```
 1 import pandas as pd
 2 import datetime
 3 from pandas_datareader import data as pdr
 4 import fix_yahoo_finance as yf
 5 import matplotlib.pyplot as plt
 6
 7
 8 # Compute the Bollinger Bands
 9 def BBANDS(data, ndays):
10     data_close = data['Close']
11     MA = pd.Series(data_close.rolling(window=ndays, center=
   False).mean())
12     SD = pd.Series(data_close.rolling(window=ndays, center=
   False).std())
13     b1 = MA + (2 * SD)
14     B1 = pd.Series(b1, name='Upper BollingerBand')
15     data = data.join(B1)
16
17     b2 = MA - (2 * SD)
18     B2 = pd.Series(b2, name='Lower BollingerBand')
19     data = data.join(B2)
20
21     return data
22
23
24 # Commodity Channel Index
25 def CCI(data, ndays):
26  TP = (data['High'] + data['Low'] + data['Close']) / 3
27  #print TP.rolling(window=ndays,center=False).mean()
28  CCI = pd.Series((TP - TP.rolling(window=ndays,center=False
   ).mean()) / (0.015 * TP.rolling(window=ndays,center=False).
   std()),
29  name = 'CCI')
30  data = data.join(CCI)
31  return data
32
33
34 # Ease of Movement
35 def EVM(data, ndays):
36  dm = ((data['High'] + data['Low'])/2) - ((data['High'].
   shift(1) + data['Low'].shift(1))/2)
37  br = (data['Volume'] / 100000000) / ((data['High'] - data[
   'Low']))
38  EVM = dm / br
39  EVM_MA = pd.Series(EVM.rolling(window=ndays,center=False).
   mean(), name = 'EVM')
40  data = data.join(EVM_MA)
41  return data
42
43
```

```
44 # Force Index
45 def ForceIndex(data, ndays):
46  FI = pd.Series(data['Close'].diff(ndays) * data['Volume'],
    name = 'ForceIndex')
47  data = data.join(FI)
48  return data
49
50
51 # Simple Moving Average
52 def SMA(data, ndays):
53  data_close = data["Close"]
54  SMA = pd.Series(data_close.rolling(window=ndays,center=
   False).mean(), name = 'SMA')
55  data = data.join(SMA)
56  return data
57
58
59 # Exponentially-weighted Moving Average
60 def EWMA(data, ndays):
61  data_close = data["Close"]
62  EMA = pd.Series(data_close.ewm(ignore_na=False,span=ndays,
   min_periods=ndays-1,adjust=True).mean(),
63  name = 'EWMA_' + str(ndays))
64  data = data.join(EMA)
65  return data
66
67
68 # Rate of Change (ROC)
69 def ROC(data,n):
70  N = data['Close'].diff(n)
71  D = data['Close'].shift(n)
72  ROC = pd.Series(N/D,name='Rate of Change')
73  data = data.join(ROC)
74  return data
75
76
77
78
79 ###############
80 ####from Bruno Franca and Peter Bakker, I refer from https
   ://www.quantopian.com/posts/technical-analysis-indicators-
   without-talib-code
81 #############
82
83
84 #Moving Average
85 def MA(df, n):
86     MA = pd.Series(pd.rolling_mean(df['Close'], n), name =
   'MA_' + str(n))
87     df = df.join(MA)
```

```
 88     return df
 89
 90 #Exponential Moving Average
 91 def EMA(df, n):
 92     EMA = pd.Series(pd.ewma(df['Close'], span = n,
    min_periods = n - 1), name = 'EMA_' + str(n))
 93     df = df.join(EMA)
 94     return df
 95
 96 #Momentum
 97 def MOM(df, n):
 98     M = pd.Series(df['Close'].diff(n), name = 'Momentum_'
    + str(n))
 99     df = df.join(M)
100     return df
101
102 #Rate of Change
103 def ROC(df, n):
104     M = df['Close'].diff(n - 1)
105     N = df['Close'].shift(n - 1)
106     ROC = pd.Series(M / N, name = 'ROC_' + str(n))
107     df = df.join(ROC)
108     return df
109
110 #Average True Range
111 def ATR(df, n):
112     i = 0
113     TR_l = [0]
114     while i < df.index[-1]:
115         TR = max(df.get_value(i + 1, 'High'), df.get_value
    (i, 'Close')) - min(df.get_value(i + 1, 'Low'), df.
    get_value(i, 'Close'))
116         TR_l.append(TR)
117         i = i + 1
118     TR_s = pd.Series(TR_l)
119     ATR = pd.Series(pd.ewma(TR_s, span = n, min_periods =
    n), name = 'ATR_' + str(n))
120     df = df.join(ATR)
121     return df
122
123 # #Bollinger Bands
124 # def BBANDS(df, n):
125 #     MA = pd.Series(pd.rolling_mean(df['Close'], n))
126 #     MSD = pd.Series(pd.rolling_std(df['Close'], n))
127 #     b1 = 4 * MSD / MA
128 #     B1 = pd.Series(b1, name = 'BollingerB_' + str(n))
129 #     df = df.join(B1)
130 #     b2 = (df['Close'] - MA + 2 * MSD) / (4 * MSD)
131 #     B2 = pd.Series(b2, name = 'Bollinger%b_' + str(n))
132 #     df = df.join(B2)
```

```
133 #     return df
134 
135 #Pivot Points, Supports and Resistances
136 def PPSR(df):
137     PP = pd.Series((df['High'] + df['Low'] + df['Close'])
    / 3)
138     R1 = pd.Series(2 * PP - df['Low'])
139     S1 = pd.Series(2 * PP - df['High'])
140     R2 = pd.Series(PP + df['High'] - df['Low'])
141     S2 = pd.Series(PP - df['High'] + df['Low'])
142     R3 = pd.Series(df['High'] + 2 * (PP - df['Low']))
143     S3 = pd.Series(df['Low'] - 2 * (df['High'] - PP))
144     psr = {'PP':PP, 'R1':R1, 'S1':S1, 'R2':R2, 'S2':S2, '
    R3':R3, 'S3':S3}
145     PSR = pd.DataFrame(psr)
146     df = df.join(PSR)
147     return df
148 
149 #Stochastic oscillator %K
150 def STOK(df):
151     SOk = pd.Series((df['Close'] - df['Low']) / (df['High'
    ] - df['Low']), name = 'SO%k')
152     df = df.join(SOk)
153     return df
154 
155 #Stochastic oscillator %D
156 def STO(df, n):
157     SOk = pd.Series((df['Close'] - df['Low']) / (df['High'
    ] - df['Low']), name = 'SO%k')
158     SOd = pd.Series(SOk.ewm(ignore_na=False,span=n,
    min_periods=n-1,adjust=True).mean() , name = 'SO%d_' + str
    (n))
159     df = df.join(SOd)
160     return df
161 
162 #Trix
163 def TRIX(df, n):
164     EX1 = pd.ewma(df['Close'], span = n, min_periods = n -
     1)
165     EX2 = pd.ewma(EX1, span = n, min_periods = n - 1)
166     EX3 = pd.ewma(EX2, span = n, min_periods = n - 1)
167     i = 0
168     ROC_l = [0]
169     while i + 1 <= df.index[-1]:
170         ROC = (EX3[i + 1] - EX3[i]) / EX3[i]
171         ROC_l.append(ROC)
172         i = i + 1
173     Trix = pd.Series(ROC_l, name = 'Trix_' + str(n))
174     df = df.join(Trix)
175     return df
```

```
176
177 #Average Directional Movement Index
178 def ADX(df, n, n_ADX):
179     i = 0
180     UpI = []
181     DoI = []
182     while i + 1 <= df.index[-1]:
183         UpMove = df.get_value(i + 1, 'High') - df.
    get_value(i, 'High')
184         DoMove = df.get_value(i, 'Low') - df.get_value(i +
     1, 'Low')
185         if UpMove > DoMove and UpMove > 0:
186             UpD = UpMove
187         else: UpD = 0
188         UpI.append(UpD)
189         if DoMove > UpMove and DoMove > 0:
190             DoD = DoMove
191         else: DoD = 0
192         DoI.append(DoD)
193         i = i + 1
194     i = 0
195     TR_l = [0]
196     while i < df.index[-1]:
197         TR = max(df.get_value(i + 1, 'High'), df.get_value
    (i, 'Close')) - min(df.get_value(i + 1, 'Low'), df.
    get_value(i, 'Close'))
198         TR_l.append(TR)
199         i = i + 1
200     TR_s = pd.Series(TR_l)
201     ATR = pd.Series(pd.ewma(TR_s, span = n, min_periods =
    n))
202     UpI = pd.Series(UpI)
203     DoI = pd.Series(DoI)
204     PosDI = pd.Series(pd.ewma(UpI, span = n, min_periods =
     n - 1) / ATR)
205     NegDI = pd.Series(pd.ewma(DoI, span = n, min_periods =
     n - 1) / ATR)
206     ADX = pd.Series(pd.ewma(abs(PosDI - NegDI) / (PosDI +
    NegDI), span = n_ADX, min_periods = n_ADX - 1), name = '
    ADX_' + str(n) + '_' + str(n_ADX))
207     df = df.join(ADX)
208     return df
209
210 #MACD, MACD Signal and MACD difference
211 def MACD(df, n_fast, n_slow):
212     data_close = df['Close']
213     EMAfast = pd.Series(data_close.ewm(span=n_fast,
    min_periods=n_slow-1,adjust=True,ignore_na=False).mean())
214     EMAslow = pd.Series(data_close.ewm(span=n_slow,
    min_periods=n_slow-1,adjust=True,ignore_na=False).mean())
```

```
215     MACD = pd.Series(EMAfast - EMAslow, name = 'MACD_' +
    str(n_fast) + '_' + str(n_slow))
216     MACDsign = pd.Series(MACD.ewm(span=9, min_periods=8,
    adjust=True,ignore_na=False).mean(), name = 'MACDsign_' +
    str(n_fast) + '_' + str(n_slow))
217     MACDdiff = pd.Series(MACD - MACDsign, name = '
    MACDdiff_' + str(n_fast) + '_' + str(n_slow))
218     df = df.join(MACD)
219     df = df.join(MACDsign)
220     df = df.join(MACDdiff)
221     return df
222
223
224
225 #Mass Index
226 def MassI(df):
227     Range = df['High'] - df['Low']
228     EX1 = pd.ewma(Range, span = 9, min_periods = 8)
229     EX2 = pd.ewma(EX1, span = 9, min_periods = 8)
230     Mass = EX1 / EX2
231     MassI = pd.Series(pd.rolling_sum(Mass, 25), name = '
    Mass Index')
232     df = df.join(MassI)
233     return df
234
235 #Vortex Indicator: http://www.vortexindicator.com/
    VFX_VORTEX.PDF
236 def Vortex(df, n):
237     i = 0
238     TR = [0]
239     while i < df.index[-1]:
240         Range = max(df.get_value(i + 1, 'High'), df.
    get_value(i, 'Close')) - min(df.get_value(i + 1, 'Low'),
    df.get_value(i, 'Close'))
241         TR.append(Range)
242         i = i + 1
243     i = 0
244     VM = [0]
245     while i < df.index[-1]:
246         Range = abs(df.get_value(i + 1, 'High') - df.
    get_value(i, 'Low')) - abs(df.get_value(i + 1, 'Low') - df
    .get_value(i, 'High'))
247         VM.append(Range)
248         i = i + 1
249     VI = pd.Series(pd.rolling_sum(pd.Series(VM), n) / pd.
    rolling_sum(pd.Series(TR), n), name = 'Vortex_' + str(n))
250     df = df.join(VI)
251     return df
252
253
```

```
254
255
256
257 #KST Oscillator
258 def KST(df, r1, r2, r3, r4, n1, n2, n3, n4):
259     M = df['Close'].diff(r1 - 1)
260     N = df['Close'].shift(r1 - 1)
261     ROC1 = M / N
262     M = df['Close'].diff(r2 - 1)
263     N = df['Close'].shift(r2 - 1)
264     ROC2 = M / N
265     M = df['Close'].diff(r3 - 1)
266     N = df['Close'].shift(r3 - 1)
267     ROC3 = M / N
268     M = df['Close'].diff(r4 - 1)
269     N = df['Close'].shift(r4 - 1)
270     ROC4 = M / N
271     KST = pd.Series(pd.rolling_sum(ROC1, n1) + pd.
    rolling_sum(ROC2, n2) * 2 + pd.rolling_sum(ROC3, n3) * 3 +
     pd.rolling_sum(ROC4, n4) * 4, name = 'KST_' + str(r1) + '
    _' + str(r2) + '_' + str(r3) + '_' + str(r4) + '_' + str(
    n1) + '_' + str(n2) + '_' + str(n3) + '_' + str(n4))
272     df = df.join(KST)
273     return df
274
275 #Relative Strength Index touble!!
276 def RSI(df, n):
277     i = 0
278     UpI = [0]
279     DoI = [0]
280
281     for i in range(len(df.index)-1):
282         # print range(len(df.index)-1)[-1]
283         # print i
284         UpMove = df.loc[df.index[i+1]]['High'] - df.loc[df
    .index[i]]['High']
285         DoMove = df.loc[df.index[i+1]]['Low'] - df.loc[df.
    index[i]]['Low']
286         if UpMove > DoMove and UpMove > 0:
287             UpD = UpMove
288         else: UpD = 0
289         UpI.append(UpD)
290         if DoMove > UpMove and DoMove > 0:
291             DoD = DoMove
292         else: DoD = 0
293         DoI.append(DoD)
294
295     UpI = pd.Series(UpI)
296     DoI = pd.Series(DoI)
297     PosDI = pd.Series(UpI.ewm(span=n, min_periods=n-1,
```

```
297 adjust=True,ignore_na=True).mean())
298     NegDI = pd.Series(DoI.ewm(span=n, min_periods=n-1,
    adjust=True,ignore_na=True).mean())
299     RSI = pd.Series(PosDI / (PosDI + NegDI), name = 'RSI_'
     + str(n))
300     df = df.join(RSI)
301     return df
302
303 #True Strength Index
304 def TSI(df, r, s):
305     M = pd.Series(df['Close'].diff(1))
306     aM = abs(M)
307     EMA1 = pd.Series(pd.ewma(M, span = r, min_periods = r
    - 1))
308     aEMA1 = pd.Series(pd.ewma(aM, span = r, min_periods =
    r - 1))
309     EMA2 = pd.Series(pd.ewma(EMA1, span = s, min_periods =
     s - 1))
310     aEMA2 = pd.Series(pd.ewma(aEMA1, span = s, min_periods
     = s - 1))
311     TSI = pd.Series(EMA2 / aEMA2, name = 'TSI_' + str(r) +
     '_' + str(s))
312     df = df.join(TSI)
313     return df
314
315 #Accumulation/Distribution
316 def ACCDIST(df, n):
317     ad = (2 * df['Close'] - df['High'] - df['Low']) / (df[
    'High'] - df['Low']) * df['Volume']
318     M = ad.diff(n - 1)
319     N = ad.shift(n - 1)
320     ROC = M / N
321     AD = pd.Series(ROC, name = 'Acc/Dist_ROC_' + str(n))
322     df = df.join(AD)
323     return df
324
325 #Chaikin Oscillator
326 def Chaikin(df):
327     ad = (2 * df['Close'] - df['High'] - df['Low']) / (df[
    'High'] - df['Low']) * df['Volume']
328     Chaikin = pd.Series(ad.ewm(ignore_na=False,span=3,
    min_periods=3-1,adjust=True).mean() - ad.ewm(ignore_na=
    False,span=10,min_periods=10-1,adjust=True).mean(), name =
     'Chaikin')
329     df = df.join(Chaikin)
330     return df
331
332 #Money Flow Index and Ratio
333 def MFI(df, n):
334     PP = (df['High'] + df['Low'] + df['Close']) / 3
```

```
335     i = 0
336     PosMF = [0]
337     while i < df.index[-1]:
338         if PP[i + 1] > PP[i]:
339             PosMF.append(PP[i + 1] * df.get_value(i + 1, '
    Volume'))
340         else:
341             PosMF.append(0)
342         i = i + 1
343     PosMF = pd.Series(PosMF)
344     TotMF = PP * df['Volume']
345     MFR = pd.Series(PosMF / TotMF)
346     MFI = pd.Series(pd.rolling_mean(MFR, n), name = 'MFI_'
     + str(n))
347     df = df.join(MFI)
348     return df
349
350 #On-balance Volume
351 def OBV(df, n):
352     i = 0
353     OBV = [0]
354     while i < df.index[-1]:
355         if df.get_value(i + 1, 'Close') - df.get_value(i,
    'Close') > 0:
356             OBV.append(df.get_value(i + 1, 'Volume'))
357         if df.get_value(i + 1, 'Close') - df.get_value(i,
    'Close') == 0:
358             OBV.append(0)
359         if df.get_value(i + 1, 'Close') - df.get_value(i,
    'Close') < 0:
360             OBV.append(-df.get_value(i + 1, 'Volume'))
361         i = i + 1
362     OBV = pd.Series(OBV)
363     OBV_ma = pd.Series(pd.rolling_mean(OBV, n), name = '
    OBV_' + str(n))
364     df = df.join(OBV_ma)
365     return df
366
367 #Force Index
368 def FORCE(df, n):
369     F = pd.Series(df['Close'].diff(n) * df['Volume'].diff(
    n), name = 'Force_' + str(n))
370     df = df.join(F)
371     return df
372
373 #Ease of Movement
374 def EOM(df, n):
375     EoM = (df['High'].diff(1) + df['Low'].diff(1)) * (df['
    High'] - df['Low']) / (2 * df['Volume'])
376     Eom_ma = pd.Series(pd.rolling_mean(EoM, n), name = '
```

```
376 EoM_' + str(n))
377     df = df.join(Eom_ma)
378     return df
379 
380 # #Commodity Channel Index
381 # def CCI(df, n):
382 #     PP = (df['High'] + df['Low'] + df['Close']) / 3
383 #     CCI = pd.Series((PP - pd.rolling_mean(PP, n)) / pd.
    rolling_std(PP, n), name = 'CCI_' + str(n))
384 #     df = df.join(CCI)
385 #     return df
386 
387 #Coppock Curve
388 def COPP(df, n):
389     M = df['Close'].diff(int(n * 11 / 10) - 1)
390     N = df['Close'].shift(int(n * 11 / 10) - 1)
391     ROC1 = M / N
392     M = df['Close'].diff(int(n * 14 / 10) - 1)
393     N = df['Close'].shift(int(n * 14 / 10) - 1)
394     ROC2 = M / N
395     Copp = pd.Series(pd.ewma(ROC1 + ROC2, span = n,
    min_periods = n), name = 'Copp_' + str(n))
396     df = df.join(Copp)
397     return df
398 
399 #Keltner Channel
400 def KELCH(df, n):
401     KelChM = pd.Series(pd.rolling_mean((df['High'] + df['
    Low'] + df['Close']) / 3, n), name = 'KelChM_' + str(n))
402     KelChU = pd.Series(pd.rolling_mean((4 * df['High'] - 2
     * df['Low'] + df['Close']) / 3, n), name = 'KelChU_' +
    str(n))
403     KelChD = pd.Series(pd.rolling_mean((-2 * df['High'] +
    4 * df['Low'] + df['Close']) / 3, n), name = 'KelChD_' +
    str(n))
404     df = df.join(KelChM)
405     df = df.join(KelChU)
406     df = df.join(KelChD)
407     return df
408 
409 #Ultimate Oscillator
410 def ULTOSC(df):
411     i = 0
412     TR_l = [0]
413     BP_l = [0]
414     while i < df.index[-1]:
415         TR = max(df.get_value(i + 1, 'High'), df.get_value
    (i, 'Close')) - min(df.get_value(i + 1, 'Low'), df.
    get_value(i, 'Close'))
416         TR_l.append(TR)
```

```
417         BP = df.get_value(i + 1, 'Close') - min(df.
    get_value(i + 1, 'Low'), df.get_value(i, 'Close'))
418         BP_l.append(BP)
419         i = i + 1
420     UltO = pd.Series((4 * pd.rolling_sum(pd.Series(BP_l),
    7) / pd.rolling_sum(pd.Series(TR_l), 7)) + (2 * pd.
    rolling_sum(pd.Series(BP_l), 14) / pd.rolling_sum(pd.
    Series(TR_l), 14)) + (pd.rolling_sum(pd.Series(BP_l), 28)
    / pd.rolling_sum(pd.Series(TR_l), 28)), name = '
    Ultimate_Osc')
421     df = df.join(UltO)
422     return df
423
424 #Donchian Channel
425 def DONCH(df, n):
426     i = 0
427     DC_l = []
428     while i < n - 1:
429         DC_l.append(0)
430         i = i + 1
431     i = 0
432     while i + n - 1 < df.index[-1]:
433         DC = max(df['High'].ix[i:i + n - 1]) - min(df['Low
    '].ix[i:i + n - 1])
434         DC_l.append(DC)
435         i = i + 1
436     DonCh = pd.Series(DC_l, name = 'Donchian_' + str(n))
437     DonCh = DonCh.shift(n - 1)
438     df = df.join(DonCh)
439     return df
440
441 #Standard Deviation
442 def STDDEV(df, n):
443     df = df.join(pd.Series(pd.rolling_std(df['Close'], n),
     name = 'STD_' + str(n)))
444     return df
```