NYSE Stockmarket Prediction Notebook

Extract Transform and Loading of data

The following notebook will be used to extract Transform and load the data, this notebook also includes the initial data exploration.

The following lines of code import all the necessary libraries that are required for the success of this Data Science Project

In [1]:

```
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import pandas as pd
import numpy as np
import math, time
from sklearn import preprocessing
import tensorflow as tf
from math import sqrt
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation
from keras.layers.recurrent import LSTM
from keras.models import load_model
import keras

import warnings
%matplotlib inline
warnings.filterwarnings('ignore')
```

```
Using TensorFlow backend.
```

In [2]:

```
# This block of code reads all the CSV information for the stock exchange data until 20
16 and shifts close to the end of the column
DATA_URL = 'prices-split-adjusted.csv'
df = pd.read_csv(DATA_URL)
df['Close'] = df['close']
df.drop(['close'], 1, inplace=True)
df.head()
```

Out[2]:

	date	symbol	open	low	high	volume	Close
0	2016-01-05	WLTW	123.430000	122.309998	126.250000	2163600.0	125.839996
1	2016-01-06	WLTW	125.239998	119.940002	125.540001	2386400.0	119.980003
2	2016-01-07	WLTW	116.379997	114.930000	119.739998	2489500.0	114.949997
3	2016-01-08	WLTW	115.480003	113.500000	117.440002	2006300.0	116.620003
4	2016-01-11	WLTW	117.010002	114.089996	117.330002	1408600.0	114.970001

In [3]:

```
#Checks to see if there are any empty values in the data frame which there are none
df.isnull().values.any()
```

Out[3]:

```
False
```

In [4]:

```
#Checks to see if all the stock prices are a float, which they are
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 851264 entries, 0 to 851263
Data columns (total 7 columns):
 #   Column  Non-Null Count   Dtype  
---  ------  --------------   -----  
 0   date    851264 non-null  object 
 1   symbol  851264 non-null  object 
 2   open    851264 non-null  float64
 3   low     851264 non-null  float64
 4   high    851264 non-null  float64
 5   volume  851264 non-null  float64
 6   Close   851264 non-null  float64
dtypes: float64(5), object(2)
memory usage: 45.5+ MB
```

In [5]:

```
#Just looking to better understand the data and noting how much data is available
df.describe()
```

Out[5]:

	open	low	high	volume	Close
count	851264.000000	851264.000000	851264.000000	8.512640e+05	851264.000000
mean	64.993618	64.336541	65.639748	5.415113e+06	65.011913
std	75.203893	74.459518	75.906861	1.249468e+07	75.201216
min	1.660000	1.500000	1.810000	0.000000e+00	1.590000
25%	31.270000	30.940001	31.620001	1.221500e+06	31.292776
50%	48.459999	47.970001	48.959999	2.476250e+06	48.480000
75%	75.120003	74.400002	75.849998	5.222500e+06	75.139999
max	1584.439941	1549.939941	1600.930054	8.596434e+08	1578.130005

In [6]:

```
# Checks the correlation of the data
df.corr()
```

Out[6]:

	open	low	high	volume	Close
open	1.000000	0.999914	0.999928	-0.132526	0.999845
low	0.999914	1.000000	0.999889	-0.133087	0.999926
high	0.999928	0.999889	1.000000	-0.132168	0.999925
volume	-0.132526	-0.133087	-0.132168	1.000000	-0.132675
Close	0.999845	0.999926	0.999925	-0.132675	1.000000

The above correlation shows strong linear relationships between open, low, high and close. Volume has very little correlation.

In [7]:

```
#Retrieves a list of all the companies on the stock market which there are 501 and list
ed below
symbols = list(set(df.symbol))
print(len(symbols))
for i in symbols:
    print(i, end= " : " )
```

```
501
FAST : HIG : ADS : ESS : AEE : INTC : STZ : HCP : LVLT : ALK : MTB : PFE :
ECL : VZ : WFC : BSX : CTSH : GGP : HSIC : EXPD : PBI : VNO : UAL : EBAY :
SBUX : PBCT : WLTW : EIX : DISCA : QRVO : RF : DPS : HAS : AIZ : FLIR : SL
B : LLY : AEP : WMT : KMX : XEL : HUM : MPC : CVX : D : SPG : TGT : PRU :
EFX : MNK : ADI : JNPR : DLPH : MA : NOV : WM : TGNA : WAT : BEN : FMC : P
SX : NLSN : AIV : TXT : ES : DIS : ESRX : SWN : URI : ITW : COTY : AZO : G
OOGL : V : TRV : EVHC : AON : XOM : APH : NWS : VFC : NBL : STX : LOW : IV
Z : ABBV : FDX : SE : PDCO : RAI : COH : TDG : OKE : PYPL : GM : AJG : ADB
E : SWKS : RSG : KLAC : GS : WU : MCK : CMCSA : PPG : URBN : PPL : CPB : C
TAS : PCLN : AMT : AVY : AMP : ENDP : BLL : AXP : TSCO : MSI : AAL : ATVI
: JWN : MCO : DHI : A : EMN : SRCL : IP : LKQ : HOLX : AVGO : FIS : TXN :
AMGN : NWL : MRK : RRC : TIF : FTR : SRE : PWR : BBT : DAL : SCHW : CLX :
CCI : NRG : PX : SYMC : YHOO : SJM : PNR : COL : FTV : MAS : COO : UNH : C
B : XLNX : PXD : MO : HPQ : IRM : SCG : KHC : ETFC : MKC : BLK : CAH : GPC
: ADP : ZION : LNT : ETN : TSS : PSA : LH : MET : CF : GT : ABC : WHR : AL
XN : RHT : GOOG : FISV : VRSN : JBHT : BBBY : KMI : CSRA : PNW : FOXA : O
: KSU : WY : C : MUR : SNA : CL : EXR : YUM : ADM : KSS : DISCK : AFL : TR
IP : MRO : CHK : NFX : XRAY : FLR : NSC : GIS : XL : PGR : CI : MMM : NEM
: ETR : PH : AET : HP : HST : JNJ : DD : AN : MYL : TSN : CHD : STI : BBY
: WBA : MJN : ORLY : ACN : PCAR : UNP : ORCL : CA : HRS : CTXS : WMB : FTI
: UPS : TAP : MNST : FE : DRI : GE : NTRS : ROK : KR : HCA : CBG : HCN : C
ME : HPE : GLW : KEY : BCR : NDAQ : WYNN : HSY : WEC : FOX : M : FITB : TM
O : SHW : JCI : COG : EXPE : ABT : SYK : CNP : LNC : FRT : MAT : IPG : HBI
: ANTM : ZTS : PAYX : QCOM : MMC : F : CRM : CBS : GPN : SIG : BMY : HON :
BK : ROP : PG : CINF : GPS : APA : AAP : NTAP : CTL : JPM : EOG : GILD : S
NI : CHRW : EL : LLL : EW : IDXX : FLS : CERN : MAA : DGX : CMI : TSO : FF
IV : KMB : PFG : FB : EQR : MOS : CMG : FCX : DVA : PLD : BIIB : CSCO : DE
: MU : VIAB : SPLS : PEP : CVS : LRCX : RIG : FBHS : DFS : IR : MDT : BWA
: UHS : SO : BHI : CAG : OXY : MHK : RCL : ZBH : PM : ISRG : KIM : AIG : R
L : FSLR : TEL : KO : TROW : ALL : CMS : DOV : CNC : EXC : PRGO : SLG : NK
E : L : AWK : INTU : BA : HES : FL : AMG : ALB : UTX : RHI : NOC : DLR : M
ON : DTE : CFG : PNC : COP : EA : PCG : WFM : SEE : PEG : DNB : XEC : SWK
: MAR : JEC : AES : COF : RTN : CSX : ARNC : BXP : AGN : LB : AMAT : APC :
CMA : HD : DVN : NVDA : LEN : ALLE : VRTX : SPGI : HBAN : LMT : ROST : ULT
A : EMR : MCD : UNM : HRB : XYL : LUV : EQIX : ED : ICE : R : UDR : CELG :
WRK : MDLZ : GWW : VLO : SYY : LUK : MTD : CHTR : NFLX : GRMN : WDC : IBM
: LLTC : AVB : NWSA : BAX : MLM : LYB : NAVI : T : UAA : USB : EQT : MSFT
: CCL : IFF : SYF : DUK : REGN : TMK : VAR : AKAM : ILMN : PKI : KORS : DL
TR : DG : HAR : NEE : HOG : PVH : K : MAC : XRX : CAT : OMC : VRSK : CXO :
STT : AAPL : MCHP : TDC : VMC : BAC : AME : DOW : TWX : LEG : APD : NUE :
AYI : TJX : GD : COST : HAL : ADSK : AMZN : NI : DHR : BDX : WYN : VTR : P
HM : HRL :
```

In [8]:

```
# Choosing apple as a company to look at as AAPL
company_code = 'VZ'
df = df[df.symbol == company_code]
df.drop(['symbol'], 1, inplace=True)
```

In [9]:

```
#Brief look over the data and it starts in 2010
df.head()
```

Out[9]:

	date	open	low	high	volume	Close
691	2010-01-04	31.328709	31.037932	31.375608	16176600.0	31.216151
1159	2010-01-05	31.300569	30.747158	31.328709	23722900.0	31.272429
1627	2010-01-06	30.615841	29.827933	30.709639	37506400.0	29.940490
2095	2010-01-07	30.109329	29.574676	30.109329	25508200.0	29.762274
2563	2010-01-08	29.696614	29.471497	29.809172	20658300.0	29.781033

In [10]:

```
#Looking at the end of the data and it ends in December 2016
df.tail()
```

Out[10]:

	date	open	low	high	volume	Close
849233	2016-12-23	53.650002	53.439999	53.750000	7201200.0	53.680000
849733	2016-12-27	53.689999	53.439999	53.830002	6255400.0	53.639999
850233	2016-12-28	53.630001	53.369999	53.830002	6849500.0	53.439999
850733	2016-12-29	53.590000	53.509998	53.900002	7150800.0	53.740002
851233	2016-12-30	53.549999	53.169998	53.639999	12440800.0	53.380001

In [11]:

```
#Looking at the trend of the stock from start to finish and excluding volume
temp = df[['date','open','low','high', 'Close']]
temp.plot(figsize=(23,15))
```

Out[11]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x20f33994488>
```

Feature Creation

This section of the note book is where the data is modified so that it may be interpreted by the computer in order to perform machine learning

Due to the nature of the dataset, there are no null values which means there is no need to input any missing values. However I do feel it is important to address my actions if there were missing values.

- To address the missing open stock prices I will make use of the previous days closed price as they are usually close enough and good enough to not punish the machine learning algorithm with a complete zero but still have a value that is relative to the stock.
- The same for missing closed stock prices, I will make use of the next days opening price as a value.
- Addressing low and High values, one would make use of the open and closed stock prices in a single day to see which is higher and lower. The the lowest stock price will be the lower of the two minus a certain interval like -0.8 dollars and the same for the higher value. Again this is to not penalise the learning algorithm too much.
- Addressing missing data values, this would make use of looking at the previous date and the next date and averaging it assuming that the missing date is inbetween them.
- Addressing missing volume values will make use of the previous and next day values and average it to an integer.
- Any other missing values will have to be manually inserted using appropriate values according to its sitation that can arise.

The above points (excluding the last one) will make use of a function that iterates through the entire data frame in order to change every value if there were a large amount of missing values. The following code is an example of such a function that would achieve the above suggestion:

```
#Iterates through the rows of the data and fixes missing values accordingly
for x in range(len(data)):
    #Assigns values to the index as well as each column in order to make it easi
er to work with the data
    index = data.iloc[x].date
    open = data['open'].iloc[x]
    close = data['Close'].iloc[x]
    volume = data['volume'].iloc[x]

    if "NaN" in open:
        data.at[index, 'open'] = data['Close'].iloc[x-1]  #Previous day close

    if "NaN" in close:
        data.at[index, 'Close'] = data['open'].iloc[x+1]  #next day open

    if "NaN" in volume:
        #Retrieves next day and previous day volumes and places the average
        previous_Volume = data['volume'].iloc[x-1]
        next_Volume = data['volume'].iloc[x+1]
        data.at[index, 'volume'] = (previous_Volume + next_Volume)//2 # Using in
teger division of average volume
```

This function is not exhaustive but it does demonstrate the manner in which the values can be placed in to help clean up the data.

Normalise the data

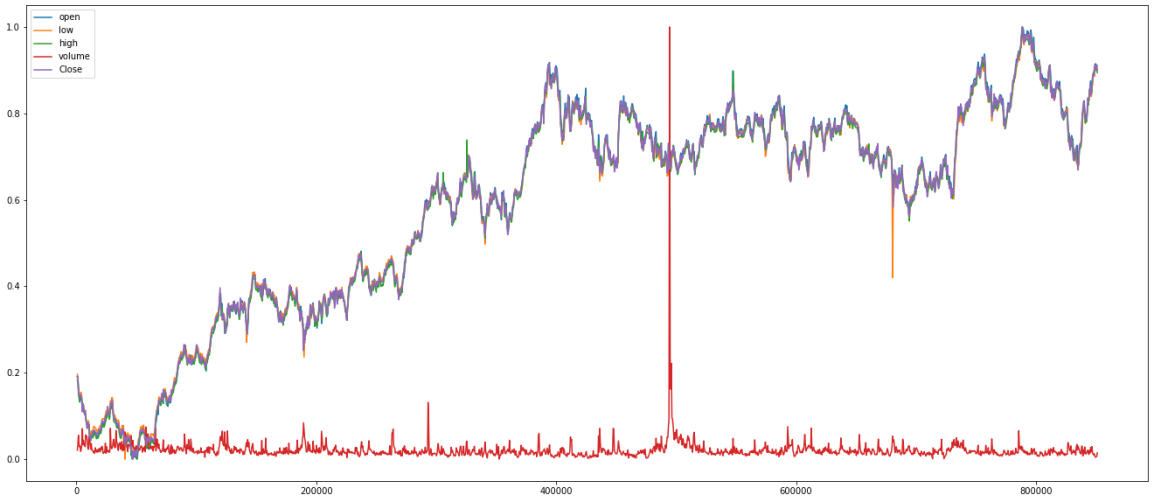
The data is normalized according to each column and plotted to see visually if this was successful

In [12]:

```
#Creating a normalise function and using the MinMaxScaler Library
def normalize_data(df):
    min_max_scaler = preprocessing.MinMaxScaler()
    df['open'] = min_max_scaler.fit_transform(df.open.values.reshape(-1,1))
    df['high'] = min_max_scaler.fit_transform(df.high.values.reshape(-1,1))
    df['low'] = min_max_scaler.fit_transform(df.low.values.reshape(-1,1))
    df['volume'] = min_max_scaler.fit_transform(df.volume.values.reshape(-1,1))
    df['Close'] = min_max_scaler.fit_transform(df.Close.values.reshape(-1,1))
    return df
df = normalize_data(df)
df.plot(figsize=(23,10))
```

Out[12]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x20f348ba148>
```



Creating test and training set

In [13]:

```
#Creates a function to create the test and training data with the stock df and sequence
length as parameters
def load_data(stock, seq_len):
    amount_of_features = len(stock.columns) # The features are the columns excluding da
te
    data = stock.to_numpy() #changes the code over to a matrix
    sequence_length = seq_len + 1 # adjusts the index
    result = []  # Empty list to store the full data

    for index in range(len(data) - sequence_length): # Checks to see how far we are fro
m the most recent date
        result.append(data[index: index + sequence_length]) # Appending the data to res
ult

    result = np.array(result)
    row = round(0.8 * result.shape[0]) # 80% split for test and training data
    train = result[:int(row), :] # Creates a train list with all the training data in i
t including the x and y values

    x_train = train[:, :-1] #Extracts the training x and y values from the train list
    y_train = train[:, -1][:,-1]

    x_test = result[int(row):, :-1] #Extracts the test data from result list
    y_test = result[int(row):, -1][:,-1]

    x_train = np.reshape(x_train, (x_train.shape[0], x_train.shape[1], amount_of_featur
es))  #Transfroms the shape of the data to be ready for
    x_test = np.reshape(x_test, (x_test.shape[0], x_test.shape[1], amount_of_features))

    return [x_train, y_train, x_test, y_test]   #Returns the test and training data fro
m the function to be used in the LSTM
```

Build the model

In [14]:

```
#Creates a new data frame containing just the features and excludes the date
new_df = df[['open', 'low', 'high', 'volume', 'Close']]
new_df.head()
#Quick overview of the data going to be used in the model
```

Out[14]:

	open	low	high	volume	Close
691	0.191481	0.196700	0.182335	0.020875	0.190476
1159	0.190571	0.187461	0.180835	0.033180	0.192276
1627	0.168419	0.158254	0.161043	0.055656	0.149681
2095	0.152032	0.150207	0.141849	0.036091	0.143982
2563	0.138680	0.146929	0.132253	0.028183	0.144582

In [15]:

```
new_df.tail()
```

Out[15]:

	open	low	high	volume	Close
849233	0.913620	0.908492	0.897689	0.006239	0.908859
849733	0.914914	0.908492	0.900247	0.004696	0.907579
850233	0.912973	0.906268	0.900247	0.005665	0.901183
850733	0.911679	0.910716	0.902485	0.006156	0.910777
851233	0.910385	0.899913	0.894173	0.014783	0.899265

In [16]:

```
new_df.shape
```

Out[16]:

```
(1762, 5)
```

Building the LSTM Model

This model is adjusted and tweaked according to the performance of the model

The compiled model makes use of the RMSprop optimiser as this paper (http://www.ijitee.org/wp-content/uploads/papers/v9i3/C8026019320.pdf) has shown better accuracy with its use. Mean Squared error is used to find the the line of best fit to the predicted points and is most appropriate.

In [17]:

```
def build_model(layers):
    d = 0.2 #drop rate of 20% This value is adjusted according to the fit of the data s
ince the memory use is not expensive
    LSTM_input = 50
    model = Sequential() #Creates a sequential model stored in the variable 'model'

    #Adds LSTMs to the sequential model and dropsout 20% of the neurons after each iter
ation
    model.add(LSTM(LSTM_input, input_shape=(layers[1], layers[0]), return_sequences=Tru
e))
    model.add(Dropout(d))
    model.add(Dense(64))

    model.add(LSTM(LSTM_input, input_shape=(layers[1], layers[0]), return_sequences=Fal
se))

    model.add(Dropout(d))

    # ACitvation function 'Recitifed linear unit' appropriate due to the non-linear nat
ure of the stock market.
    model.add(Dense(1,kernel_initializer="uniform",activation='relu'))

    opt = tf.keras.optimizers.RMSprop(
    learning_rate=0.001,
    rho=0.9,
    momentum=0.0,
    epsilon=1e-07,
    centered=False,
    name="RMSprop",
    )
    start = time.time()
    model.compile(loss='mse',optimizer= opt, metrics=['accuracy'])
    return model
```

Retrieving data to train and considering a window period of x days

In [18]:

```
window = 15 #x is the values of day intervals to be changed according to the performanc
e of the model
X_train, y_train, X_test, y_test = load_data(new_df, window)
print (X_train[0], y_train[0])
```

```
[[0.19148141 0.19670026 0.1823349  0.02087452 0.19047638]
 [0.19057104 0.18746136 0.18083543 0.03318004 0.19227613]
 [0.1684187  0.15825435 0.16104251 0.05565638 0.1496814 ]
 [0.15203201 0.15020749 0.14184938 0.03609128 0.14398214]
 [0.13867988 0.14692914 0.13225277 0.0281827  0.14458204]
 [0.14808709 0.14692914 0.13795079 0.03032963 0.14848154]
 [0.14232135 0.154976   0.13645126 0.01798971 0.14938146]
 [0.14748014 0.15408191 0.14004999 0.02239447 0.14818161]
 [0.14353518 0.14037247 0.13495182 0.03208962 0.12868404]
 [0.12472083 0.11802016 0.11845771 0.07055385 0.1094864 ]
 [0.1107618  0.1239808  0.11845771 0.03352689 0.128984  ]
 [0.12563123 0.12249067 0.11905752 0.04380337 0.1133859 ]
 [0.1122791  0.11235762 0.10796147 0.03293463 0.11098625]
 [0.10560304 0.11354974 0.1067619  0.03400467 0.10228733]
 [0.11258258 0.11265565 0.11066052 0.02890834 0.11248604]] 0.0971879690475
4112
```

In [19]:

```
model = build_model([5,window,1])
```

In [20]:

```
#Setting the epochs and batch size. This is according to model performance as well as c
omputational cost.
epoch = 10
batch = 32
```

In [21]:

```
model.fit(X_train,y_train,batch_size=batch,epochs=epoch,validation_split=0.1,verbose=1)
```

```
Train on 1257 samples, validate on 140 samples
Epoch 1/10
1257/1257 [==============================] - 1s 916us/step - loss: 0.0293
- accuracy: 7.9554e-04 - val_loss: 0.0023 - val_accuracy: 0.0000e+00
Epoch 2/10
1257/1257 [==============================] - 0s 367us/step - loss: 0.0072
- accuracy: 7.9554e-04 - val_loss: 0.0051 - val_accuracy: 0.0000e+00
Epoch 3/10
1257/1257 [==============================] - 0s 354us/step - loss: 0.0064
- accuracy: 7.9554e-04 - val_loss: 7.6723e-04 - val_accuracy: 0.0000e+00
Epoch 4/10
1257/1257 [==============================] - 0s 351us/step - loss: 0.0060
- accuracy: 7.9554e-04 - val_loss: 7.7737e-04 - val_accuracy: 0.0000e+00
Epoch 5/10
1257/1257 [==============================] - 0s 363us/step - loss: 0.0056
- accuracy: 7.9554e-04 - val_loss: 0.0010 - val_accuracy: 0.0000e+00
Epoch 6/10
1257/1257 [==============================] - 0s 355us/step - loss: 0.0052
- accuracy: 7.9554e-04 - val_loss: 0.0084 - val_accuracy: 0.0000e+00
Epoch 7/10
1257/1257 [==============================] - 0s 356us/step - loss: 0.0055
- accuracy: 7.9554e-04 - val_loss: 0.0024 - val_accuracy: 0.0000e+00
Epoch 8/10
1257/1257 [==============================] - 0s 355us/step - loss: 0.0045
- accuracy: 7.9554e-04 - val_loss: 0.0021 - val_accuracy: 0.0000e+00
Epoch 9/10
1257/1257 [==============================] - 0s 355us/step - loss: 0.0044
- accuracy: 7.9554e-04 - val_loss: 0.0010 - val_accuracy: 0.0000e+00
Epoch 10/10
1257/1257 [==============================] - 0s 351us/step - loss: 0.0043
- accuracy: 7.9554e-04 - val_loss: 0.0015 - val_accuracy: 0.0000e+00
```

Out[21]:

```
<keras.callbacks.callbacks.History at 0x20f3a060c48>
```

In [22]:

```
diff=[]
ratio=[]
predict = model.predict(X_test)
print (predict.shape)
# for each data index in test data
for i in range(len(y_test)):
    temp = predict[i][0]
    ratio.append((y_test[i]/temp)-1)
    diff.append(abs(y_test[i]- temp))
```

```
(349, 1)
```

In [23]:

```
#This block of code imports the data again and denormalises the data to retrieve the ac
tual values and not the normalised values
df = pd.read_csv(DATA_URL)
df["Close"] = df.close # Moving close to the last column
df.drop(['close'], 1, inplace=True) # Moving close to the last column
df = df[df.symbol == company_code] #Looking only at the company in question
df.drop(['symbol'],1,inplace=True)

# Denormalised function
def denormalize(df, normalized_value):
    df = df['Close'].values.reshape(-1,1)
    normalized_value = normalized_value.reshape(-1,1)

    min_max_scaler = preprocessing.MinMaxScaler()
    a = min_max_scaler.fit_transform(df)
    new = min_max_scaler.inverse_transform(normalized_value)
    return new

Prediction = denormalize(df, predict)
Test = denormalize(df, y_test)
```

In [24]:

```
# Checking the model score using the Mean sequared error and root mean squared error me
thod
def model_score(model, X_train, y_train, X_test, y_test):
    trainScore = model.evaluate(X_train, y_train, verbose=0)
    print('Train Score: %.5f MSE (%.2f RMSE)' % (trainScore[0], math.sqrt(trainScore[0
])))

    testScore = model.evaluate(X_test, y_test, verbose=0)
    print('Test Score: %.5f MSE (%.2f RMSE)' % (testScore[0], math.sqrt(testScore[0])))
    return trainScore[0], testScore[0]


trainscore, testscore = model_score(model, X_train, y_train, X_test, y_test)
train_score = str(round(trainscore, 5)) + " - Train MSE"
test_score = str(round(testscore, 5)) + " - Test MSE"
```

```
Train Score: 0.00140 MSE (0.04 RMSE)
Test Score: 0.00614 MSE (0.08 RMSE)
```

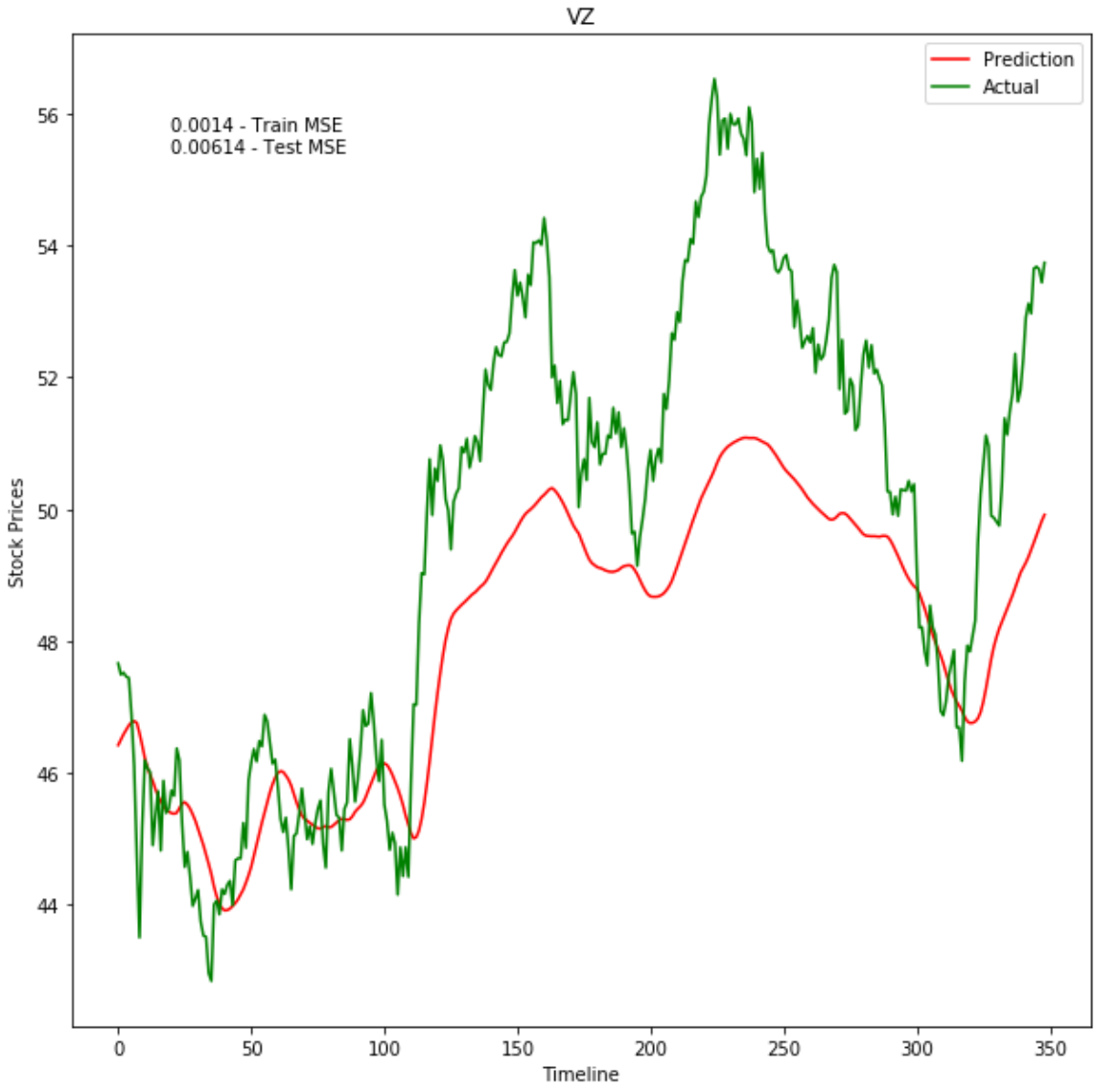
In [25]:

```
# Plotting the predicted and actual values using pyplot

plt.figure(figsize=(10,10))
plt.plot(Prediction,color='Red', label='Prediction')
plt.plot(Test,color='Green', label='Actual')
Train_MSE = mpatches.Patch(color='blue', label=train_score)
Test_MSE = mpatches.Patch(color='yellow', label=test_score)
# Add title, axis names and Legend

extraString = train_score + '\n' + test_score
plt.figtext(0.2, 0.79, extraString)


plt.title(company_code)
plt.xlabel('Timeline')
plt.ylabel('Stock Prices')
plt.legend()
plt.show()
```



Final Comments

The graph is definitely able to interpret the trends and gives an excellent RMSE and MSE score. This model can be used to predict the trends of stock prices of any stock on the NYSE.

In [ ]: