (to save time!)

for i,ticker in enumerate(ytickers):

    with open(f'../inputs/preprepared/SOM/{ticker}.pkl','rb') as f:
        data[ticker]=pickle.load(f)

"""
```

```
Preparing dataset for:  BATS.L
Preparing dataframes...


Converting dataframes to arrays...

Preparing dataset for:  DTE.DE
Preparing dataframes...


Converting dataframes to arrays...

Preparing dataset for:  RNO.PA
Preparing dataframes...


Converting dataframes to arrays...

Preparing dataset for:  SIE.DE
Preparing dataframes...
```

```
Converting dataframes to arrays...

Preparing dataset for:  TTE.PA
Preparing dataframes...


Converting dataframes to arrays...
```

Out[359… "\nAlternative: Load already pre-prepared datasets (to save time!)\n\nfor i,ticker in enumerate(ytickers):\n   \n   with open (f'../inputs/preprepared/SOM/{ticker}.pkl','rb') as f:\n   data[ticker]=pickle.load(f)\n\n"

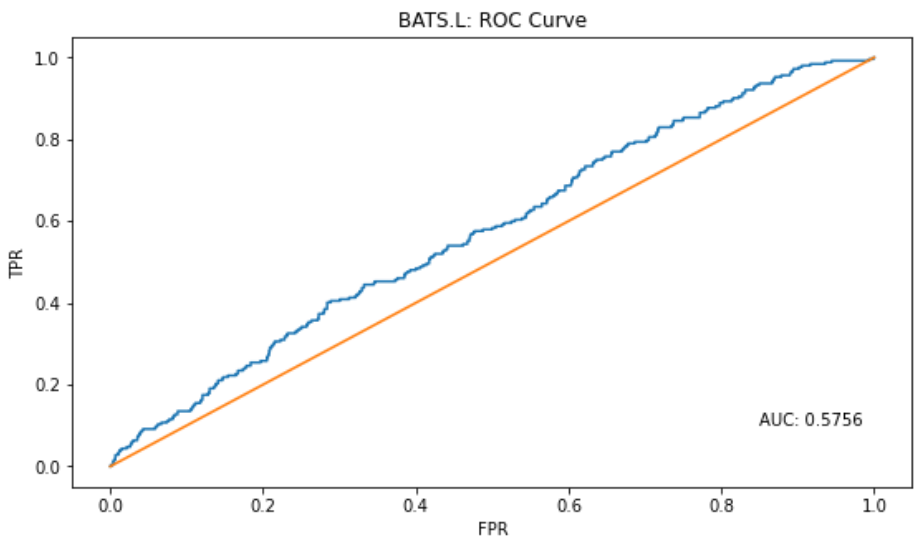
In [360…
```
#Train and valid datasets

data['TTE.PA'][0].shape,data['TTE.PA'][1].shape,data['TTE.PA'][2].shape,data['TTE.PA'][3].shape
```

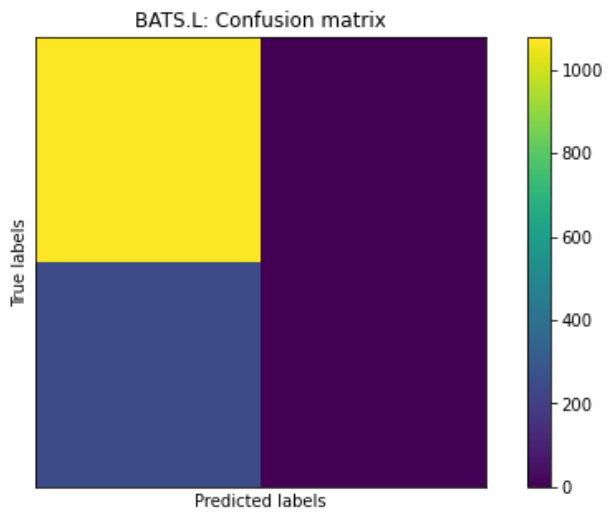
Out[360… ((3144, 42, 10), (1348, 42, 10), (3144, 1), (1348, 1))

LSTM with SOM features

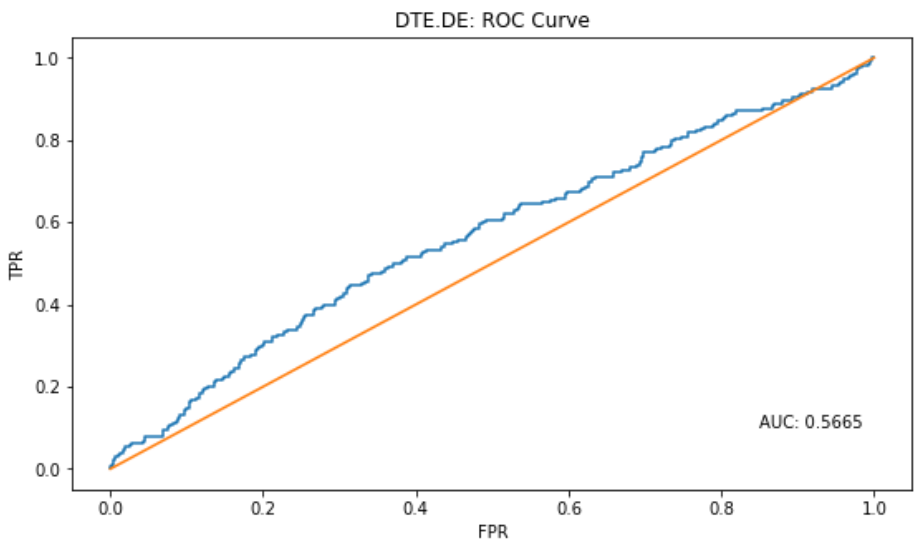
In [361…
```
lstm_models={}

for i,ticker in enumerate(ytickers):

    print('\nFitting model for: ', ticker)

    lstm_models[ticker]=_LSTM(data[ticker],timestep=10,type='vanilla lstm')
    lstm_models[ticker]._fit(epochs=5)
    check_metrics(ticker,data[ticker],lstm_models[ticker])
```

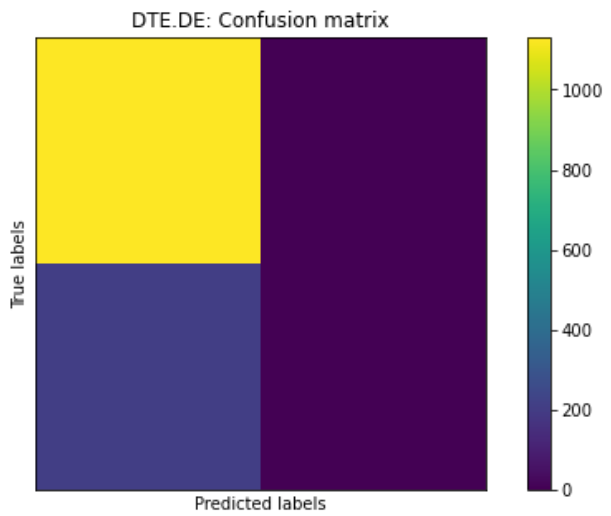
```
Fitting model for:  BATS.L

Fitting model...
```



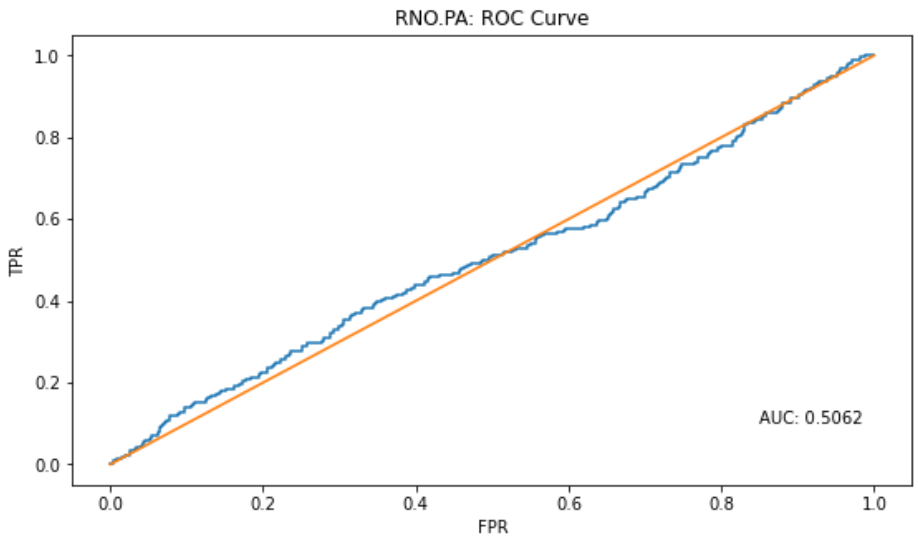


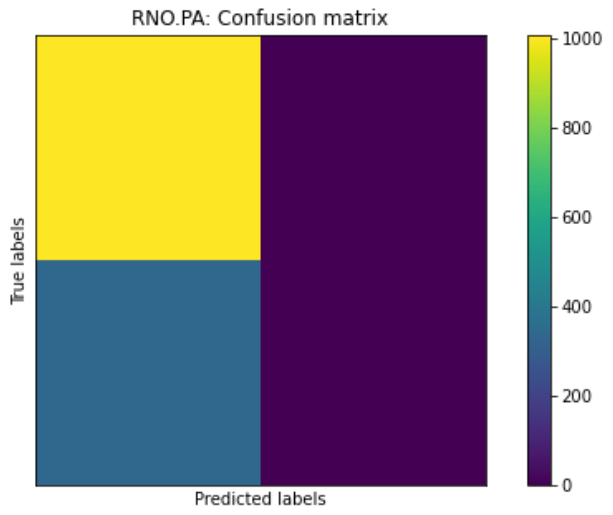

```
Fitting model for:  DTE.DE

Fitting model...
```





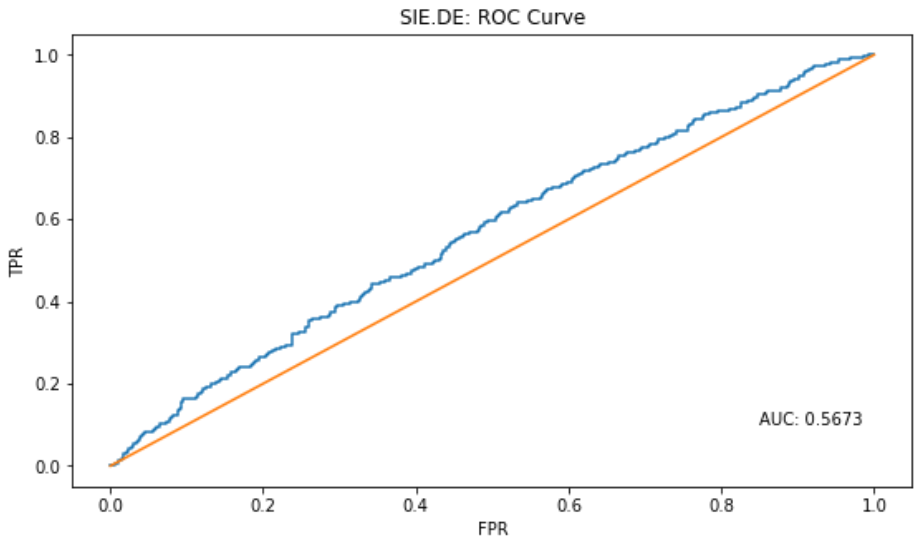

```
Fitting model for:  RNO.PA

Fitting model...
```

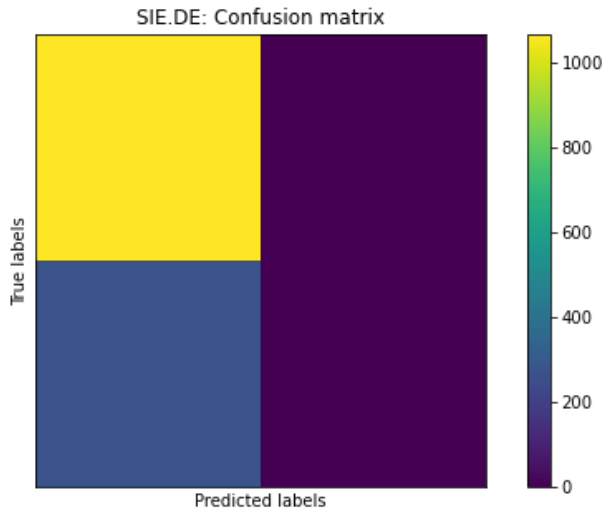





```
Fitting model for:  SIE.DE
```

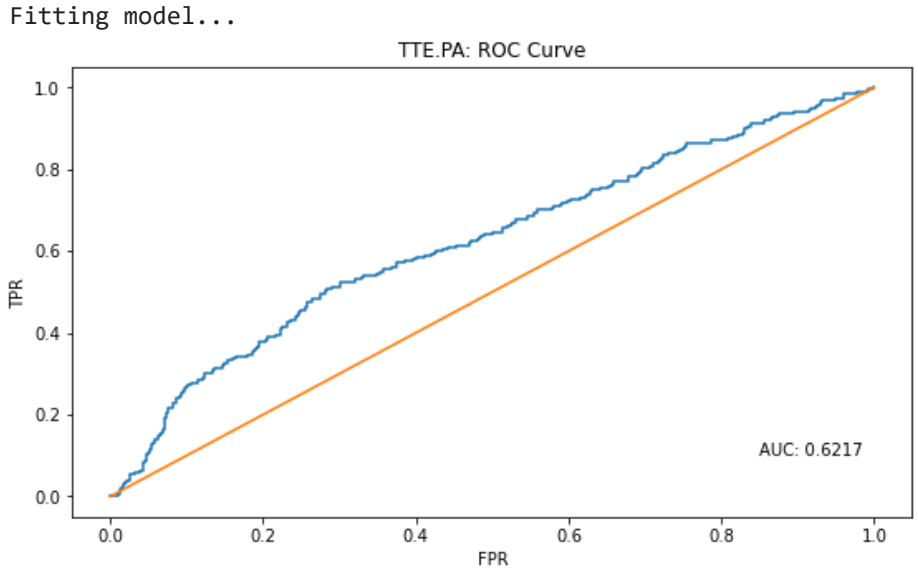
```
Fitting model...
```

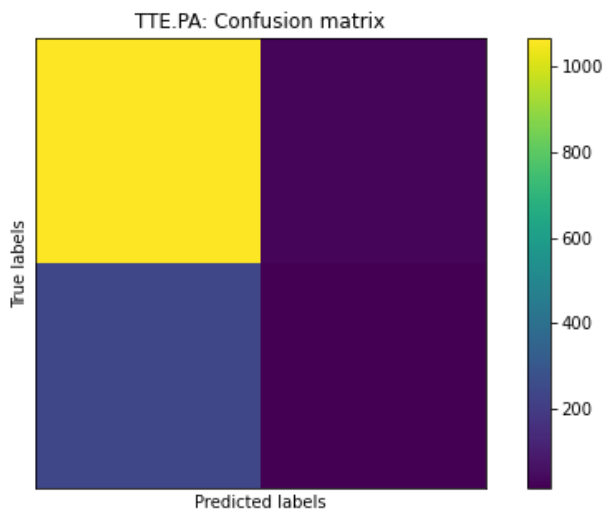





```
Fitting model for:  TTE.PA
```

```
Fitting model...
```





Varying parameters (with SOM features)

In [362…

```
summary=pd.DataFrame(columns=['Ticker','LSTM architecture','Feature Selection','Epochs','AUC','Accuracy'])

lstms_=['stacked lstm','bidirectional lstm']
epochs_=[10,20]

for ticker in ytickers:

    print('\nCalculating for:',ticker)

    for lstm in lstms_:
        for epoch in epochs_:

            #print('Parameters: epochs=',epoch,' lstm=',lstm)

            model=_LSTM(data[ticker],timestep=10,type=lstm)
            model._fit(epochs=epoch)

            act=data[ticker][3]
            pred=model._predict(X=data[ticker][1])
            auc,acc=model._metrics(act,pred)

            summary=summary.append({'Ticker':ticker,'LSTM architecture':lstm,'Feature Selection':'SOM','Epochs':epoch,'AUC':auc,'A
            #print('Fitted!')

summary=summary.reset_index(drop=True)
```

```
Calculating for: BATS.L


Fitting model...
```

```
Fitting model...

Fitting model...

Fitting model...
Calculating for: DTE.DE

Fitting model...

Fitting model...

Fitting model...

Fitting model...

Fitting model...
Calculating for: RNO.PA

Fitting model...

Fitting model...

Fitting model...

Fitting model...

Fitting model...
Calculating for: SIE.DE

Fitting model...

Fitting model...

Fitting model...

Fitting model...

Fitting model...
Calculating for: TTE.PA

Fitting model...

Fitting model...

Fitting model...

Fitting model...
```

In [363…

```
summary.drop('Accuracy',axis=1).style.background_gradient(cmap=sns.color_palette('RdYlGn',as_cmap=True),axis=None)
```

Out[363…

	Ticker	LSTM architecture	Feature Selection	Epochs	AUC
0	BATS.L	stacked lstm	SOM	10	0.559075
1	BATS.L	stacked lstm	SOM	20	0.572261
2	BATS.L	bidirectional lstm	SOM	10	0.535692
3	BATS.L	bidirectional lstm	SOM	20	0.573722
4	DTE.DE	stacked lstm	SOM	10	0.594260
5	DTE.DE	stacked lstm	SOM	20	0.594991
6	DTE.DE	bidirectional lstm	SOM	10	0.612330
7	DTE.DE	bidirectional lstm	SOM	20	0.582220
8	RNO.PA	stacked lstm	SOM	10	0.571987
9	RNO.PA	stacked lstm	SOM	20	0.565411
10	RNO.PA	bidirectional lstm	SOM	10	0.573343
11	RNO.PA	bidirectional lstm	SOM	20	0.599236
12	SIE.DE	stacked lstm	SOM	10	0.600746
13	SIE.DE	stacked lstm	SOM	20	0.609047
14	SIE.DE	bidirectional lstm	SOM	10	0.674913
15	SIE.DE	bidirectional lstm	SOM	20	0.616579

	Ticker	LSTM architecture	Feature Selection	Epochs	AUC
16	TTE.PA	stacked lstm	SOM	10	0.625222
17	TTE.PA	stacked lstm	SOM	20	0.625102
18	TTE.PA	bidirectional lstm	SOM	10	0.621532
19	TTE.PA	bidirectional lstm	SOM	20	0.619556

LSTM with KMC features

In [364…

```
data_={}

for i,ticker in enumerate(ytickers):

    print('\nPreparing dataset for: ', ticker)
    data_[ticker]=_Dataset(stocks[ticker][features_km[ticker]],timestep=10,forward=5).prepare(convert=True)


"""
Alternative: Load already pre-prepared datasets (to save time!)

for i,ticker in enumerate(ytickers):

    with open(f'../inputs/preprepared/KMC/{ticker}.pkl','rb') as f:
        data_[ticker]=pickle.load(f)

"""
```

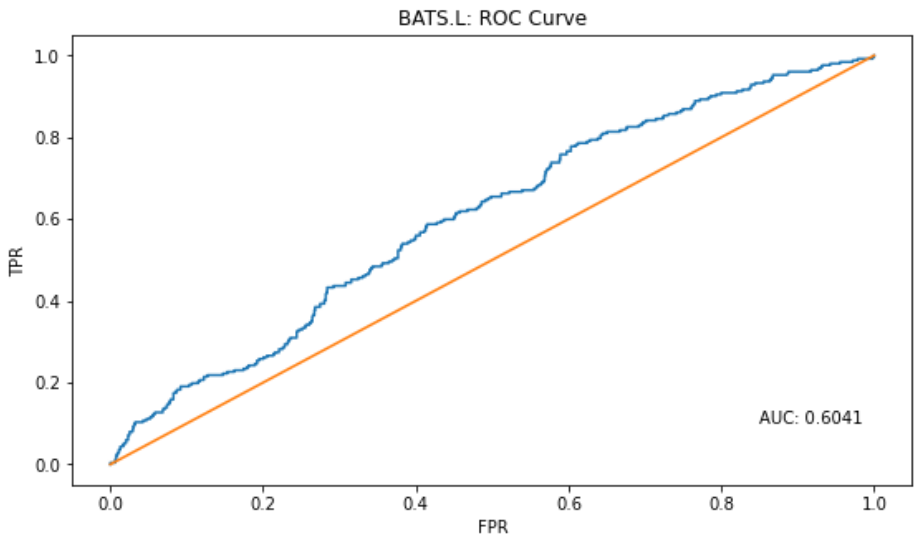
```
Preparing dataset for:  BATS.L
Preparing dataframes...


Converting dataframes to arrays...

Preparing dataset for:  DTE.DE
Preparing dataframes...


Converting dataframes to arrays...

Preparing dataset for:  RNO.PA
Preparing dataframes...


Converting dataframes to arrays...

Preparing dataset for:  SIE.DE
Preparing dataframes...


Converting dataframes to arrays...

Preparing dataset for:  TTE.PA
Preparing dataframes...


Converting dataframes to arrays...
```

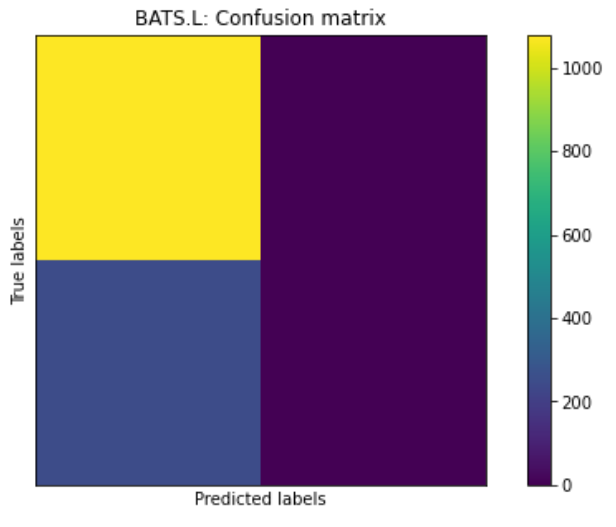
Out[364… `"\nAlternative: Load already pre-prepared datasets (to save time!)\n\nfor i,ticker in enumerate(ytickers):\n    \n    with open (f'../inputs/preprepared/KMC/{ticker}.pkl','rb') as f:\n        data_[ticker]=pickle.load(f)\n\n"`

In [365…

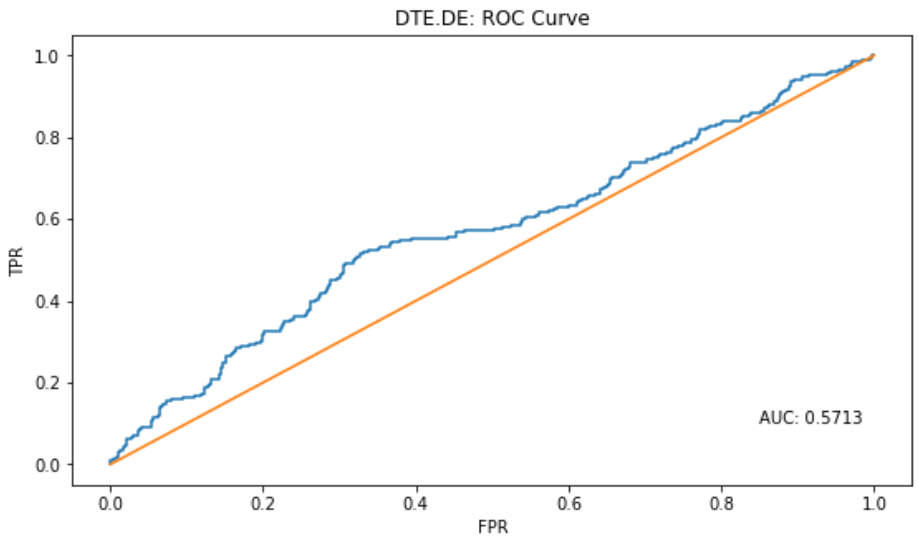
```
lstm_models_={}

for i,ticker in enumerate(ytickers):

    print('\nFitting model for: ', ticker)

    lstm_models_[ticker]=_LSTM(data_[ticker],timestep=10,type='vanilla lstm')
    lstm_models_[ticker]._fit(epochs=5)
    check_metrics(ticker,data_[ticker],lstm_models_[ticker])
```

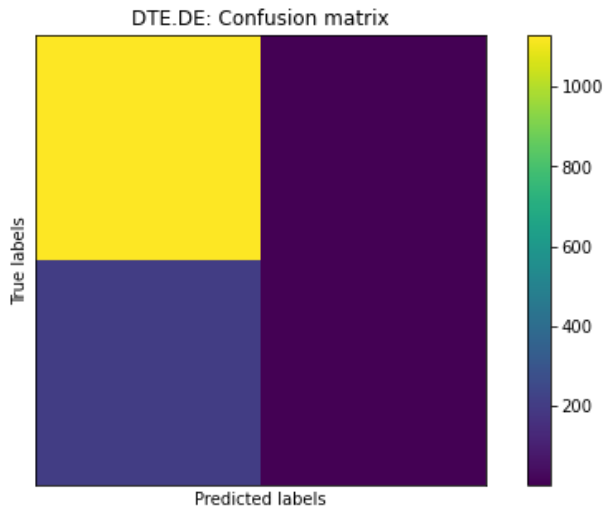
```
Fitting model for:  BATS.L

Fitting model...
```



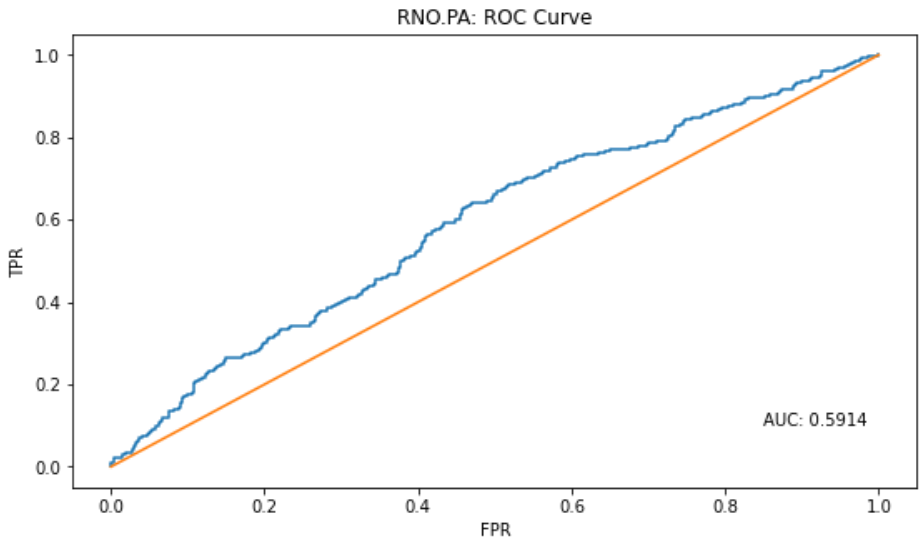


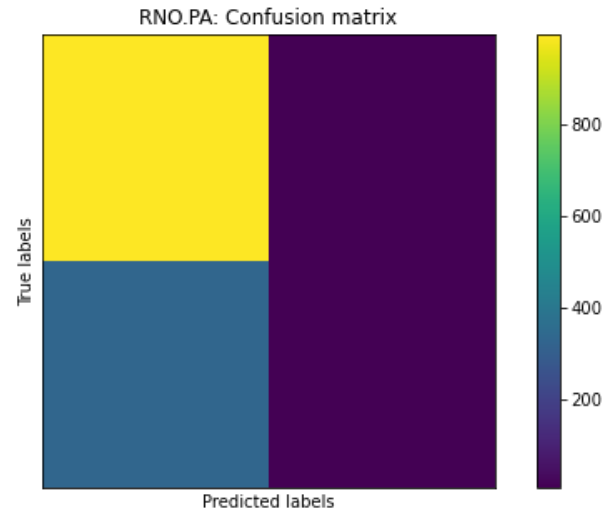
```
Fitting model for:  DTE.DE

Fitting model...
```



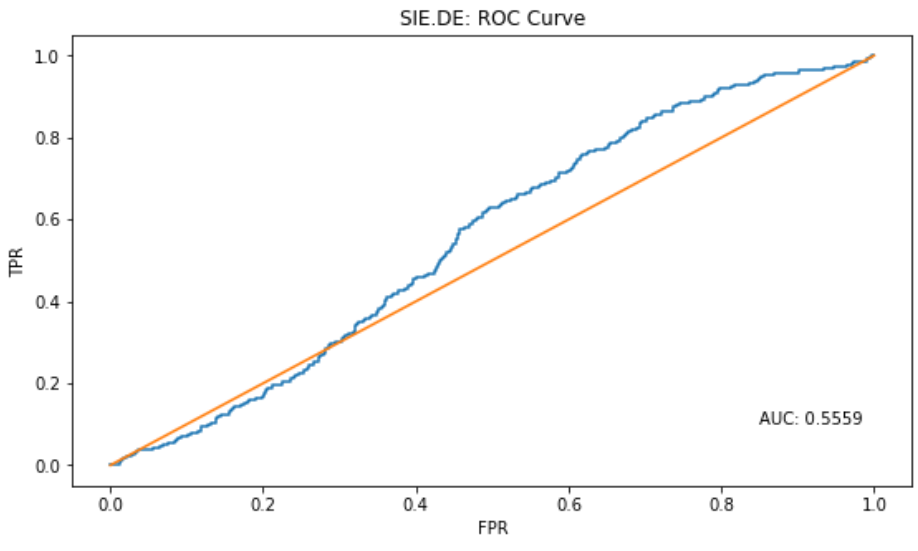


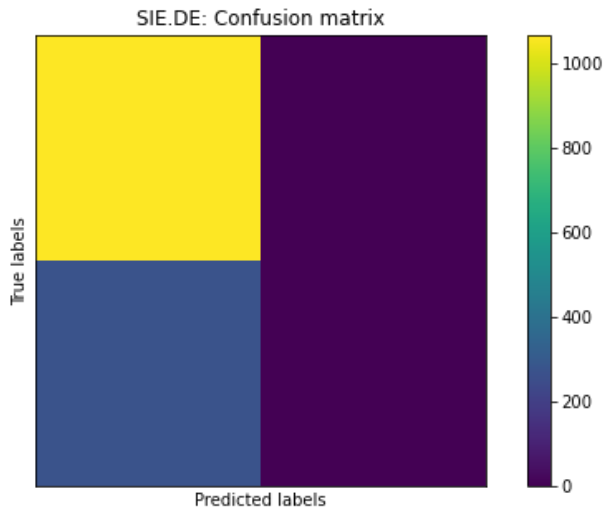

```
Fitting model for:  RNO.PA

Fitting model...
```



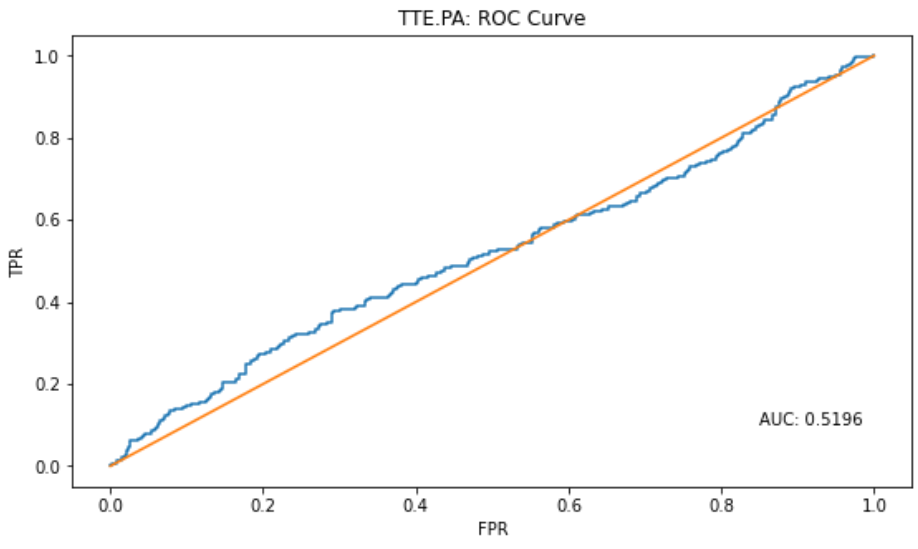


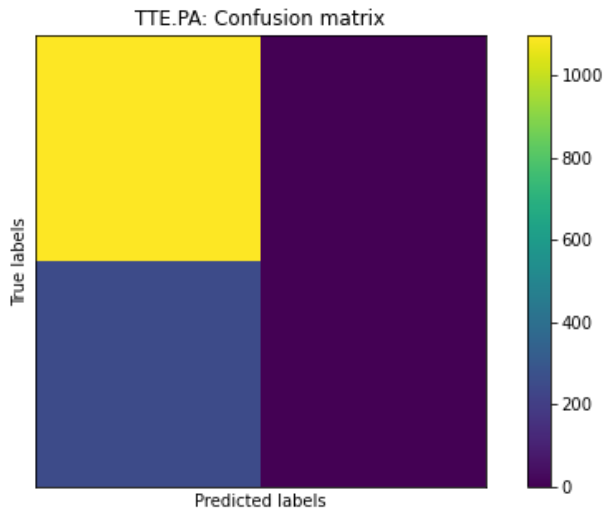

```
Fitting model for:  SIE.DE

Fitting model...
```






```
Fitting model for:  TTE.PA

Fitting model...
```





Varying parameters (with KMC features)

In [366…

```
summary_=pd.DataFrame(columns=['Ticker','LSTM architecture','Feature Selection','Epochs','AUC','Accuracy'])

lstms_=['stacked lstm','bidirectional lstm']
epochs_=[10,20]
```

```
for ticker in ytickers:

    print('\nCalculating for:',ticker)

    for lstm in lstms_:
        for epoch in epochs_:

            #print('Parameters: epochs=',epoch,' lstm=',lstm)

            model=_LSTM(data_[ticker],timestep=10,type=lstm)
            model._fit(epochs=epoch)

            act=data_[ticker][3]
            pred=model._predict(X=data_[ticker][1])
            auc,acc=model._metrics(act,pred)

            summary_=summary_.append({'Ticker':ticker,'LSTM architecture':lstm,'Feature Selection':'KMC','Epochs':epoch,'AUC':auc,
            #print('Fitted!')

summary_=summary_.reset_index(drop=True)
```

```
Calculating for: BATS.L


Fitting model...


Fitting model...


Fitting model...


Fitting model...

Calculating for: DTE.DE


Fitting model...


Fitting model...


Fitting model...


Fitting model...

Calculating for: RNO.PA


Fitting model...


Fitting model...


Fitting model...


Fitting model...

Calculating for: SIE.DE


Fitting model...


Fitting model...


Fitting model...


Fitting model...

Calculating for: TTE.PA


Fitting model...


Fitting model...


Fitting model...


Fitting model...
```

In [367…

```
summary_.drop('Accuracy',axis=1).style.background_gradient(cmap=sns.color_palette('RdYlGn',as_cmap=True),axis=None)
```

Out[367…

	Ticker	LSTM architecture	Feature Selection	Epochs	AUC
0	BATS.L	stacked lstm	KMC	10	0.616305

	Ticker	LSTM architecture	Feature Selection	Epochs	AUC
1	BATS.L	stacked lstm	KMC	20	0.601417
2	BATS.L	bidirectional lstm	KMC	10	0.608023
3	BATS.L	bidirectional lstm	KMC	20	0.613982
4	DTE.DE	stacked lstm	KMC	10	0.549531
5	DTE.DE	stacked lstm	KMC	20	0.535282
6	DTE.DE	bidirectional lstm	KMC	10	0.595822
7	DTE.DE	bidirectional lstm	KMC	20	0.604256
8	RNO.PA	stacked lstm	KMC	10	0.590256
9	RNO.PA	stacked lstm	KMC	20	0.594065
10	RNO.PA	bidirectional lstm	KMC	10	0.602357
11	RNO.PA	bidirectional lstm	KMC	20	0.611371
12	SIE.DE	stacked lstm	KMC	10	0.513498
13	SIE.DE	stacked lstm	KMC	20	0.639437
14	SIE.DE	bidirectional lstm	KMC	10	0.553481
15	SIE.DE	bidirectional lstm	KMC	20	0.617764
16	TTE.PA	stacked lstm	KMC	10	0.487094
17	TTE.PA	stacked lstm	KMC	20	0.516390
18	TTE.PA	bidirectional lstm	KMC	10	0.614559
19	TTE.PA	bidirectional lstm	KMC	20	0.609676

Comparison with tree-based methods

Tree-based ML models are give some of the best results for tabular data. If we view our time-series data as a tabular data, it's be worth comparing the LSTM models to tree based gradient boosting models like XGBoost.

In [368…

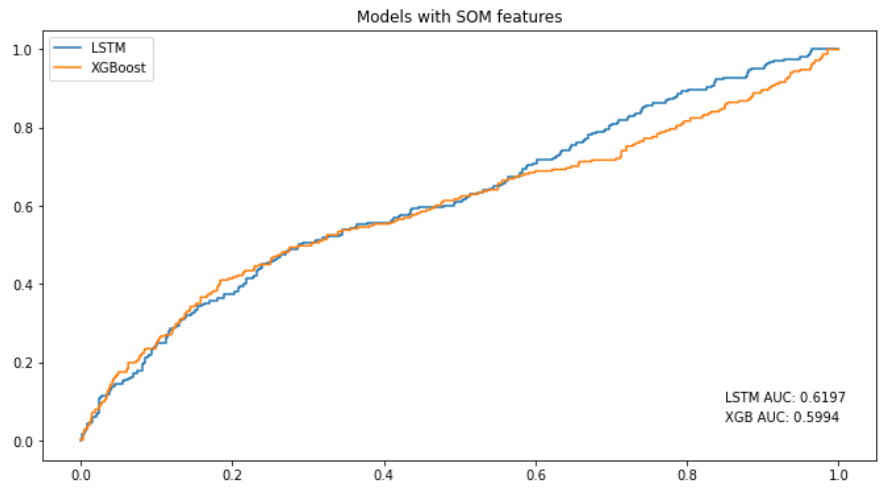
```
#!pip install xgboost
import xgboost as xgb

ticker=='RNO.PA'

#Preparing dataset for Timestep=1 and 5D forward return predictions:
df=_Dataset(stocks[ticker][features_som[ticker]],timestep=1,forward=5).prepare(convert=True)
df_=_Dataset(stocks[ticker][features_km[ticker]],timestep=1,forward=5).prepare(convert=True)

model_lstm={}
model_xgb={}
fpr_lstm={};tpr_lstm={};auc_lstm={}
fpr_xgb={};tpr_xgb={};auc_xgb={}

for i,d in enumerate([df,df_]):

    Xtrain=d[0].reshape(-1,d[0].shape[1])
    Xtest=d[1].reshape(-1,d[1].shape[1])
    ytrain=d[2].reshape(-1,d[2].shape[1])
    ytest=d[3].reshape(-1,d[3].shape[1])

    model_lstm[i]=_LSTM((Xtrain,Xtest,ytrain,ytest),timestep=1,type='vanilla lstm')
    model_lstm[i]._fit(epochs=5)
    auc_lstm[i],_=model_lstm[i]._metrics(ytest,model_lstm[i]._predict(Xtest))
    fpr_lstm[i],tpr_lstm[i]=model_lstm[i].fpr,model_lstm[i].tpr

    model_xgb[i]=xgb.XGBClassifier(n_estimators=1000,max_depth=7,eval_metric='auc',random_state=21).fit(Xtrain,ytrain,verbose=Fals
    fpr_xgb[i],tpr_xgb[i],_=metrics.roc_curve(ytest,model_xgb[i].predict_proba(Xtest)[:,1])
    auc_xgb[i]=metrics.auc(fpr_xgb[i],tpr_xgb[i])
```

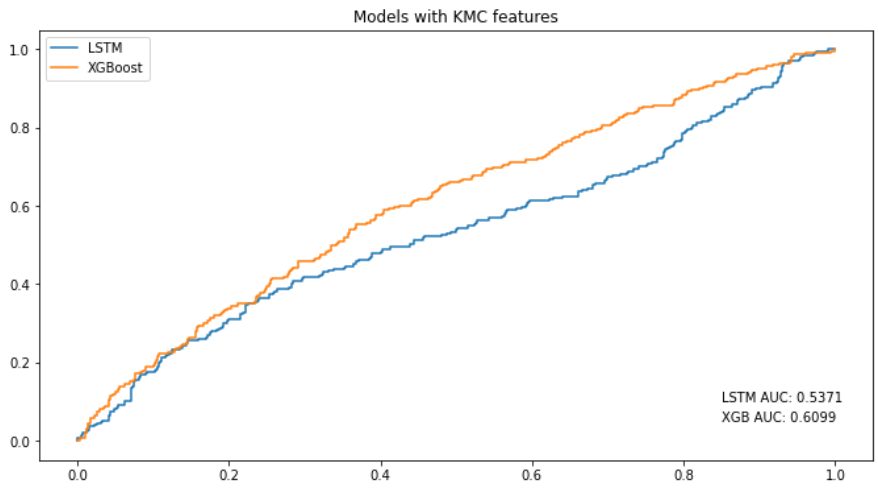
```
Preparing dataframes...


Converting dataframes to arrays...
Preparing dataframes...


Converting dataframes to arrays...


Fitting model...


Fitting model...
```

In [369…

```
fig,ax=plt.subplots(1,2,figsize=(25,6))

ax[0].set_title('Models with SOM features')
ax[0].plot(fpr_lstm[0],tpr_lstm[0],label='LSTM')
ax[0].plot(fpr_xgb[0],tpr_xgb[0],label='XGBoost')
ax[0].annotate('LSTM AUC: '+ str(np.round(auc_lstm[0],4)),xy=(0.85,0.1))
```

```
ax[0].annotate('XGB AUC: '+ str(np.round(auc_xgb[0],4)),xy=(0.85,0.05))
ax[0].legend()

ax[1].set_title('Models with KMC features')
ax[1].plot(fpr_lstm[1],tpr_lstm[1],label='LSTM')
ax[1].plot(fpr_xgb[1],tpr_xgb[1],label='XGBoost')
ax[1].annotate('LSTM AUC: '+ str(np.round(auc_lstm[1],4)),xy=(0.85,0.1))
ax[1].annotate('XGB AUC: '+ str(np.round(auc_xgb[1],4)),xy=(0.85,0.05))
ax[1].legend()

plt.show()
```





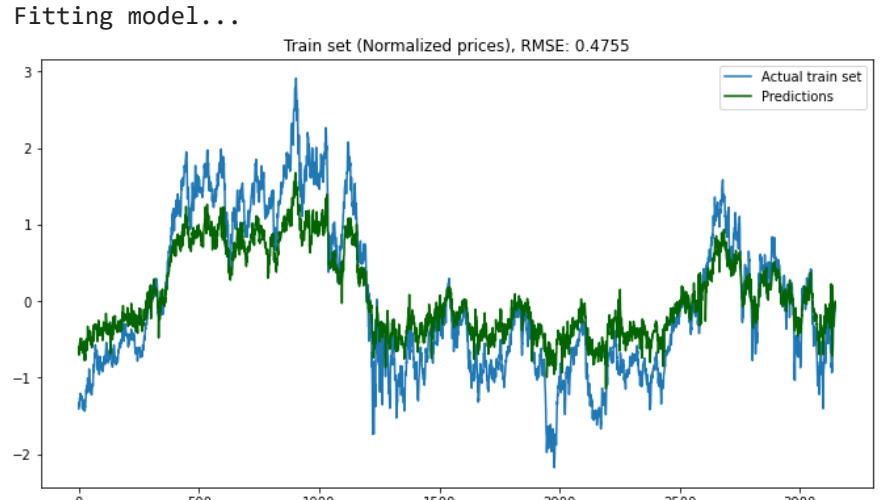
Regression problem?

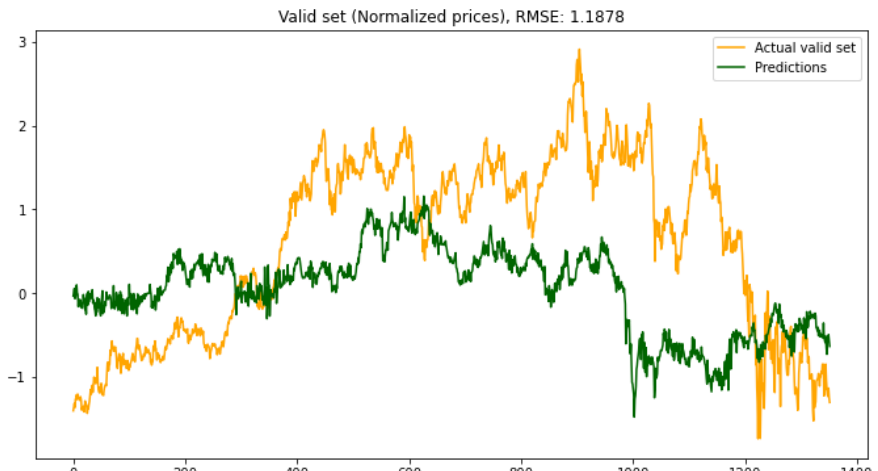
The above problem could be viewed as a regression problem too and the n-day forward Closing prices, for instance, could be predicted as shown below.

In [370…

```
ticker='TTE.PA'

reg_data=_Dataset(stocks[ticker][features_som[ticker] + ['Close'] if 'Close' not in features_som[ticker] else features_som[ticker]
reg=_LSTM(reg_data,timestep=10,type='vanilla lstm')
reg._fit_regression(epochs=10)
reg._reg_preds()
```

```
Preparing dataframes...


Converting dataframes to arrays...


Fitting model...
```





Of course, a different LSTM architecture as compared to the one used for classification and different hyperparameter optimzations would lead to lower MSEs than the ones shown above.

Conclusion

5 day forward returns for five different stocks were predicted using the LSTM deep learning framework. It's quite clear from the experimentations above that due to the stochastic nature of the stocks and due to the fact that different stocks (especially those from different industries and sectors) behave differently and might have different idiosyncratic properties, an LSTM architecture that might work well for a particular stock might not do that well on another stock, and worse still might give completely misleading results. For instance, the bidirectional LSTM architectures do work well for most of the stocks, but the performance of the stacked LSTM architectures are somewhat mixed (works well for stocks like British American Tobacco, not quite for Total Energies!). It's here that domain expertise when predictions are made for certain stocks is extremely vital and would lead to far better results when compared to blindly following the models. This also holds true when the feature engineering and feature selection steps are carried out before defining the model architectures.

A comparison with gradient boosting algorithms reveals that there's not much difference when the timestep equals one. However, LSTMs do give us the added flexibility of varying the timesteps and feeding those as a single input to the model, which might lead to better predictions for long time series data and which is not possible with xgboost and other boosting algorithms.

Further research work on LSTMs with added attention masks and which have worked wonders on NLP tasks might be worth a try on time series predictions.

References

1. https://en.wikipedia.org/wiki/Self-organizing_map
2. https://github.com/JustGlowing/minisom
3. https://stanford.edu/~cpiech/cs221/handouts/kmeans.html
4. https://en.wikipedia.org/wiki/Recurrent_neural_network
5. https://en.wikipedia.org/wiki/Long_short-term_memory

In [ ]: