Trading-FacebookProphet-Model

September 29, 2021

1 Using and Backtesting Facebook Prophet

[189]:
```
# Importing libraries
from fbprophet import Prophet
from fbprophet.diagnostics import cross_validation, performance_metrics
import itertools
import pandas as pd
import numpy as np
import plotly.express as px
from statistics import mean, median
import plotly.graph_objects as go
from tqdm.notebook import tqdm
from sklearn.metrics import mean_squared_error
from datetime import date, timedelta
import yfinance as yf
```

1.1 Getting the Data

[35]:
```
# Getting the date five years ago to download the current timeframe
years = (date.today() - timedelta(weeks=260)).strftime("%Y-%m-%d")

# Stocks to analyze
stocks = ['GE', 'GPRO', 'FIT', 'F']

# Getting the data for multiple stocks
df = yf.download(stocks, start=years).dropna()

print("Rows in DataFrame: ", df.shape[0])
```

```
[**********100%**********]  4 of 4 completed
Rows in DataFrame:  1255
```

[36]:
```
# Storing the dataframes in a dictionary
stock_df = {}

for col in set(df.columns.get_level_values(0)):
```

```
    # Assigning the information (High, Low, etc.) for each stock in the␣
↪dictionary
    stock_df[col] = df[col]
```

2 Preprocessing the Data

[37]:
```
# Finding the log returns
stock_df['LogReturns'] = stock_df['Adj Close'].apply(np.log).diff().dropna()

# Trying out Moving average
stock_df['MovAvg'] = stock_df['Adj Close'].rolling(10).mean().dropna()

# Logarithmic scaling of the data and rounding the result
stock_df['Log'] = stock_df['MovAvg'].apply(np.log).apply(lambda x: round(x, 2))
```

3 Visualizing the Data

[38]:
```
px.line(stock_df['MovAvg'],
        x=stock_df['MovAvg'].index,
        y=stock_df['MovAvg'].columns,
        labels={'variable': 'Stock',
                'value': 'Price'},
        title='Moving Average')
```

[39]:
```
px.line(stock_df['Log'],
        x=stock_df['Log'].index,
        y=stock_df['Log'].columns,
        labels={'variable': 'Stock',
                'value': 'Log Scale'},
        title='Log of Moving Averages')
```

4 Using FBProphet

[253]:
```
def fb_opt_param(data, cv_len=5):
    """
    Finds the best parameters for FBProphet

    Warning: Running this function will take a large amount of time
    """
    param_grid = {
    'changepoint_prior_scale': [0.001, 0.05],
    'seasonality_prior_scale': [0.01, 10],
    }
```

```
    # Generate all combinations of parameters
    all_params = [dict(zip(param_grid.keys(), v)) for v in itertools.
↪product(*param_grid.values())]
    rmses = []  # Store the RMSEs for each params here

    # Use cross validation to evaluate all parameters
    for params in tqdm(all_params):
        m = Prophet(**params,
                    daily_seasonality=True,
                    yearly_seasonality=False).fit(data)  # Fit model with given␣
↪params
        df_cv = cross_validation(m,
                                 initial=f'{len(data)} days',
                                 horizon=f'{cv_len} days',
                                 parallel='processes')
        df_p = performance_metrics(df_cv, rolling_window=1)
        rmses.append(df_p['rmse'].values[0])

    # Find the best parameters
    tuning_results = pd.DataFrame(all_params)
    tuning_results['rmse'] = rmses

    return all_params[np.argmin(rmses)]
```

Formatting the Data to fit to FBProphet's specifications

[254]:
```
proph_df = {}

for stock in stocks:

    # Creating a quick dictionary for the datafram
    d = {'ds': stock_df['MovAvg'][stock].index,
         'y': stock_df['MovAvg'][stock].values}

    # Creating the dataframe
    proph_df[stock] = pd.DataFrame(d)
```

[260]:
```
# Training amount of days
train_days = 50

# Forecasting amount
pred_ahead = 5

# Creating a new DF for the predictions
stock_df['Predictions'] = pd.DataFrame(index=stock_df['MovAvg'].index,
                                       columns=stock_df['MovAvg'].columns)
```

```
for stock in tqdm(stocks):

    # Current predicted value
    pred_val = 0

    # Training the model in a predetermined date range
    for day in tqdm(range(1100,
                          stock_df['MovAvg'].shape[0]-pred_ahead)):

        # Data to use, containing a specific amount of days
        training = proph_df[stock].iloc[day-train_days:day+1].dropna()

        # Determining if the actual value crossed the predicted value
        cross = ((training['y'].iloc[-1] >= pred_val >= training['y'].iloc[-2])␣
↪or
                 (training['y'].iloc[-1] <= pred_val <= training['y'].iloc[-2]))

        # Running the model when the latest training value crosses the␣
↪predicted value or every other day
        if cross or day % 2 == 0:

            # Finding the optimum parameters
            #params = fb_opt_param(training, cv_len=pred_ahead)

            # Instantiating FBprophet
            m = Prophet(interval_width=.95,
                        daily_seasonality=True,
                        weekly_seasonality=True,
                        yearly_seasonality=False)

            # Fitting the model
            m.fit(training)

            # Forecasting prices and getting predictions
            forecast = m.make_future_dataframe(periods=pred_ahead)

            predictions = m.predict(forecast)

            preds = predictions['yhat'].tail(pred_ahead)

            #display(preds)

            # Inserting the predicted values into our own DF
            stock_df['Predictions'][stock].iloc[day:day+pred_ahead] =␣
↪mean(preds.values)
```

```
            # Updating the current predicted value
            pred_val = mean(preds.values)
```

```
HBox(children=(FloatProgress(value=0.0, max=4.0), HTML(value='')))
```

```
HBox(children=(FloatProgress(value=0.0, max=141.0), HTML(value='')))
```

```
HBox(children=(FloatProgress(value=0.0, max=141.0), HTML(value='')))
```

```
HBox(children=(FloatProgress(value=0.0, max=141.0), HTML(value='')))
```

```
HBox(children=(FloatProgress(value=0.0, max=141.0), HTML(value='')))
```

5 Predictions vs Actual Values

[261]:
```
# Shift ahead by 1 to compare the actual values to the predictions
pred_df = stock_df['Predictions'].shift(1).astype(float).dropna()

pred_df
```

[261]:

Date	F	FIT	GE	GPRO
2020-02-27	7.188938	6.015830	11.854247	3.711765
2020-02-28	7.188938	6.015830	11.854247	3.711765
2020-03-02	7.045680	5.993516	11.407551	3.642991
2020-03-03	7.045680	5.993516	11.407551	3.642991
2020-03-04	6.727583	5.965442	10.770034	3.514603
…	…	…	…	…
2020-09-16	6.569301	5.922577	5.755102	3.643297
2020-09-17	6.569301	5.922577	5.755102	3.643297
2020-09-18	6.569301	5.922577	5.755102	3.643297
2020-09-21	6.569301	5.922577	5.755102	3.643297
2020-09-22	6.569301	5.922577	5.755102	3.643297

[145 rows x 4 columns]

5.1 Plotting the Predictions

[262]:
```
for stock in stocks:

    fig = go.Figure()
```

```
    # Plotting the actual values
    fig.add_trace(go.Scatter(x=pred_df.index,
                             y=stock_df['MovAvg'][stock].loc[pred_df.index],
                             name='Actual Moving Average',
                             mode='lines'))

    # Plotting the predicted values
    fig.add_trace(go.Scatter(x=pred_df.index,
                             y=pred_df[stock],
                             name='Predicted Moving Average',
                             mode='lines'))

    # Setting the labels
    fig.update_layout(title=f'Predicting the Moving Average for the Next␣
↪{pred_ahead} days for {stock}',
                      xaxis_title='Date',
                      yaxis_title='Prices')

    fig.show()
```

5.2 Evaluation Metric

[171]:
```
for stock in stocks:

    # Finding the root mean squared error
    rmse = mean_squared_error(stock_df['MovAvg'][stock].loc[pred_df.index],
                              pred_df[stock],
                              squared=False)

    print(f"On average, the model is off by {rmse} for {stock}\n")
```

```
On average, the model is off by 0.44981015370331723 for GE

On average, the model is off by 0.30559565891052454 for GPRO

On average, the model is off by 0.32467678375706926 for FIT

On average, the model is off by 0.2377107701107879 for F
```

6 Trading Signal

[172]:
```
def get_positions(difference, thres=3, short=True):
    """
    Compares the percentage difference between actual values and the respective␣
 ↪predictions.

    Returns the decision or positions to long or short based on the difference.

    Optional: shorting in addition to buying
    """

    if difference > thres/100:

        return 1


    elif short and difference < -thres/100:

        return -1


    else:

        return 0
```

6.1 Creating a Trading DF

[181]:
```
# Creating a DF for trading the model
trade_df = {}

# Getting the percentage difference between the predictions and the actual␣
 ↪values
trade_df['PercentDiff'] = (stock_df['Predictions'].dropna() /
                           stock_df['MovAvg'].loc[stock_df['Predictions'].
 ↪dropna().index]) - 1

# Getting positions
trade_df['Positions'] = trade_df['PercentDiff'].applymap(lambda x:␣
 ↪get_positions(x,
                                                                        ␣
 ↪ thres=.5,
                                                                        ␣
 ↪ short=True) / len(stocks))

# Preventing lookahead bias by shifting the positions
```

```
trade_df['Positions'] = trade_df['Positions'].shift(2).dropna()

# Getting Log Returns
trade_df['LogReturns'] = stock_df['LogReturns'].loc[trade_df['Positions'].index]

display(trade_df['PercentDiff'].tail(20))
display(trade_df['Positions'].tail(20))
```

Date	F	FIT	GE	GPRO
2020-08-24	0.0104959	-0.0092639	0.0112625	-0.0354462
2020-08-25	0.0147221	-0.0111317	0.0134383	-0.0360455
2020-08-26	0.0189837	-0.0129926	0.01719	-0.0344459
2020-08-27	0.0207577	-0.0146922	0.0190763	-0.0300195
2020-08-28	0.0222407	-0.0160784	0.0198643	-0.0283998
2020-08-31	-0.00466611	-0.00145431	-0.00169812	-0.0195454
2020-09-01	-0.00379568	-0.000517003	-0.0011742	-0.0119723
2020-09-02	-0.00495591	-0.000829634	-0.00210565	-0.000765158
2020-09-03	-0.00466611	-0.00036061	-0.00288054	-0.00115163
2020-09-04	-0.00813266	-4.76853e-05	-0.00458105	-0.000599801
2020-09-08	-0.00860074	0.00306488	-0.00322975	-0.00289909
2020-09-09	-0.00903184	0.00479648	-0.000877459	-0.0144604
2020-09-10	-0.0103229	0.00748436	0.000554242	-0.015145
2020-09-11	-0.0116106	0.0100273	0.00902945	-0.0151009
2020-09-14	-0.014176	0.0109842	0.0165025	-0.0189587
2020-09-15	-0.00212866	-0.00278066	-0.00286324	-0.0314918
2020-09-16	-0.00484648	-0.00482285	-0.0118076	-0.0185088
2020-09-17	-0.0095319	-0.00529294	-0.0213697	-0.00857213
2020-09-18	-0.0152921	-0.00591903	-0.0299866	-0.00504661
2020-09-21	-0.0148729	-0.00560608	-0.0289178	-0.00454092

Date	F	FIT	GE	GPRO
2020-08-24	0.00	-0.25	-0.25	0.25
2020-08-25	0.25	-0.25	-0.25	0.25
2020-08-26	0.25	-0.25	0.25	-0.25
2020-08-27	0.25	-0.25	0.25	-0.25
2020-08-28	0.25	-0.25	0.25	-0.25
2020-08-31	0.25	-0.25	0.25	-0.25
2020-09-01	0.25	-0.25	0.25	-0.25
2020-09-02	0.00	0.00	0.00	-0.25
2020-09-03	0.00	0.00	0.00	-0.25
2020-09-04	0.00	0.00	0.00	0.00
2020-09-08	0.00	0.00	0.00	0.00
2020-09-09	-0.25	0.00	0.00	0.00
2020-09-10	-0.25	0.00	0.00	0.00
2020-09-11	-0.25	0.00	0.00	-0.25
2020-09-14	-0.25	0.25	0.00	-0.25

```
2020-09-15 -0.25  0.25  0.25 -0.25
2020-09-16 -0.25  0.25  0.25 -0.25
2020-09-17  0.00  0.00  0.00 -0.25
2020-09-18  0.00  0.00 -0.25 -0.25
2020-09-21 -0.25 -0.25 -0.25 -0.25
```

6.2 Plotting Positions

[182]:
```
# Getting the number of positions
pos = trade_df['Positions'].apply(pd.value_counts)

# Plotting total positions
fig = px.bar(pos,
             x=pos.index,
             y=pos.columns,
             title='Total Positions',
             labels={'variable':'Stocks',
                     'value':'Count of Positions',
                     'index':'Type of Position'})

fig.show()
```

7 Calculating and Plotting Potential Returns

7.1 Returns on Each Individual Stock

[183]:
```
# Calculating Returns by multiplying the positions by the log returns
returns = trade_df['Positions'] * trade_df['LogReturns']

# Calculating the performance as we take the cumulative sum of the returns and␣
 ↪transform the values back to normal
performance = returns.cumsum().apply(np.exp)

# Plotting the performance per stock
px.line(performance,
        x=performance.index,
        y=performance.columns,
        title='Returns Per Stock Using ARIMA Forecast',
        labels={'variable':'Stocks',
                'value':'Returns'})
```

7.2 Returns on Overall Portfolio

[184]:
```
# Returns for the portfolio
returns = (trade_df['Positions'] * trade_df['LogReturns']).sum(axis=1)

# Returns for SPY
```

```
spy = yf.download('SPY', start=returns.index[0]).loc[returns.index]

spy = spy['Adj Close'].apply(np.log).diff().dropna().cumsum().apply(np.exp)

# Calculating the performance as we take the cumulative sum of the returns and␣
 ↪transform the values back to normal
performance = returns.cumsum().apply(np.exp)

# Plotting the comparison between SPY returns and ARIMA returns
fig = go.Figure()

fig.add_trace(go.Scatter(x=spy.index,
                         y=spy,
                         name='SPY Returns',
                         mode='lines'))

fig.add_trace(go.Scatter(x=performance.index,
                         y=performance.values,
                         name='Portfolio Returns',
                         mode='lines'))

fig.update_layout(title='SPY vs ARIMA Overall Portfolio Returns',
                  xaxis_title='Date',
                  yaxis_title='Returns')

fig.show()
```

```
[**********************100%**********************]  1 of 1 completed
```

[ ]:

[ ]: