Table of Contents

- 1 Business Objective
- 2 Methodology
- 3 Collecting stock data
  - 3.1 Stock performance
  - 3.2 Returns
  - 3.3 Beta value
  - 3.4 P/E ratio
  - 3.5 Dividend History
- 4 Summing up performance...
- 5 Time Series Modelling
  - 5.1 Stationarity Check
  - 5.2 Convert non-stationary to stationary
  - 5.3 ACF and PACF
  - 5.4 ARIMA model
  - 5.5 SARIMA model
  - 5.6 Facebook Prophet
    - 5.6.1 Prophet with differencing - comparing with test
    - 5.6.2 regularization on differenced data
  - 5.7 Model performance comparisions
  - 5.8 Using Prophet to get forecast of 'AAPL' for the next year
- 6 Building Portfolio
  - 6.1 Predictions of chosen stock
- 7 Conclusions
  - 7.1 Limitations
  - 7.2 Recommendations

1 Business Objective

To help the average investor build a portfolio of stock and ensure maximum returns. Goals of the investor could range between short-term goals like saving for a dream vacation, down payment of a home etc. to long-term goals like saving for child's education, retirement etc.

The assumption here is that the investor already has an idea where they would like to invest. The aim here is to provide the investor an overview of the company performance and make predictions on future thereby helping the investor decide it it's worth investing in. Since the model is purely mathematical and cannot take into account black swan events, the onus still lies on the investor to look at a company holistically before taking the plunge.

I will also focus on diversification, a core tenet of a good investement strategy i.e by investing in companies across different sectors, the investor can minimize their risk and maximize returns.

2 Methodology

1. Data of the chosen stock from 2017-2022 will be scraped from [Yahoo Finance (https://finance.yahoo.com/)](https://finance.yahoo.com/) using python's `yfinance` (documentation can be found [here (https://aroussi.com/post/python-yahoo-finance)](https://aroussi.com/post/python-yahoo-finance)) and `YahooFinancials` (documentation can be found [here (https://pypi.org/project/yahoofinancials/)](https://pypi.org/project/yahoofinancials/)).
2. Using the data, 4 commonly used metrics to evaluate a stock will be plotted: ***returns, beta ratio, p/e ratio and dividend***
3. Different machine learning models will then be built to predict future stock price. Their errors will be compared and the model with the least error will be used gage future stock performance.
4. Combined with stock performance and forecast information, then by feeding the chosen stock into the portfolio builder, the investor can look at combined returns and decide which portfolio is best-aligned with his/her goals.

3 Collecting stock data

In [1]:
```
#importing libraries

import yfinance as yf
from yahoofinancials import YahooFinancials
import matplotlib.pyplot as plt
plt.style.use('seaborn-darkgrid')
%matplotlib inline
import numpy as np
import pandas as pd
import itertools
from datetime import date
from sklearn.metrics import mean_squared_error
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.model_selection import TimeSeriesSplit
from statsmodels.tsa.stattools import adfuller
from scipy.signal._signaltools import _centered
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.statespace.sarimax import SARIMAX
from fbprophet import Prophet
import warnings
warnings.filterwarnings('ignore')
import logging
logging.basicConfig(level='INFO')
mlogger = logging.getLogger('matplotlib')
mlogger.setLevel(logging.WARNING)
```

3.1 Stock performance

Following is a function that will plot stock prices of a chosen stock from 2017 till date:

In [2]:
```
#function to get stock data of a company
start_date = '2017-01-01'
end_date = date.today()

def stock_info(ticker):
    #get stock prices for the specified date ranges
    df = yf.download(ticker,start=start_date,end=end_date)
    #plot the stock price over the years
    fig,ax = plt.subplots(figsize=(15,5))
    ax.plot(df['Adj Close']);
    ax.set_title(f'Stock Price of {ticker} from 2017 till date')
```

3.2 Returns

Following is a function that plots returns from a chosen stock from 2017 till date:

In [3]:
```
#function to calculate stock return

def stock_return(ticker):
    #get stock prices for the specified date ranges
    df = yf.download(ticker,start=start_date,end=end_date)
    df_return = df[['Adj Close']]
    df_return['pct_change'] = df_return['Adj Close'].pct_change() # use pct ch
    df_return.drop(df_return.index[0],inplace=True) # drop the Nan value from 

    #plot the returns over 5 years
    fig,ax=plt.subplots(figsize=(15,5))
    ax.plot(df_return['pct_change']);
    ax.set_title('Stock Return Pct from 2017 till date:' +' ' + ticker)
```

3.3 Beta value

Beta value of a stock is used to signify risk i.e. if a stock is risky or not. By comparing the stock movement relative to the overall market such as the S&P 500, the stock can be classified as risky or not. By definition, the market has a beta value of 1.0. If the beta value of the stock is greater than 1.0, then it is classified as risky and less so if the value is less than 1.0.

```python
In [4]: #function to calculate beta value of stock

def calculate_beta(ticker):

    #get data for ticker and SPY whih serves as the market index
    symbols = [ticker,'SPY']
    data = yf.download(symbols, start = start_date,end = end_date)['Adj Close'
    price_change = data.pct_change()
    price_change.drop(price_change.index[0],inplace=True)

    #reshape for linear regression
    X = np.array(price_change[ticker]).reshape((-1,1))
    y = np.array(price_change['SPY'])

    #create splits
    X_train,X_test,y_train,y_test = train_test_split(X,y)
    lr = LinearRegression()
    lr.fit(X_train,y_train)

    #predictions
    y_preds = lr.predict(X_test)

    #plot
    fig,ax = plt.subplots(figsize=(8,8));
    ax.plot(X_test,y_preds,linestyle=':',color='orange')
    ax.scatter(y_test,y_preds,alpha=0.5)
    ax.set_title(f' Beta value = {lr.coef_}')
    ax.set_xlabel('Market Index:SPY')
    ax.set_ylabel(f'{ticker} value')
```

3.4 P/E ratio

Price-to-Earnings(P/E) ratio is a metric that compares a company's share price to it's earnings per share. It helps an investor determine whether a stock is undervalued or overvalued. Hence, if a stock is overvalued, then the investor is paying more for the stock and betting on future growth and vice-versa.

In [5]:
```
#function to get historical PE ratios

def get_pe_ratio(ticker):

    #get financial statement of ticker using yahoofinancials
    financials = YahooFinancials(ticker)
    statement = financials.get_financial_stmts('annual', 'income', reformat=Tr

    #create a dict of the income statement alone
    dicts ={}
    for i in statement['incomeStatementHistory'][ticker]:
        dicts.update(i)

    #create a dataframe for easy use
    df = pd.DataFrame(dicts)
    df = df.T
    df['dilutedAverageShares'].fillna(df['dilutedAverageShares'].median(),inpl

    #calculate pe ratio
    eps = df['netIncomeContinuousOperations']/df['dilutedAverageShares'] #calc
    eps_df = pd.DataFrame(eps,index=df.index,columns=['EPS'])
    eps_df['PE ratio'] = financials.get_current_price()/eps_df['EPS']#get curr

    #plot result
    fig,ax=plt.subplots(figsize=(8,8));
    ax.plot(eps_df['PE ratio'],marker = '*', markerfacecolor = 'black')
    ax.set_title(f'Historical PE ratio: {ticker}')
    ax.set_ylabel('PE ratio')
```

3.5 Dividend History

A Dividend is the distirbution's of the company's profit to it's shareholders. Not every company pays dividends. Companies can also choose to re-invest their profits for future growth than reward shareholders. For an investor, investing in a company that pays dividends is an easy way to earn extra income on top of their initial investment.

In [6]:
```
#function to get historical dividend data

def get_dividend(ticker):

    #get dividend data
    financials = YahooFinancials(ticker)
    div = financials.get_daily_dividend_data(start_date=start_date,end_date='2(

    #check if the company pays dividends
    if div[ticker] == None:
        fig,ax = plt.subplots(figsize=(5,5))
        ax.annotate(f'Sorry, {ticker} does not offer dividends',xy=(0.3,0.5),f(
        ax.axis('off')
        return fig,ax

    else:#if company does pay dividends,then convert to a df and create a plot
        df = pd.DataFrame(div[ticker])
        df.drop('date',axis=1,inplace=True)
        df.rename(columns={'formatted_date':'date'},inplace=True)
        df['date'] = pd.to_datetime(df['date'])
        df.set_index('date',inplace=True)
        fig,ax=plt.subplots(figsize=(8,8))
        ax.plot(df['amount']);
        ax.set_title(f'Dividend rate: {ticker}');
```

4 Summing up performance...

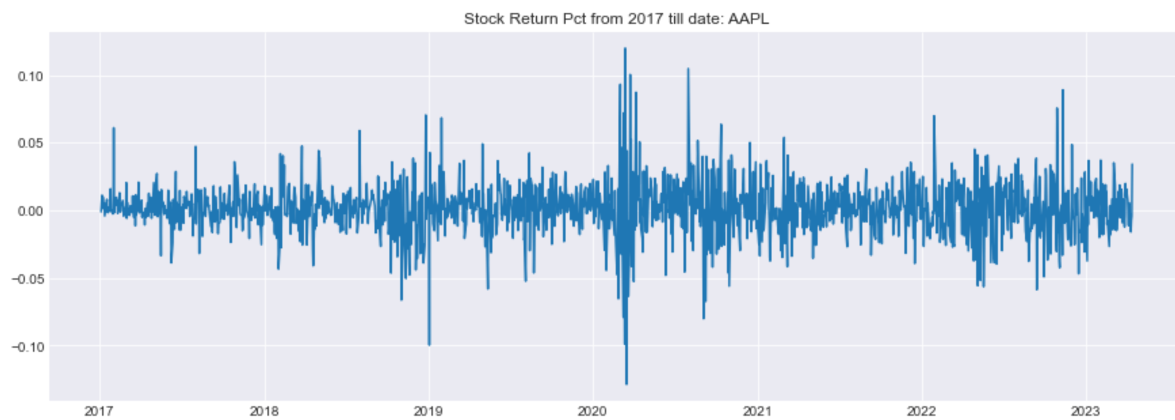
By combining all of the above into one function called `summary_info` , the investor can get a cohesive of his/her chosen stock. Below is an example of stock perfromance of **AAPL(tickr:'AAPL')**

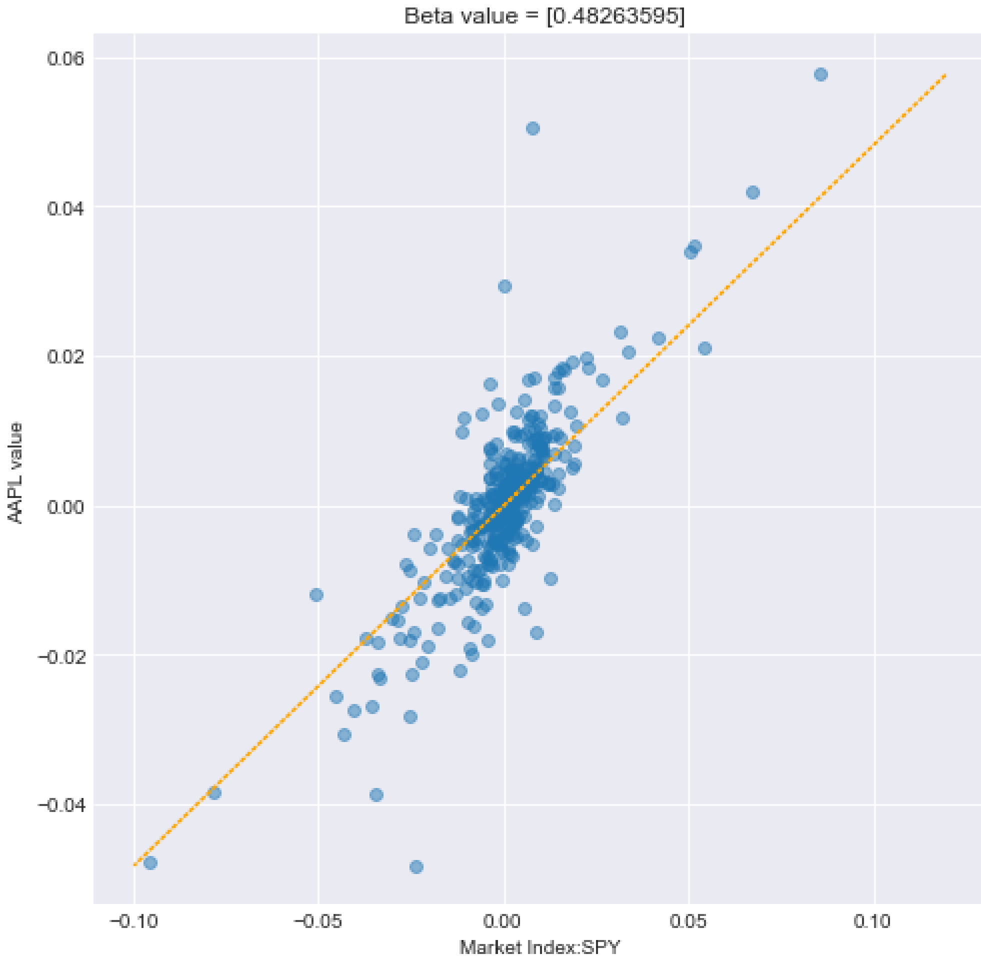
In [7]:
```
def summary_info(ticker):
    return stock_info(ticker),stock_return(ticker),calculate_beta(ticker),get_|
```

In [8]:
```
summary_info('AAPL')
```

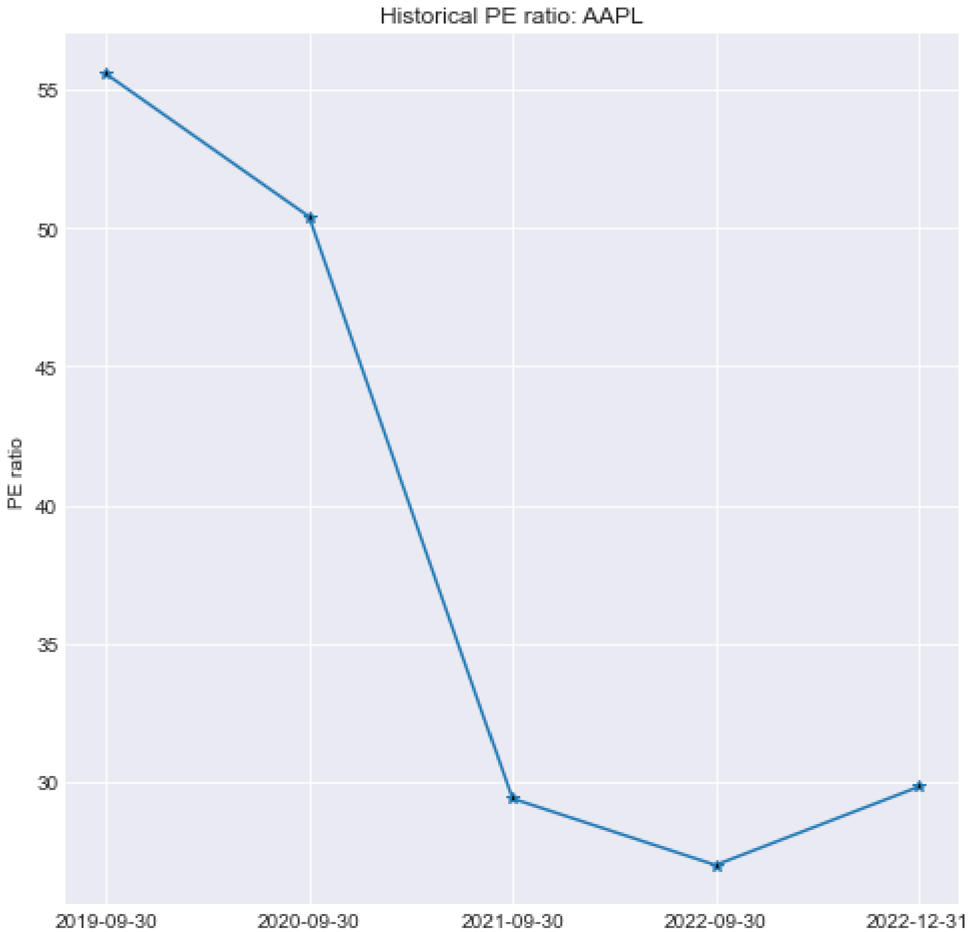
```
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  2 of 2 completed
```

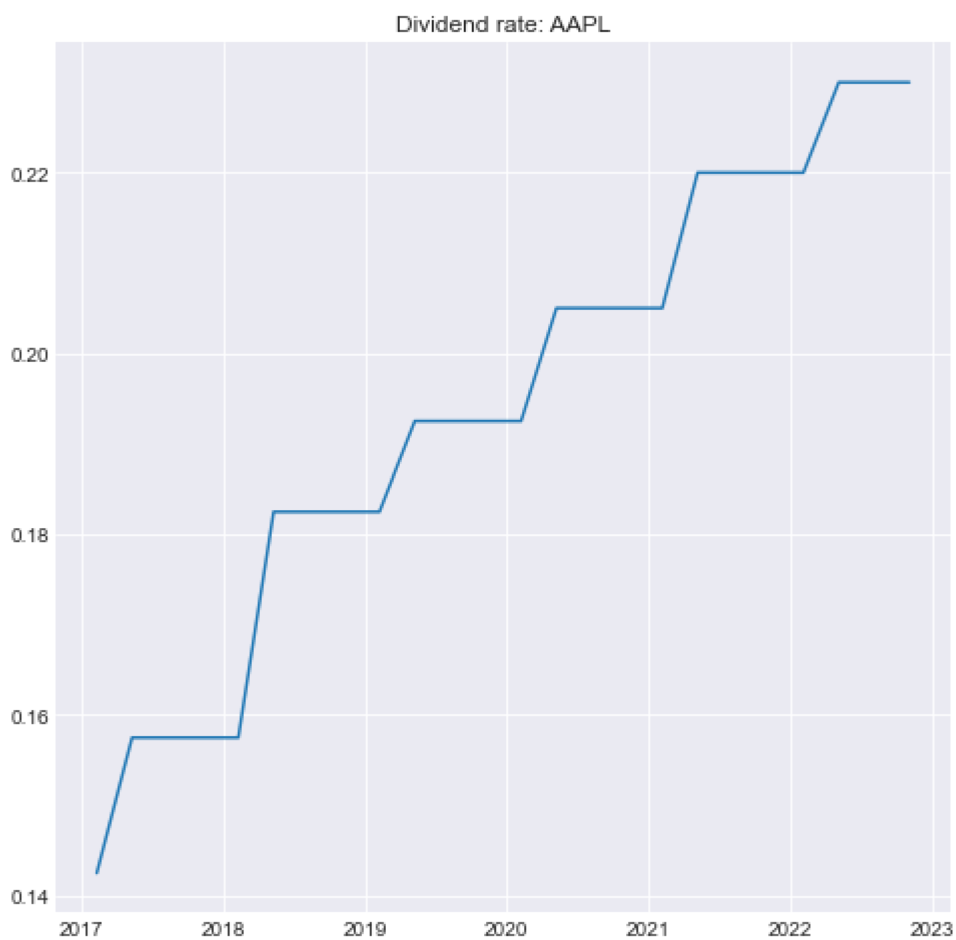
Out[8]:
```
(None, None, None, None, None)
```





Beta value = [0.48263595]
0.06
0.04
0.02
0.00
−0.02
−0.04
AAPL value
−0.10
−0.05
0.00
0.05
0.10
Market Index:SPY


Historical PE ratio: AAPL
PE ratio
55
50
45
40
35
30
2019-09-30
2020-09-30
2021-09-30
2022-09-30
2022-12-31


Dividend rate: AAPL
0.22
0.20
0.18
0.16
0.14
2017
2018
2019
2020
2021
2022
2023

1. Stock Performance: Stock is generally on an upward trend.
2. Beta Value : Less than 1.0 signifying that it is low-risk.
3. P/E ration : Around 30 highlighting that it is over valued. Perhaps it would be better to wait for the p/e to come down.
4. Dividend: Like the stock price, is on an upward trend. A nice boost to the shareholders for investing in the company.

5 Time Series Modelling

Again, we will use data of **Apple** as an example.

5.1 Stationarity Check

Time series models are usually built on the premise that models are stationary i.e there are patters to the data and by analyzing these patterns, future performance can be predicted with a degree of certainity. However, this rarely happens in real life. There is always some trend or seasonality or a combination of both in the data.Hence the first step is to check for stationarity.

The function below plots rolling-statistics and the ouptut of the Dickey-Fuller test

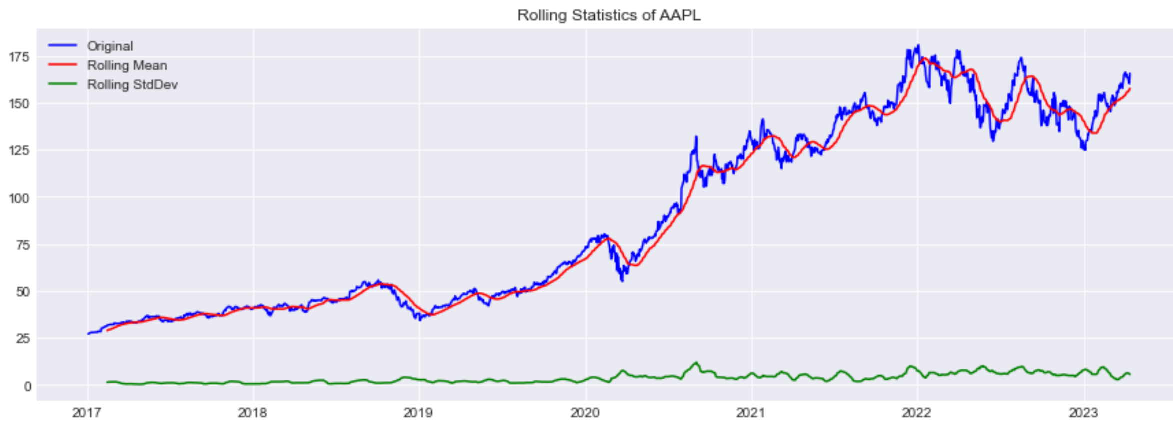
In [9]:
```
def plot_trends(ticker):
    df = yf.download(ticker,start=start_date,end=end_date)
    close = df.loc[:,['Adj Close']]

#compute rolling mean and std to see if they are constant
    roll_mean = close.rolling(window=30,center=False).mean()
    roll_std = close.rolling(window=30,center=False).std()

#plot the data
    fig,ax=plt.subplots(figsize=(15,5))
    ax.plot(close,color='blue',label='Original')
    ax.plot(roll_mean,color='red',label='Rolling Mean')
    ax.plot(roll_std,color='green',label='Rolling StdDev')
    ax.legend(loc='best')
    ax.set_title(f'Rolling Statistics of {ticker}');

#dickey Fuller Test
    dftest = adfuller(close['Adj Close'])
    dfoutput = pd.DataFrame(dftest[0:4], index=['Test Statistic', 'p-value', '

    return dfoutput
```

In [10]:
```
plot_trends('AAPL')
```

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

Out[10]:

	Values
Test Statistic	-0.552768
p-value	0.881301
#Lags Used	18.000000
Number of Observations Used	1561.000000



By computing the rolling mean we can see that there is an upward trend in the data. Since, the rolling stddev is fairly straight, we can conclude that there is not much seasonality in the data. Also shown is the results of the **Dickey-Fuller** test, a statistical method to check for stationarity. The large p-value points to a non-stationary dataset

5.2 Convert non-stationary to stationary

In [11]:

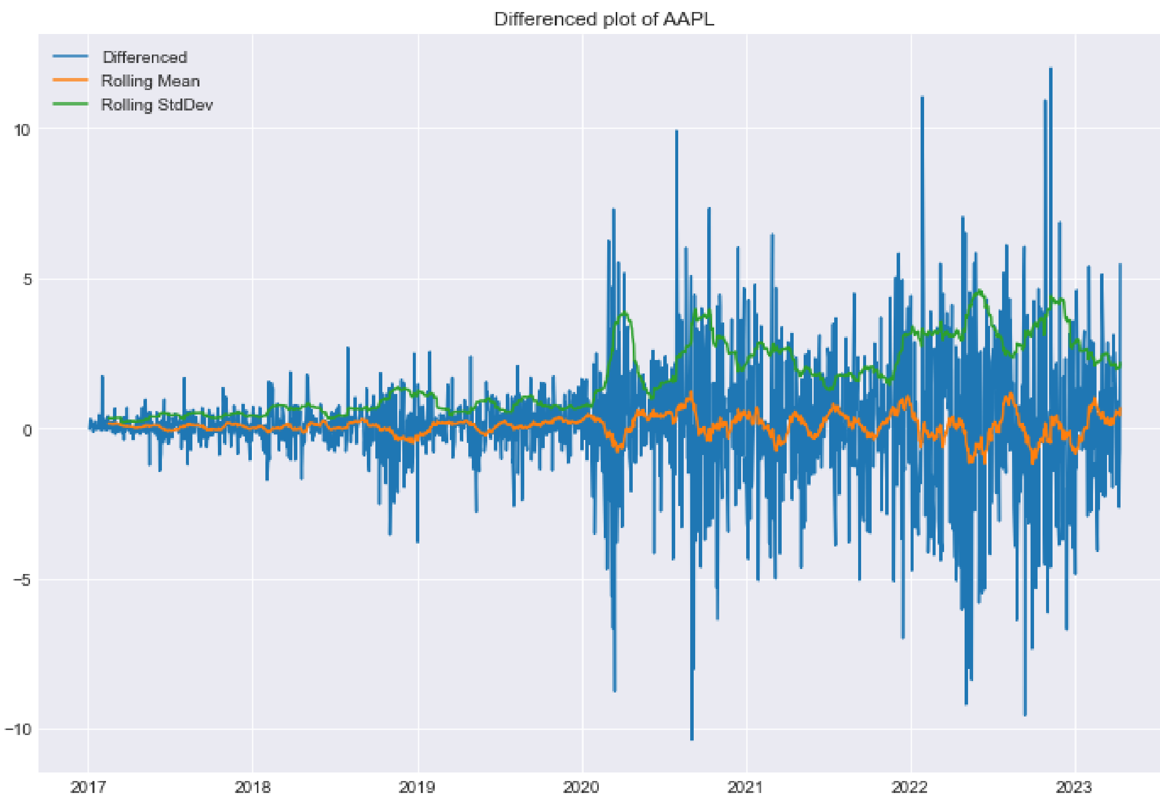
```
def differencing(ticker):
    df = yf.download(ticker,start=start_date,end=end_date)
    df_return = df[['Adj Close']]
    df_diff =df_return.diff(periods=1)
    df_diff.dropna(inplace=True)

    #plot the results
    fig,ax = plt.subplots(figsize = (12,8));
    ax.plot(df_diff, label = 'Differenced');
    ax.plot(df_diff.rolling(30).mean(),label = 'Rolling Mean')
    ax.plot(df_diff.rolling(30).std(),label = 'Rolling StdDev')
    ax.set_title(f'Differenced plot of {ticker}')
    ax.legend(loc=2);

    #dickey fuller test
    dftest = adfuller(df_diff['Adj Close'])
    dfoutput = pd.DataFrame(dftest[0:4], index=['Test Statistic', 'p-value', '

    return dfoutput
```

In [12]:
```
differencing('AAPL')
```

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

Out[12]:

	Values
Test Statistic	-8.753872e+00
p-value	2.789202e-14
#Lags Used	1.700000e+01
Number of Observations Used	1.561000e+03



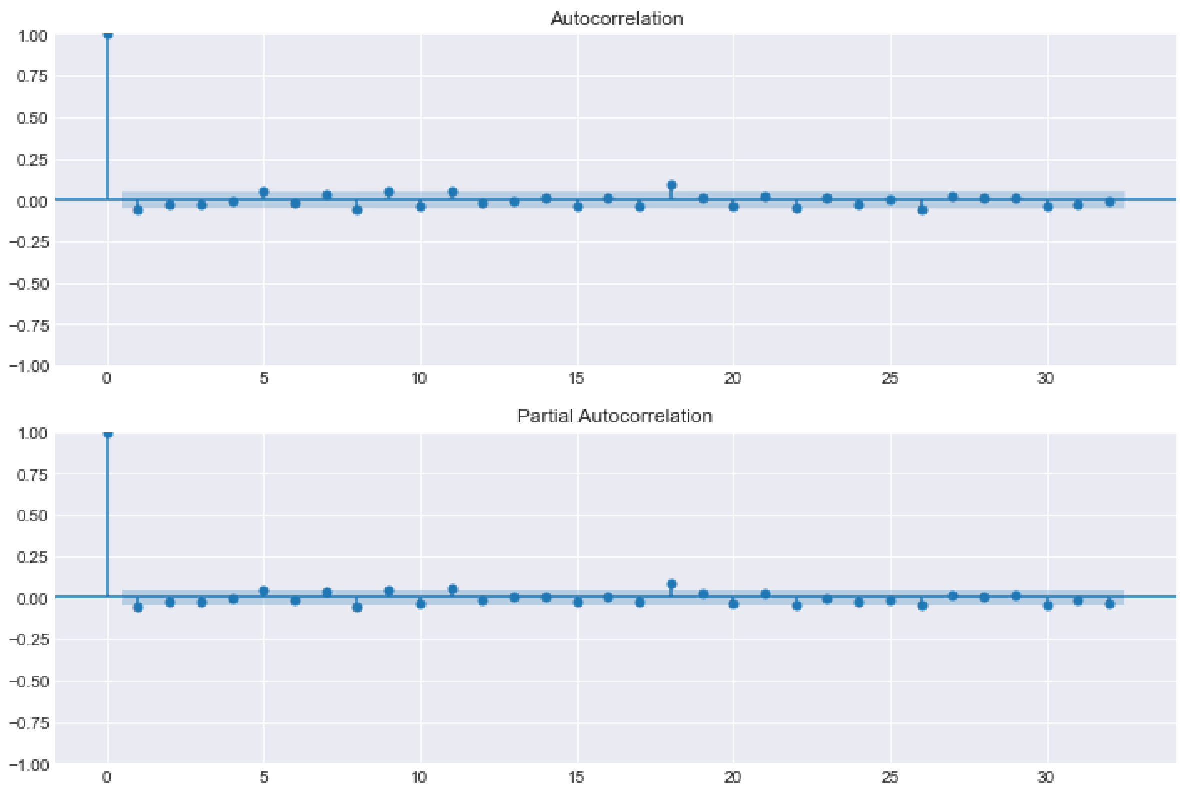
We can from the above plot that the mean though not perfectly flat is fairly linear signifying that we have removed the trend. The miniscule p-value from the Dicley-Fuller test also points towards a stationary dataset

5.3 ACF and PACF

Since the ARIMA and SARIMA modelss are linear regression models, we need to decide on how many regression terms we will use for the model. The Auto-Correlation and Partial Auto Correlation plot(for the AR and MA models respectively) will show the number of lag terms that have the most effect on future price.

```
In [13]: #get stock data
df = yf.download('AAPL',start=start_date,end=end_date)
df_close = df[['Adj Close']]
df_diff =df_close.diff(periods=1)
df_diff.dropna(inplace=True)

#plot ACF and PACF values
fig,(ax1,ax2)=plt.subplots(nrows=2,figsize=(12,8))
acf = plot_acf(df_diff,ax=ax1)
pacf = plot_pacf(df_diff,ax=ax2)
```

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



From both the plots, we can see that the 1st lag term i.e the previous day's stock price, will have the most effect on the next day's price. There are other terms that have an effect as well, but for the sake of simplicity we will use only one term for the model.

5.4 ARIMA model

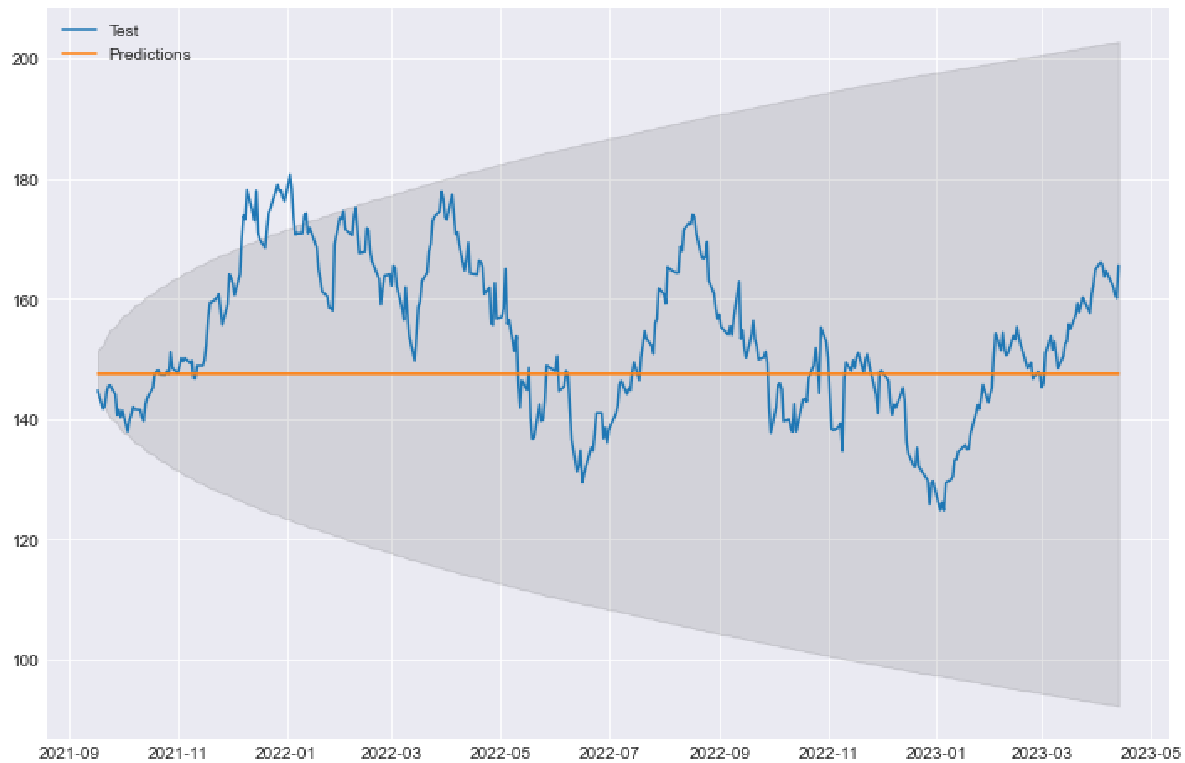
We will first build and integrated AR and MA model known as ARIMA.

```
In [14]: #defining train and test sets
cutoff= int(df_close.shape[0]*0.75)
train = df_close[:cutoff]
test = df_close[cutoff:]
```

Parameters for the model: **(p,d,q)=(1,1,1)** for the model based on the plots

1. p = number of terms for the AR model
2. d = order of differencing
3. q = numer of the terms for the MA model

In [15]:
```
# (1,1,1) ARIMA(p,d,q) based on ACF and PACF plots
#instantiate
model = ARIMA(train,order=(1,1,1))

#fit
model_fit = model.fit()
# print(model_fit.summary())

#getting predictions using get_prediction method
arima_predictions = model_fit.get_prediction(start = len(train)+1 , end = len(
#predicted_mean gives lists the values
arima_pred_price=arima_predictions.predicted_mean
#converting into a df
arima_pred_price_df = pd.DataFrame(data=arima_pred_price)
#seetting the index to the test dates
arima_pred_price_df.index= test.index

# arima_pred_price_df.head()

#confidence intervals of predictions
arima_conf_int = arima_predictions.conf_int()
arima_conf_int.set_index(test.index,inplace=True)

# arima_conf_int.head()

#plotting predictions with confidence intervals
fig,ax =plt.subplots(figsize=(12,8))
# ax.plot(train,label='Train');
ax.plot(test,label='Test');
ax.plot(arima_pred_price_df,label='Predictions');
# ax.plot(arima_conf_int,label='Confidence Intervals')
ax.fill_between(arima_conf_int.index,arima_conf_int.iloc[:,0],arima_conf_int.i
ax.legend(loc=2);
```



In [16]:
```
#RMSE and AIC
error_arima = round(np.sqrt(mean_squared_error(test,arima_pred_price_df)),2)
aic_arima = round(model_fit.aic,2)
print(f'AIC score of the ARIMA model is {aic_arima}')
print(f'RMSE of the model is ${error_arima}')
```

```
AIC score of the ARIMA model is 4372.67
RMSE of the model is $13.89
```

As we can tell, the model does not perform very well when compared to the test values

5.5 SARIMA model

Next, we will build a SARIMA model. Like the ARIMA models, SARIMA model also depends on past values but has an extra seasonality component to take into account any seasonality patterns

the the p,d,q values from the ARIMA models as a guide, we can run different combinations to check for the most optimal paramenters. The model that outputs the lowest AIC score , will be used as our model

In [17]:
```
#defining a range for the p,d,q values
p=d=q=range(0,2)
pdq = list(itertools.product(p,d,q))
pdqs = [(x[0], x[1], x[2], 12) for x in list(itertools.product(p, d, q))]
```

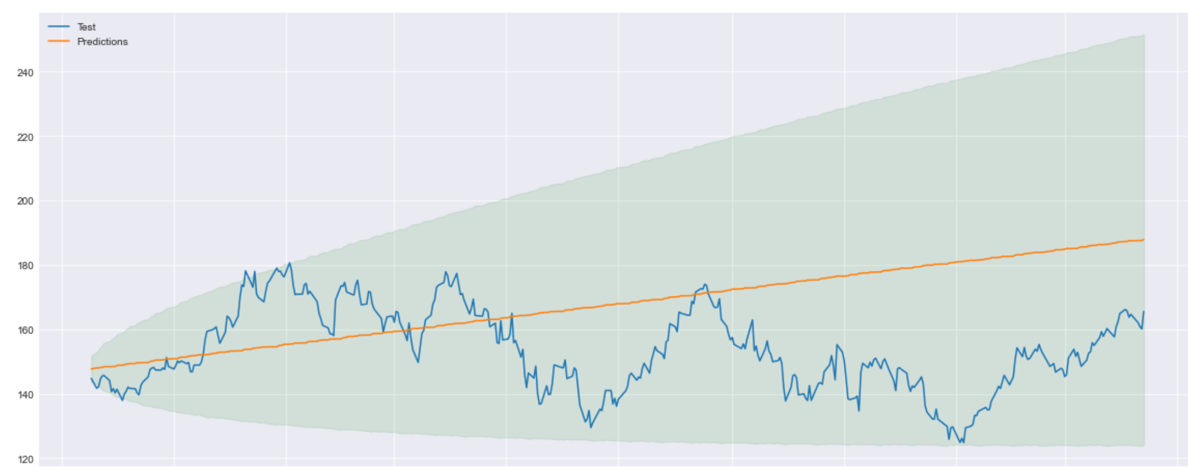
In [18]:
```
# Run a grid with pdq and seasonal pdq parameters calculated above and get the
ans = []
for comb in pdq:
    for combs in pdqs:
        try:
            mod = SARIMAX(train,
                          order=comb,
                          seasonal_order=combs,
                          enforce_stationarity=False,
                          enforce_invertibility=False)

            output = mod.fit()
            ans.append([comb, combs, output.aic])
            print('SARIMA {} x {}: AIC Calculated={}'.format(comb, combs, outpu
        except:
            continue
```

```
SARIMA (0, 0, 0) x (0, 0, 0, 12): AIC Calculated=13648.525125476559
SARIMA (0, 0, 0) x (0, 0, 1, 12): AIC Calculated=12015.33071590064
SARIMA (0, 0, 0) x (0, 1, 0, 12): AIC Calculated=7098.910657904551
SARIMA (0, 0, 0) x (0, 1, 1, 12): AIC Calculated=7019.460038338637
SARIMA (0, 0, 0) x (1, 0, 0, 12): AIC Calculated=7022.021051129367
SARIMA (0, 0, 0) x (1, 0, 1, 12): AIC Calculated=7016.256962305424
SARIMA (0, 0, 0) x (1, 1, 0, 12): AIC Calculated=7026.954949367409
SARIMA (0, 0, 0) x (1, 1, 1, 12): AIC Calculated=7020.800892401479
SARIMA (0, 0, 1) x (0, 0, 0, 12): AIC Calculated=12043.743655288119
SARIMA (0, 0, 1) x (0, 0, 1, 12): AIC Calculated=10460.950058526576
SARIMA (0, 0, 1) x (0, 1, 0, 12): AIC Calculated=6147.351932686597
SARIMA (0, 0, 1) x (0, 1, 1, 12): AIC Calculated=6096.417724856125
SARIMA (0, 0, 1) x (1, 0, 0, 12): AIC Calculated=6103.322919402553
SARIMA (0, 0, 1) x (1, 0, 1, 12): AIC Calculated=6085.691020735404
SARIMA (0, 0, 1) x (1, 1, 0, 12): AIC Calculated=6105.8059719460525
SARIMA (0, 0, 1) x (1, 1, 1, 12): AIC Calculated=6097.755329425278
SARIMA (0, 1, 0) x (0, 0, 0, 12): AIC Calculated=4381.354416950779
SARIMA (0, 1, 0) x (0, 0, 1, 12): AIC Calculated=4350.663293519591
SARIMA (0, 1, 0) x (0, 1, 0, 12): AIC Calculated=5142.041720415686
SARIMA (0, 1, 0) x (0, 1, 1, 12): AIC Calculated=4350.1872205295695
SARIMA (0, 1, 0) x (1, 0, 0, 12): AIC Calculated=4353.389551298473
SARIMA (0, 1, 0) x (1, 0, 1, 12): AIC Calculated=4351.7955096042215
SARIMA (0, 1, 0) x (1, 1, 0, 12): AIC Calculated=4837.710684455359
SARIMA (0, 1, 0) x (1, 1, 1, 12): AIC Calculated=4352.187254287945
SARIMA (0, 1, 1) x (0, 0, 0, 12): AIC Calculated=4367.363818392725
SARIMA (0, 1, 1) x (0, 0, 1, 12): AIC Calculated=4336.866257496928
SARIMA (0, 1, 1) x (0, 1, 0, 12): AIC Calculated=5132.111733334201
SARIMA (0, 1, 1) x (0, 1, 1, 12): AIC Calculated=4335.749682218427
SARIMA (0, 1, 1) x (1, 0, 0, 12): AIC Calculated=4342.279188460179
SARIMA (0, 1, 1) x (1, 0, 1, 12): AIC Calculated=4338.4980184454
SARIMA (0, 1, 1) x (1, 1, 0, 12): AIC Calculated=4828.648076688056
SARIMA (0, 1, 1) x (1, 1, 1, 12): AIC Calculated=4337.7496716150235
SARIMA (1, 0, 0) x (0, 0, 0, 12): AIC Calculated=4381.050639373734
SARIMA (1, 0, 0) x (0, 0, 1, 12): AIC Calculated=4351.753489454006
SARIMA (1, 0, 0) x (0, 1, 0, 12): AIC Calculated=5091.083580405054
SARIMA (1, 0, 0) x (0, 1, 1, 12): AIC Calculated=4355.065007549292
SARIMA (1, 0, 0) x (1, 0, 0, 12): AIC Calculated=4350.528223225626
SARIMA (1, 0, 0) x (1, 0, 1, 12): AIC Calculated=4352.188544313671
SARIMA (1, 0, 0) x (1, 1, 0, 12): AIC Calculated=4806.072095440912
SARIMA (1, 0, 0) x (1, 1, 1, 12): AIC Calculated=4357.065050126487
SARIMA (1, 0, 1) x (0, 0, 0, 12): AIC Calculated=4365.313319138035
SARIMA (1, 0, 1) x (0, 0, 1, 12): AIC Calculated=4336.9757428522735
SARIMA (1, 0, 1) x (0, 1, 0, 12): AIC Calculated=5087.975082092069
SARIMA (1, 0, 1) x (0, 1, 1, 12): AIC Calculated=4340.338975480652
SARIMA (1, 0, 1) x (1, 0, 0, 12): AIC Calculated=4337.602788123384
SARIMA (1, 0, 1) x (1, 0, 1, 12): AIC Calculated=4336.851003798329
SARIMA (1, 0, 1) x (1, 1, 0, 12): AIC Calculated=4802.378972247991
SARIMA (1, 0, 1) x (1, 1, 1, 12): AIC Calculated=4342.338968473674
SARIMA (1, 1, 0) x (0, 0, 0, 12): AIC Calculated=4369.267839177572
SARIMA (1, 1, 0) x (0, 0, 1, 12): AIC Calculated=4338.786996252471
SARIMA (1, 1, 0) x (0, 1, 0, 12): AIC Calculated=5135.300177152003
SARIMA (1, 1, 0) x (0, 1, 1, 12): AIC Calculated=4338.197679060014
SARIMA (1, 1, 0) x (1, 0, 0, 12): AIC Calculated=4338.789144566135
SARIMA (1, 1, 0) x (1, 0, 1, 12): AIC Calculated=4340.68711106191
SARIMA (1, 1, 0) x (1, 1, 0, 12): AIC Calculated=4824.37309083344
SARIMA (1, 1, 0) x (1, 1, 1, 12): AIC Calculated=4340.197656507765
SARIMA (1, 1, 1) x (0, 0, 0, 12): AIC Calculated=4367.276012773678
```

```
SARIMA (1, 1, 1) x (0, 0, 1, 12): AIC Calculated=4336.803713029531
SARIMA (1, 1, 1) x (0, 1, 0, 12): AIC Calculated=5132.722949360944
SARIMA (1, 1, 1) x (0, 1, 1, 12): AIC Calculated=4336.091413748307
SARIMA (1, 1, 1) x (1, 0, 0, 12): AIC Calculated=4339.5164008467
SARIMA (1, 1, 1) x (1, 0, 1, 12): AIC Calculated=4338.714445690122
SARIMA (1, 1, 1) x (1, 1, 0, 12): AIC Calculated=4821.779956576529
SARIMA (1, 1, 1) x (1, 1, 1, 12): AIC Calculated=4338.104971411733
```

In [19]:

```
# Plug the optimal parameter values into a new SARIMAX model
sarimax = SARIMAX(train,
                  order=(1,1,1),
                  seasonal_order=(1 ,1, 1, 12),
                  enforce_stationarity=False,
                  enforce_invertibility=False)

# Fit the model and print results
output = sarimax.fit()

#get predictions
sarimax_predictions = output.get_prediction(start=len(train)+1,end=len(df_clos
sarimax_price=sarimax_predictions.predicted_mean
sarimax_predictions_df = pd.DataFrame(data=sarimax_price)
sarimax_predictions_df.index= test.index

#get confidence intervals
sarimax_conf_int = sarimax_predictions.conf_int()
sarimax_conf_int.set_index(test.index,inplace=True)

#plot results
fig,ax =plt.subplots(figsize=(20,8))
# ax.plot(train,label='Train');
ax.plot(test,label='Test');
ax.plot(sarimax_predictions_df,label='Predictions');
# ax.plot(sarimax_conf_int,label='Confidence Intervals')
ax.fill_between(sarimax_conf_int.index,sarimax_conf_int.iloc[:,0],sarimax_conf_
ax.legend(loc=2);
```



In [20]:
```
#RMSE
error_sarima = round(np.sqrt(mean_squared_error(test,sarimax_predictions_df)),
print(f'RMSE of the model is ${error_sarima}')
```

```
RMSE of the model is $24.37
```

Compared to the ARIMA model, the SARIMA model performance is much worse.

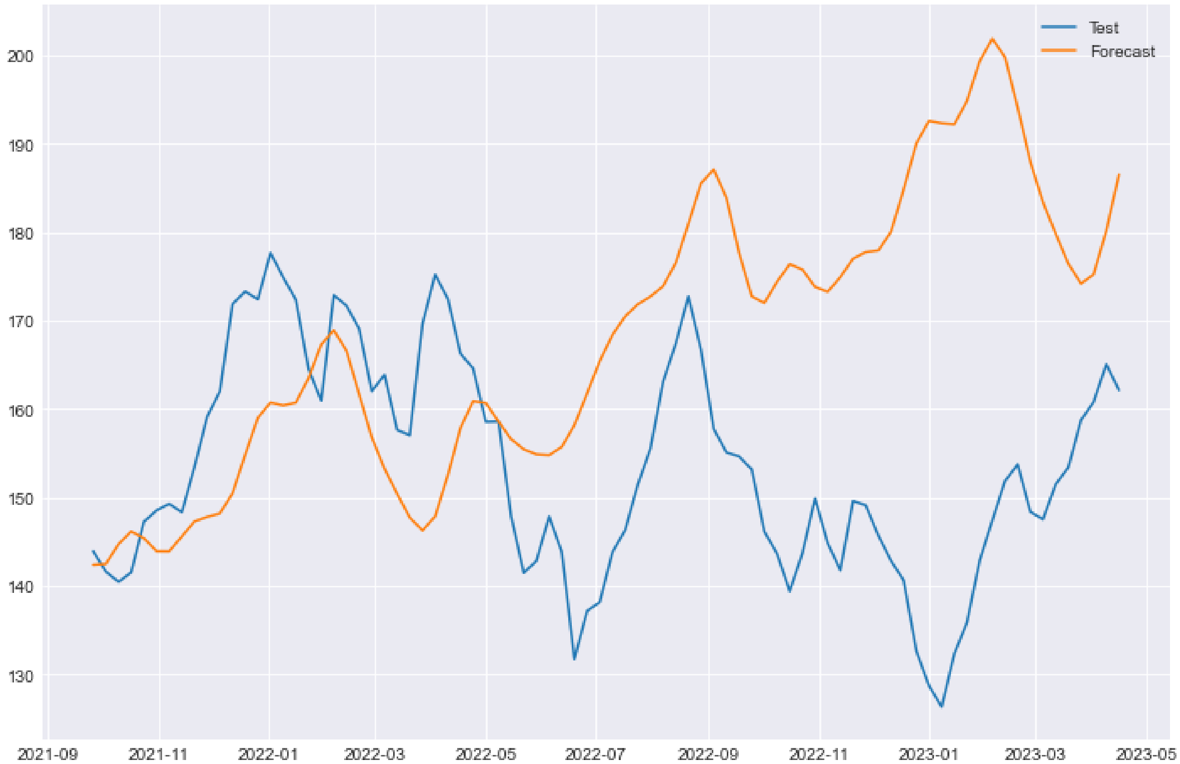
5.6 Facebook Prophet

The Prophet model is an additive model for time series predicting that was open sourced by Meta. in 2017.According to the official documentation, it works best with time series that have strong seasonal effects and several seasons of historical data Prophet is robust to missing data and shifts in the trend and typically handles outliers well.

First, we will use the default parameters and the undifferenced data to to build and evaluate the model:

In [21]:
```
#get stock data
df = yf.download('AAPL',start=start_date,end=end_date)
df_close = df[['Adj Close']]

#resmapling by weekly
df_weekly = df_close.resample('W').mean()

#setting up df to be able to run Prophet

#reset index
df_weekly.reset_index(inplace=True)

#rename columns per prophet conventions
df_weekly.rename(columns={'Adj Close': 'y',
                         'Date':'ds'},inplace=True)

#set the date as the index
# df_close.set_index('ds',inplace=True)
# df_weekly.head()

#defining train and test sets
cutoff= int(df_weekly.shape[0]*0.75)
train = df_weekly[:cutoff]
test = df_weekly[cutoff:]

#instantiate
m = Prophet(seasonality_mode='multiplicative',
            weekly_seasonality=True,
            daily_seasonality = True,
            yearly_seasonality = True,
            interval_width=0.90,
           )

#fit
m.fit(train)

#forecasts - creating future dates using in built make_future_dataframe method
future = m.make_future_dataframe(periods=len(test),freq='W',include_history=Fa

#predicting yhat
forecast = m.predict(future)

#creating a df of predicted values
forecast_values = forecast[['ds', 'yhat', 'yhat_lower', 'yhat_upper']]

#set the index for the plots
forecast_values.set_index('ds',inplace=True)
test.set_index('ds',inplace=True)

#plotting results
fig,ax=plt.subplots(figsize=(12,8))
# ax.plot(train,label='Train')
ax.plot(test,label='Test')
# ax.plot(df_weekly,label='Current Price')
ax.plot(forecast_values['yhat'],label='Forecast')
# ax.fill_between(test.index,forecast_values['yhat_lower'],forecast_values['yh
ax.legend();
```

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



In [22]:
```
#RMSE
error_fb = round(np.sqrt(mean_squared_error(test,forecast_values['yhat'])),2)
print(f'RMSE of the model is ${error_fb}')
```

```
RMSE of the model is $26.78
```

We can try to optimize the model by running a loop for the seasonality and changepoint values. These are deciding how much to penalize seasonality and changepoints changes in the data i.e if the values are small for the seasonality changes, then the effect of seasonal changes in the data is dampened and vice versa.

In [23]:

```
#run a loop for different regularization values

seasonality_scale = [0.1, 0.2, 0.3, 0.4, 0.5]
changepoint_scale = [0.1, 0.2, 0.3, 0.4, 0.5]
errors = []
new_error_fb = None

for season in seasonality_scale:
    for changepoint in changepoint_scale:
        #instantiate
        m = Prophet(seasonality_mode='multiplicative',
                    weekly_seasonality=True,
                    daily_seasonality = True,
                    yearly_seasonality = True,
                    interval_width=0.90,
                    seasonality_prior_scale=season,
                    changepoint_prior_scale=changepoint
                   )

        #fit
        m.fit(train)

        #forecasts - creating future dates using in built make_future_datafram
        future = m.make_future_dataframe(periods=len(test),freq='W',include_hi

        #predicting yhat
        forecast = m.predict(future)

        #getting only yhat values
        forecast_values = forecast[['ds', 'yhat']]

        #setting the index
        forecast_values.set_index('ds',inplace=True)

        #rmse values
        rmse = round(np.sqrt(mean_squared_error(test,forecast_values['yhat']))
        errors.append(rmse)
        new_error_fb = min(errors)
        print(f'seasonality_scale:{season}, changepoint_scale:{changepoint}, r

print('----------------------------------------------')
print(f'Smallest RMSE after looping is ${new_error_fb}')
print(f'Original RMSE is ${error_fb}')
```

```
seasonality_scale:0.1, changepoint_scale:0.1, rmse:29.917
seasonality_scale:0.1, changepoint_scale:0.2, rmse:28.679
seasonality_scale:0.1, changepoint_scale:0.3, rmse:27.51
seasonality_scale:0.1, changepoint_scale:0.4, rmse:26.744
seasonality_scale:0.1, changepoint_scale:0.5, rmse:25.965
seasonality_scale:0.2, changepoint_scale:0.1, rmse:29.176
seasonality_scale:0.2, changepoint_scale:0.2, rmse:26.93
seasonality_scale:0.2, changepoint_scale:0.3, rmse:26.098
seasonality_scale:0.2, changepoint_scale:0.4, rmse:25.082
seasonality_scale:0.2, changepoint_scale:0.5, rmse:24.607
seasonality_scale:0.3, changepoint_scale:0.1, rmse:28.379
seasonality_scale:0.3, changepoint_scale:0.2, rmse:26.472
seasonality_scale:0.3, changepoint_scale:0.3, rmse:25.031
seasonality_scale:0.3, changepoint_scale:0.4, rmse:24.704
seasonality_scale:0.3, changepoint_scale:0.5, rmse:25.225
seasonality_scale:0.4, changepoint_scale:0.1, rmse:27.164
seasonality_scale:0.4, changepoint_scale:0.2, rmse:25.628
seasonality_scale:0.4, changepoint_scale:0.3, rmse:24.497
seasonality_scale:0.4, changepoint_scale:0.4, rmse:24.625
seasonality_scale:0.4, changepoint_scale:0.5, rmse:25.015
seasonality_scale:0.5, changepoint_scale:0.1, rmse:26.387
seasonality_scale:0.5, changepoint_scale:0.2, rmse:25.229
seasonality_scale:0.5, changepoint_scale:0.3, rmse:24.203
seasonality_scale:0.5, changepoint_scale:0.4, rmse:24.02
seasonality_scale:0.5, changepoint_scale:0.5, rmse:23.963
----------------------------------------------
Smallest RMSE after looping is $23.963
Original RMSE is $26.78
```

We can see a decent jump in performance. Let's see if we can improve further

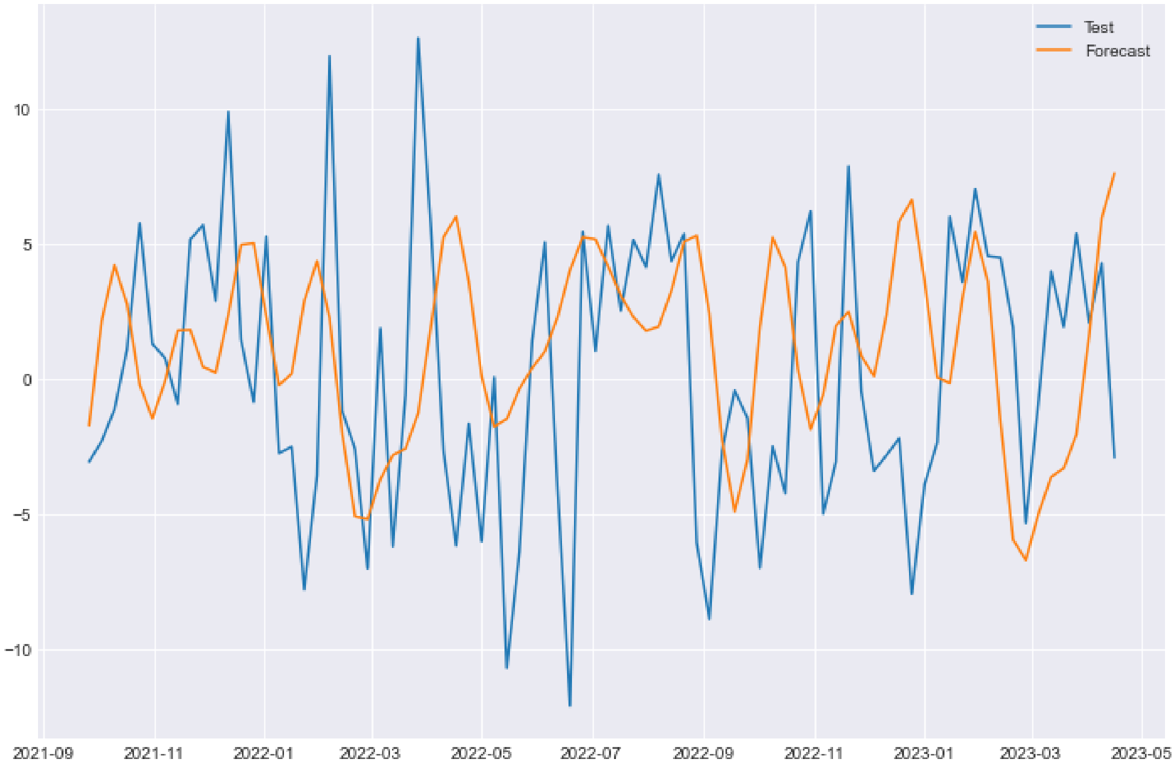
5.6.1 Prophet with differencing - comparing with test

Like the other models, we can try and run the prophet model on stationary data to see if there is an improvement. We accomplish this by diffenreceing the data by an order of 1.

In [24]:

```
#get stock data
df = yf.download('AAPL',start=start_date,end=end_date)
df_close = df[['Adj Close']]

#resmapling by weekly
df_weekly = df_close.resample('W').mean()

df_weekly = df_weekly.diff(periods=1)
df_weekly.dropna(inplace=True)

#setting up df to be able to run Prophet
#reset index
df_weekly.reset_index(inplace=True)
#rename columns per prophet conventions
df_weekly.rename(columns={'Adj Close': 'y',
                         'Date':'ds'},inplace=True)

#defining train and test sets
cutoff= int(df_weekly.shape[0]*0.75)
train = df_weekly[:cutoff]
test = df_weekly[cutoff:]

#instantiate
m = Prophet(seasonality_mode='multiplicative',
            weekly_seasonality=True,
            daily_seasonality = True,
            yearly_seasonality = True,
            interval_width=0.90,
           )

#fit
m.fit(train)

#forecasts - creating future dates using in built make_future_dataframe method
future = m.make_future_dataframe(periods=len(test),freq='W',include_history=Fa

#predicting yhat
forecast = m.predict(future)

#creating a df of predicted values
forecast_values = forecast[['ds', 'yhat']]

#set the index for the plots
forecast_values.set_index('ds',inplace=True)
test.set_index('ds',inplace=True)

#plotting results
fig,ax=plt.subplots(figsize=(12,8))
# ax.plot(train,label='Train')
ax.plot(test,label='Test')
# ax.plot(df_weekly,label='Current Price')
ax.plot(forecast_values['yhat'],label='Forecast')
# ax.fill_between(test.index,forecast_values['yhat_lower'],forecast_values['yh
ax.legend();
```

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



In [25]:
```python
#RMSE
error_fb_diff = round(np.sqrt(mean_squared_error(test,forecast_values['yhat'])
print(f'RMSE of the model is ${error_fb_diff}')
```

```
RMSE of the model is $6.08
```

Here, we can see a vast improvement in model performance compared to before.

5.6.2 regularization on differenced data

We will same optimization loop as earlier for the differenced data to see if it impacts performance.

In [26]:

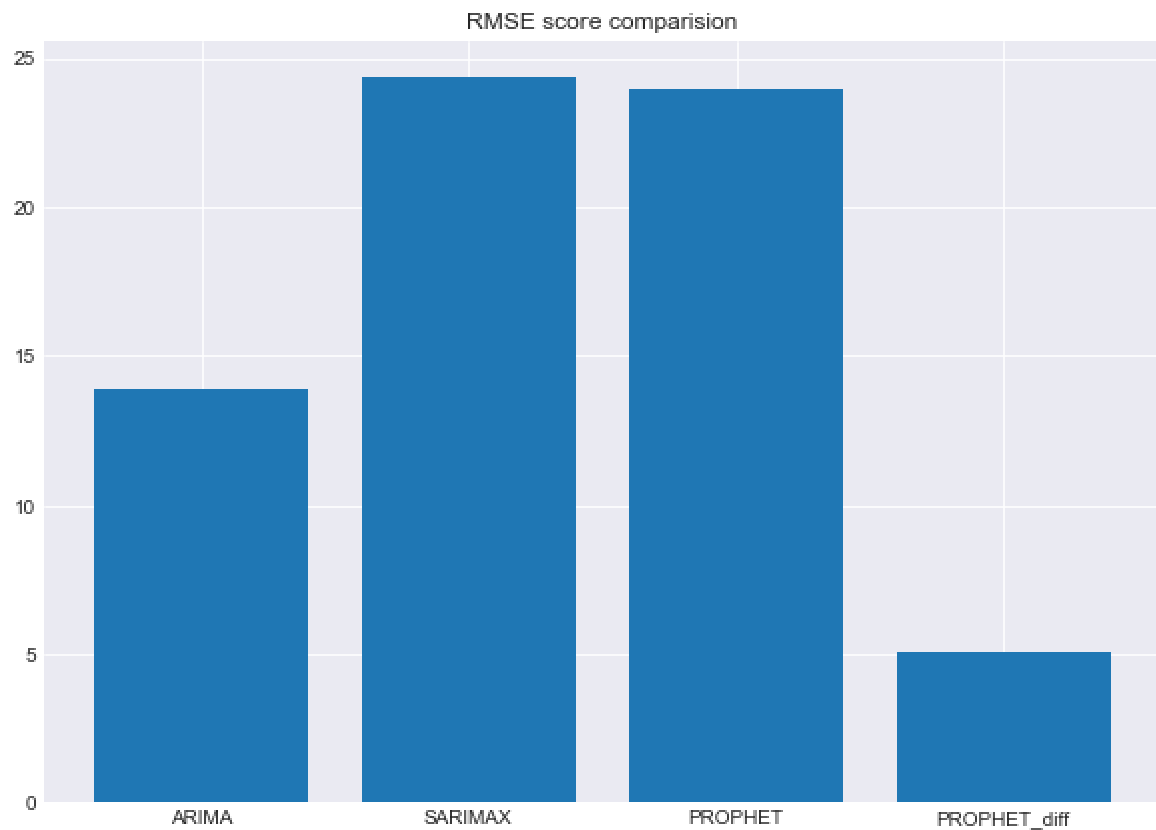
```
#run a loop for different regularization values
seasonality_scale = [0.1, 0.2, 0.3, 0.4, 0.5]
changepoint_scale = [0.1, 0.2, 0.3, 0.4, 0.5]
errors = []
new_error_fb_diff = None

for season in seasonality_scale:
    for changepoint in changepoint_scale:
        #instantiate
        m = Prophet(seasonality_mode='multiplicative',
                weekly_seasonality=True,
                daily_seasonality = True,
                yearly_seasonality = True,
                interval_width=0.90,
                seasonality_prior_scale=season,
                changepoint_prior_scale=changepoint
               )

        #fit
        m.fit(train)

        #forecasts - creating future dates using in built make_future_datafram
        future = m.make_future_dataframe(periods=len(test),freq='W',include_hi

        #predicting yhat
        forecast = m.predict(future)

        #getting only yhat values
        forecast_values = forecast[['ds', 'yhat']]

        #setting the index
        forecast_values.set_index('ds',inplace=True)

        #rmse values
        rmse = round(np.sqrt(mean_squared_error(test,forecast_values['yhat']))
        errors.append(rmse)
        new_error_fb_diff = min(errors)
        print(f'seasonality_scale:{season}, changepoint_scale:{changepoint}, r

print('----------------------------------------------')
print(f'Smallest RMSE after looping is {new_error_fb_diff}')
print(f'Original RMSE is {error_fb_diff}')
```

```
seasonality_scale:0.1, changepoint_scale:0.1, rmse:6.014
seasonality_scale:0.1, changepoint_scale:0.2, rmse:6.039
seasonality_scale:0.1, changepoint_scale:0.3, rmse:6.164
seasonality_scale:0.1, changepoint_scale:0.4, rmse:6.311
seasonality_scale:0.1, changepoint_scale:0.5, rmse:6.368
seasonality_scale:0.2, changepoint_scale:0.1, rmse:5.998
seasonality_scale:0.2, changepoint_scale:0.2, rmse:6.247
seasonality_scale:0.2, changepoint_scale:0.3, rmse:6.358
seasonality_scale:0.2, changepoint_scale:0.4, rmse:6.429
seasonality_scale:0.2, changepoint_scale:0.5, rmse:6.362
seasonality_scale:0.3, changepoint_scale:0.1, rmse:6.181
seasonality_scale:0.3, changepoint_scale:0.2, rmse:6.386
seasonality_scale:0.3, changepoint_scale:0.3, rmse:6.352
seasonality_scale:0.3, changepoint_scale:0.4, rmse:5.224
seasonality_scale:0.3, changepoint_scale:0.5, rmse:5.16
seasonality_scale:0.4, changepoint_scale:0.1, rmse:6.302
seasonality_scale:0.4, changepoint_scale:0.2, rmse:6.447
seasonality_scale:0.4, changepoint_scale:0.3, rmse:5.239
seasonality_scale:0.4, changepoint_scale:0.4, rmse:5.093
seasonality_scale:0.4, changepoint_scale:0.5, rmse:5.099
seasonality_scale:0.5, changepoint_scale:0.1, rmse:6.348
seasonality_scale:0.5, changepoint_scale:0.2, rmse:6.298
seasonality_scale:0.5, changepoint_scale:0.3, rmse:5.112
seasonality_scale:0.5, changepoint_scale:0.4, rmse:5.09
seasonality_scale:0.5, changepoint_scale:0.5, rmse:5.11
----------------------------------------------
Smallest RMSE after looping is 5.09
Original RMSE is 6.08
```

This time, the improvement is only slight.

5.7 Model performance comparisions

In [27]:
```
fig,ax =plt.subplots(figsize=(10,7))
ax.bar(x=['ARIMA', 'SARIMAX', 'PROPHET','PROPHET_diff'],height=[error_arima,er
ax.set_title('RMSE score comparision');
```



From the plot, we can see the prohet model on the differenced data performs best. Hence, we will use that for making predictions for our chose stock.

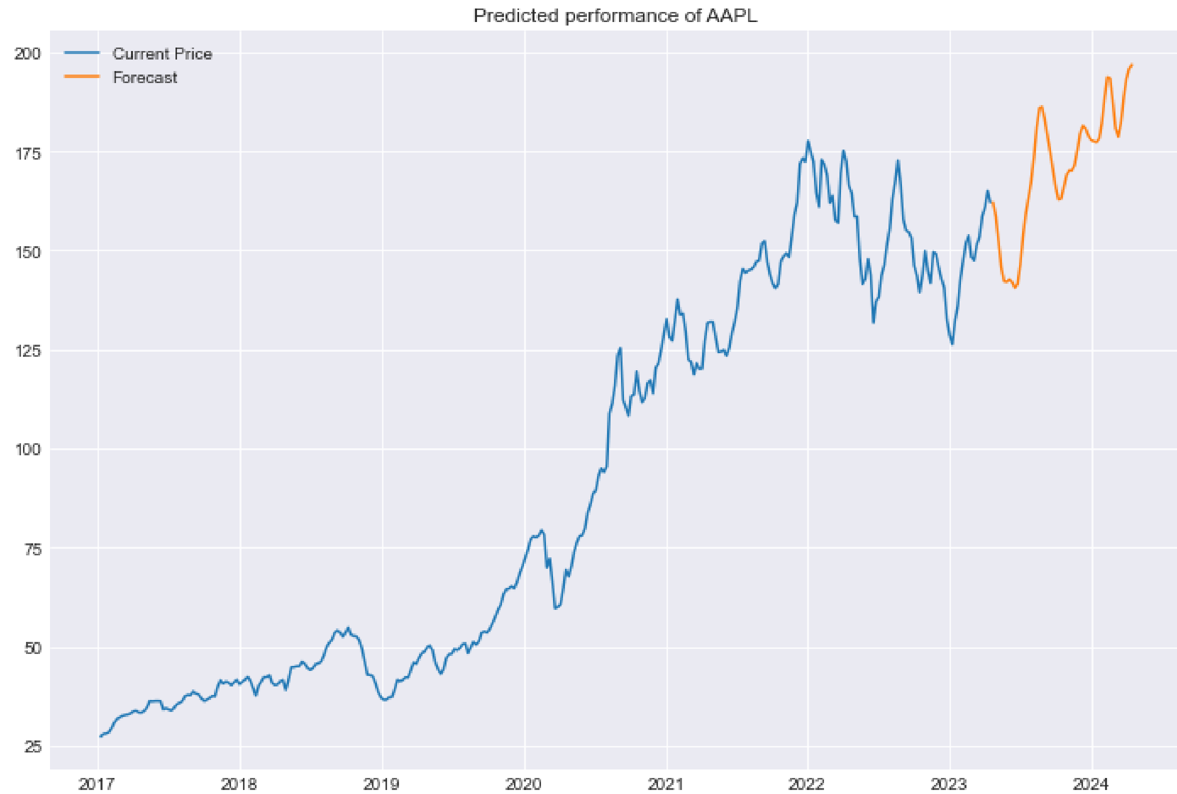
5.8 Using Prophet to get forecast of 'AAPL' for the next year

Now that we have a model, we can make predictions of the next year:

In [28]:
```
#get stock data
df = yf.download('AAPL',start=start_date,end=end_date)
df_close = df[['Adj Close']]

#resmapling by weekly
df_weekly = df_close.resample('W').mean()

df_weekly_diff = df_weekly.diff(periods=1)
df_weekly_diff.dropna(inplace=True)


#setting up df to be able to run Prophet

#reset index
df_weekly_diff.reset_index(inplace=True)

#rename columns per prophet conventions
df_weekly_diff.rename(columns={'Adj Close': 'y',
                        'Date':'ds'},inplace=True)



#instantiate
m = Prophet(seasonality_mode='multiplicative',
            weekly_seasonality=True,
            daily_seasonality = True,
            yearly_seasonality = True,
            interval_width=0.90,
            seasonality_prior_scale = 0.4,
            changepoint_prior_scale = 0.3
           )

#fit
m.fit(df_weekly_diff)

#forecasts - creating future dates using in built make_future_dataframe method
future = m.make_future_dataframe(periods=52,freq='W',include_history=False)

#predicting yhat
forecast = m.predict(future)

#creating a df of predicted values
forecast_values = forecast[['ds', 'yhat']]

#set the index for the plots
forecast_values.set_index('ds',inplace=True)
df_weekly_diff.set_index('ds',inplace=True)


#taking the inverse difference of the predicted values to get the original valu
# the inverse diff is the cumsum of the first value of the org series & the fi
forecast_values.rename(columns={'yhat':'y'},inplace=True)
invdiff = np.r_[df_weekly['Adj Close'].iloc[-1],forecast_values['y'][1:]].cums
invdiff_df = pd.DataFrame(data=invdiff,index = forecast_values.index,columns=[

#plotting results
fig,ax=plt.subplots(figsize=(12,8))
ax.plot(df_weekly,label='Current Price')
```

```
ax.plot(invdiff_df['y'],label='Forecast')
ax.set_title('Predicted performance of AAPL')
ax.legend();
```

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



The model predicts 'AAPL' to be to close to `$200` , 52 weeks from now, from it's current value of approx. `$165`

6 Building Portfolio

6.1 Predictions of chosen stock

Now that the investor has looked at some metrics and future performance of his/her chosen stock, he/she can now look at building a portfolio for the future. Following are some functions that will be used in the portfolio builder

Following function is used to plot past and forecast prices of a stock:

```
In [29]: def plot_forecast_price(ticker):
             #get stock data
             df = yf.download(ticker,start=start_date,end=end_date)
             df_close = df[['Adj Close']]

             #resmapling by weekly
             df_weekly = df_close.resample('W').mean()
             df_weekly_diff = df_weekly.diff(periods=1)
             df_weekly_diff.dropna(inplace=True)


             #setting up df to be able to run Prophet
             #reset index
             df_weekly_diff.reset_index(inplace=True)
             #rename columns per prophet conventions
             df_weekly_diff.rename(columns={'Adj Close': 'y',
                                          'Date':'ds'},inplace=True)


             #instantiate
             m = Prophet(seasonality_mode='multiplicative',
                         weekly_seasonality=True,
                         daily_seasonality = True,
                         yearly_seasonality = True,
                         interval_width=0.90,
                         seasonality_prior_scale = 0.4,
                         changepoint_prior_scale = 0.3
                        )

             #fit
             m.fit(df_weekly_diff)

             #forecasts - creating future dates using in built make_future_dataframe me
             future = m.make_future_dataframe(periods=52,freq='W',include_history=False

             #predicting yhat
             forecast = m.predict(future)

             #creating a df of predicted values
             forecast_values = forecast[['ds', 'yhat']]

             #set the index for the plots
             forecast_values.set_index('ds',inplace=True)
             df_weekly_diff.set_index('ds',inplace=True)


             #taking the inverse difference of the predicted values to get the original
             # the inverse diff is the cumsum of the first value of the org series & the
             forecast_values.rename(columns={'yhat':'y'},inplace=True)
             invdiff = np.r_[df_weekly['Adj Close'].iloc[-1],forecast_values['y'][1:]].
             invdiff_df = pd.DataFrame(data=invdiff,index = forecast_values.index,colum

             #plotting results
             fig,ax=plt.subplots(figsize=(12,8))
             ax.plot(df_weekly,label='Current Price')
             ax.plot(invdiff_df['y'],label='Forecast')
             ax.set_title(f'Predicted values of {ticker}')
```

```
    ax.legend();
```

Following functions is to get the current price of a stock:

In [30]:
```
def get_current_price(ticker):
    df = yf.download(ticker,start=date.today())
    ticker_df = df[['Adj Close']]
    current_price = round(float(ticker_df.iloc[0]),2)

    return current_price
```

Following function is to get the last forecasted price of the stock:

In [31]:
```
def get_future_price(ticker):
     #get stock data
    df = yf.download(ticker,start=start_date,end=end_date)
    df_close = df[['Adj Close']]

    #resmapling by weekly
    df_weekly = df_close.resample('W').mean()
    df_weekly_diff = df_weekly.diff(periods=1)
    df_weekly_diff.dropna(inplace=True)


    #setting up df to be able to run Prophet
    #reset index
    df_weekly_diff.reset_index(inplace=True)
    #rename columns per prophet conventions
    df_weekly_diff.rename(columns={'Adj Close': 'y',
                         'Date':'ds'},inplace=True)


    #instantiate
    m = Prophet(seasonality_mode='multiplicative',
                weekly_seasonality=True,
                daily_seasonality = True,
                yearly_seasonality = True,
                interval_width=0.90,
                seasonality_prior_scale = 0.4,
                changepoint_prior_scale = 0.3
               )

    #fit
    m.fit(df_weekly_diff)

    #forecasts - creating future dates using in built make_future_dataframe me
    future = m.make_future_dataframe(periods=52,freq='W',include_history=False

    #predicting yhat
    forecast = m.predict(future)

    #creating a df of predicted values
    forecast_values = forecast[['ds', 'yhat']]

    #set the index for the plots
    forecast_values.set_index('ds',inplace=True)
    df_weekly_diff.set_index('ds',inplace=True)

    #taking the inverse difference of the predicted values to get the original
    # the inverse diff is the cumsum of the first value of the org series & the
    forecast_values.rename(columns={'yhat':'y'},inplace=True)
    invdiff = np.r_[df_weekly['Adj Close'].iloc[-1],forecast_values['y'][1:]].
    invdiff_df = pd.DataFrame(data=invdiff,index = forecast_values.index,colum

    #get the last value
    last_price = round(float(invdiff_df.iloc[-1]),2)
```

```
    return last_price
```

The `portfolio` function combines all of the above and outputs the total returns

In [32]:

```
#putting it all together

def portfolio(amount,stocks):
    amount = amount
    stocks = stocks
    break_up = round(amount/len(stocks),2)
    current_prices = []
    future_prices = np.array([])
    n_shares = np.array([])
    return_pct = np.array([])

    for st in stocks:

        cp = get_current_price(st)
        fp = get_future_price(st)
        current_prices.append(cp)
        future_prices = np.append(future_prices,fp)

        print(f'Current price of {st} is {cp}')
        print(f'Forecast price of {st} is {fp}')

        def check():
            for i in current_prices:
                if i < break_up:
                    continue # there is nothing below continue to ignore. s
                else: # once the outer loop reaches the value of nflx and the
                    return ('Invest amount per stock is too low. Please adjust
                    break #breaks the inner loop
                break #this break is to ignore the outer loop as well as there

    check()

    if check() == None:
        for i in current_prices:
            share_buy = round(break_up/i)
            n_shares = np.append(n_shares,share_buy)
        sell_amounts = n_shares*future_prices
        returns = sell_amounts/break_up - 1
        invest_amt = round(break_up/amount,2)
        return_pct = np.append(return_pct,returns*invest_amt)
        cum_returns = round(np.sum(return_pct) * 100,2)
        print('\n')
        print(f'Amount invested in each stock: ${break_up}')
        print(f'Cumulative returns: {cum_returns}%')

    else:
        print('\n')
        print(f'Amount invested in each stock: ${break_up}')
        print('Investment is too low')
        print('Please pick a different stock or increase investment')
```

As outlined earlier, following is an example of how diversification works

If an investor is chooses to invest only in the automotive companies, here's what that would look like:

In [33]:
```
# Tesla,General Motors and Ford
portfolio(1000,['TSLA','GM','F'])
```

```
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
Current price of TSLA is 185.0
Forecast price of TSLA is 97.81
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
Current price of GM is 34.48
Forecast price of GM is 34.41
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
Current price of F is 12.52
Forecast price of F is 14.8


Amount invested in each stock: $333.33
Cumulative returns: -6.01%
```

Based on the output, the investor can expect to lose money if he/she were to invest $1000 in Tesla,GM and Ford in the next year.

Now, let's create a mix of companies:automotive,tech and health care

In [34]:
```
#General Motors, Apple, Pfizer
portfolio(1000,['GM','AAPL','PFE'])
```

```
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
Current price of GM is 34.48
Forecast price of GM is 34.2
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
Current price of AAPL is 165.21
Forecast price of AAPL is 196.72
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
Current price of PFE is 41.19
Forecast price of PFE is 44.48


Amount invested in each stock: $333.33
Cumulative returns: 9.04%
```

From the above examples, it is quiet clear to see the benefit of diversifying.

In [35]:
```
# very little investment
portfolio(50,['GM','AAPL','PFE'])
```

```
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
Current price of GM is 34.48
Forecast price of GM is 34.33
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
Current price of AAPL is 165.21
Forecast price of AAPL is 196.55
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
Current price of PFE is 41.19
Forecast price of PFE is 44.44


Amount invested in each stock: $16.67
Investment is too low
Please pick a different stock or increase investment
```

7 Conclusions

7.1 Limitations

1. All the models are purely mathematical models and cannot take into account black swan events.
2. More sophisticated models using Deep Learning can be built to get more accurate forecasts.
3. Dividend data is not incorporated while calculating overall returns.
4. Currently, the invested amount is distributed equally amongst all the stocks. The amounts can be tuned based on the investor's appetite for risk.

7.2 Recommendations

1. By plugging in amounts and companies in the model, the investor can play around and maximize his/her returns.
2. Looking at the stock market in general, there was a drastic spike around 2020. There has not been a decline to pre-2020 levels and hence, it might be prudent to collect past data only from 2020 onwards rather than from all the way back to 2017.