Project 1 - Regression Model Stock Predictor

Objective

Stock investiment is probably one of the hardest investment to master because of the unpredictable natural of stock market. Experts have studied the market and derived several technical analysis for predicting stuck market. Unfortunately, most of the technical analysis are quite complex and not many people know how to use them. Even people with good understanding of the technical analysis would need to spend a considerable amount of time analyzing the data before reaching a reasonable confident conclusion. The objective of this project is to train a regression model based on the available historical stuck price. Once the model is trained, the user should be able to provide the current stock price and get the prediction of the price for the next 30 days using the model.

Goals

The goal of this project is to analysis the available historical stock price data and use the data to train a regression model for predicting the stock price of the next 30 days.

Project Outline

The project has 3 steps:

1. Explore and analyze the dataset
2. Modify and prepare the dataset for training
3. Evaluate various regression model and compare their performance

1. Explore and analyze the dataset

Download the dataset from Quandl if it is not available in local disk. Note: For this project, AAPL stock price is used since it is one of the few stock price that Quandl provide for free

In [1]:
```
import os
import quandl
import pandas as pd
quandl.ApiConfig.api_key = "nVD4QZoCjEQijoM1Pvzz"
# download data from quandl and save it in a csv file if the file does n
ot exist
if not os.path.exists('data/AAPL.csv'):
    data = quandl.get_table("WIKI/PRICES", qopts={'columns': ['ticker',
'date', 'adj_close', 'adj_volume']},
                            ticker=AAPL, paginate=True)
    if data.shape[0] > 1:
        data.to_csv('data/AAPL.csv', '\t')
# read the data from csv file
df = pd.read_csv('data/AAPL.csv', usecols=['date', 'adj_close'], delimit
er='\t', header=0,
                 index_col='date', parse_dates=True)
df.head()
```

Out[1]:

	adj_close
date	
2018-03-27	168.340
2018-03-26	172.770
2018-03-23	164.940
2018-03-22	168.845
2018-03-21	171.270

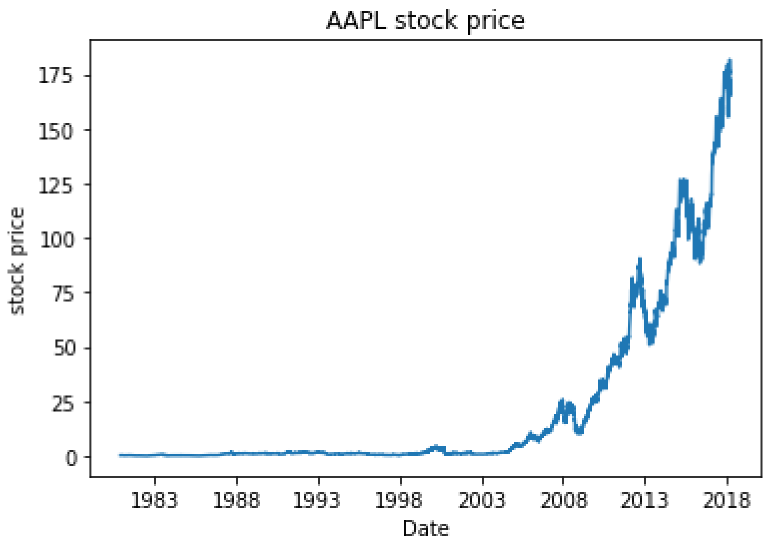
In [2]:
```
# check dataset's shape and info to see if cleaning is required
df.shape
```

Out[2]: (9400, 1)

In [3]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 9400 entries, 2018-03-27 to 1980-12-12
Data columns (total 1 columns):
adj_close    9400 non-null float64
dtypes: float64(1)
memory usage: 146.9 KB
```

In [4]:
```
# above info shows that there is no null value and the dataset is clean
# The line plot of the dataset is created which shows the tread of the d
ataset
import matplotlib.pyplot as plt
df.sort_index(inplace=True)
plt.plot(df)
plt.title('AAPL stock price')
plt.xlabel('Date')
plt.ylabel('stock price')
plt.show()
```



In [5]:
```
df.describe()
```

Out[5]:

	adj_close
count	9400.000000
mean	21.567664
std	39.271266
min	0.161731
25%	0.922730
50%	1.437445
75%	20.294924
max	181.720000

2. Modify and prepare the dataset for training

As shown in the plot above, the data has an increasing trend. To make a better prediction, we can remove the tread by differencing

In [6]:
```
from pandas import Series
def difference(dataset, interval=1):
    diff = list()
    for i in range(interval, len(dataset)):
        value = dataset[i] - dataset[i - interval]
        diff.append(value)
    return Series(diff)

raw_values = df.values
diff_series = difference(raw_values, 1)
diff_values = diff_series.values
diff_values = diff_values.reshape(len(diff_values), 1)
print(diff_values[:5])
print(diff_values[-5:])
```

```
[[array([-0.02205422])]
 [array([-0.02940563])]
 [array([ 0.00911574])]
 [array([ 0.01117414])]
 [array([ 0.02381856])]]
[[array([-3.97])]
 [array([-2.425])]
 [array([-3.905])]
 [array([ 7.83])]
 [array([-4.43])]]
```

As shown in the output above, the diff_values still has a large range. This can degrade the predictive performance of many machine learning algorighms. Unscaled data can also slow down or even prevent the convergence of many gradient-based estimators. Many estimators are designed with the assumption that each feature takes values close to zero. There are many differenct scalers. For this project, MinMaxScaler that rescale values to -1, 1 is used

In [7]:
```
from sklearn.preprocessing import MinMaxScaler
# rescale values to -1, 1
scaler = MinMaxScaler(feature_range=(-1, 1))
scaled_values = scaler.fit_transform(diff_values)
scaled_values = scaled_values.reshape(len(scaled_values), 1)
print(scaled_values[:5])
print(scaled_values[-5:])
```

```
[[ 0.02246711]
 [ 0.02155191]
 [ 0.02634758]
 [ 0.02660384]
 [ 0.02817799]]
[[-0.46902807]
 [-0.27668499]
 [-0.46093597]
 [ 1.        ]
 [-0.52629527]]
```

```
/Users/allenliu/anaconda3/lib/python3.6/site-packages/sklearn/utils/val
idation.py:475: DataConversionWarning: Data with input dtype object was
converted to float64 by MinMaxScaler.
  warnings.warn(msg, DataConversionWarning)
```

The stock prediction is a multi-step forecast problem. For a given time step, the model is required to make the next 30 day prediction. That is given t-1, forecast t, t+1, t+2... t+30. A key function to help transform time series data into multi-step forecast problem is the shift() function. By shifting the input (X) by -1 for 30 times, we can mimic the 30 days forecast.

i.e.

```
X    y1  y2  y3  .... y30
t    t+1 t+2 t+3 .... t+30
```

In [8]:
```
from pandas import DataFrame
from pandas import concat

# convert time series into multi-step problem
def series_to_supervised(data, n_in=1, n_out=1, dropnan=True):
    n_vars = 1 if type(data) is list else data.shape[1]
    df = DataFrame(data)
    cols, names = list(), list()
    # input sequence (t-n, ... t-1)
    for i in range(n_in, 0, -1):
        cols.append(df.shift(i))
        names += [('var%d(t-%d)' % (j + 1, i)) for j in range(n_vars)]
    # forecast sequence (t, t+1, ... t+n)
    for i in range(0, n_out):
        cols.append(df.shift(-i))
        if i == 0:
            names += [('var%d(t)' % (j + 1)) for j in range(n_vars)]
        else:
            names += [('var%d(t+%d)' % (j + 1, i)) for j in range(n_vars
)]
    # put it all together
    agg = concat(cols, axis=1)
    agg.columns = names
    # drop rows with NaN values
    if dropnan:
        agg.dropna(inplace=True)
    return agg

supervised = series_to_supervised(scaled_values, 1, 30)
supervised_values = supervised.values
print(supervised_values.shape)
print(supervised_values)
```

```
(9369, 31)
[[ 0.02246711  0.02155191  0.02634758 ...,  0.02429752  0.02475512
   0.02338232]
 [ 0.02155191  0.02634758  0.02660384 ...,  0.02475512  0.02338232
   0.02314436]
 [ 0.02634758  0.02660384  0.02817799 ...,  0.02338232  0.02314436
   0.02224746]
 ...,
 [-0.50015155  0.10613377  0.86430164 ..., -0.46902807 -0.27668499
  -0.46093597]
 [ 0.10613377  0.86430164  0.22813779 ..., -0.27668499 -0.46093597  1.
        ]
 [ 0.86430164  0.22813779  0.40242926 ..., -0.46093597  1.         -0.5
2629527]]
```

Next, we can split the data into training and test sets. Set n_test = 3700

In [9]:
```
n_test = 3700
train, test = supervised_values[0:-n_test], supervised_values[-n_test:]
```

In [10]:
```
#reshape training into X,y
X, y = train[:, 0:1], train[:, 1:]
print(X[0])
print(y[0])
```

```
[ 0.02246711]
[ 0.02155191  0.02634758  0.02660384  0.02817799  0.02773869  0.0275007
4
  0.02817799  0.03070396  0.02612793  0.02360197  0.02340062  0.0258899
8
  0.02383992  0.02246711  0.02270506  0.02405957  0.0281963   0.0247368
2
  0.02316267  0.02545068  0.02634758  0.02475512  0.0286356   0.0234006
2
  0.02634758  0.02588998  0.02499308  0.02429752  0.02475512  0.0233823
2]
```

3. Evaluate various regression model and compare their performance

1. SVR Model

First, train the SVR model and evaulate it's performance. To determine the best parameter for the SVR, GridSearchCV is used. Since this is a multi-step forcast problem, MultiOutputRegressor is used as a wrapper to extend SVR that does not natively support multi-target regression.

In [11]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVR
from sklearn.multioutput import MultiOutputRegressor

Cs = [0.001, 0.01, 0.1, 1, 10]
gammas = [0.001, 0.01, 0.1, 1]
kernels = ['linear','poly','rbf','sigmoid']
param_grid = {'estimator__C': Cs, 'estimator__gamma': gammas, 'estimator
__kernel': kernels }
regr = MultiOutputRegressor(SVR())
grid_search = GridSearchCV(regr, param_grid)
grid_search.fit(X, y)
print(grid_search.best_params_)
```

```
{'estimator__C': 10, 'estimator__gamma': 1, 'estimator__kernel': 'rbf'}
```

In [12]:
```
model = MultiOutputRegressor(SVR(C=10, gamma=1)).fit(X,y)
```

Using the trained model, we can make prediction using the test data

In [13]:
```
def make_forecasts(model, test):
    forecasts = list()
    for i in range(len(test)):
        X, y = test[i, 0:1], test[i, 1:]
        # make forecast
        forecast = forecast_svr(model, X)
        # store the forecast
        forecasts.append(forecast)
    return forecasts

def forecast_svr(model, X):
    X = X.reshape(1,len(X))
    forecast = model.predict(X)
    # convert to array
    return [x for x in forecast[0, :]]

forecasts = make_forecasts(model, test)
print(forecasts[0])
```

```
[-0.0071525089142082665, -0.017743490129980694, -0.019249711234373 94, -
0.01779804067192628, -0.016355577546203461, -0.016765930048189, -0.0166
2034368589714, -0.019853215052171649, -0.023605966456182566, -0.0158334
93557284851, -0.02213598322073182, -0.017267404154887062, -0.0175698583
56476536, -0.016163867990052899, -0.014115489282952934, -0.017677271316
784121, -0.020336958922423924, -0.018699210066458485, -0.01251698096353
7719, -0.018448041424192346, -0.019653691169197379, -0.0190609343402392
41, -0.017770560727786502, -0.017797989111241669, -0.02077933593805526
8, -0.019028017164611791, -0.019002138089551757, -0.022170121115243982,
-0.016213327368827905, -0.015130754293692511]
```

After the forecasts have been made, we need to invert the transforms to return the values back into the original scale. This is needed so that we can calculate error scores and plots that are comparable with the actual test output.

In [14]:
```
from numpy import array

def inverse_transform(series, forecasts, scaler, n_test):
    inverted = list()
    for i in range(len(forecasts)):
        # create array from forecast
        forecast = array(forecasts[i])
        forecast = forecast.reshape(1, len(forecast))
        # invert scaling
        inv_scale = scaler.inverse_transform(forecast)
        inv_scale = inv_scale[0, :]
        # invert differencing
        index = len(series) - n_test + i - 1
        last_ob = series.values[index]
        inv_diff = inverse_difference(last_ob, inv_scale)
        # store
        inverted.append(inv_diff)
    return inverted

# invert differenced forecast
def inverse_difference(last_ob, forecast):
    # invert first forecast
    inverted = list()
    inverted.append(forecast[0] + last_ob)
    # propagate difference forecast using inverted first value
    for i in range(1, len(forecast)):
        inverted.append(forecast[i] + inverted[i - 1])
    return inverted

forecasts = inverse_transform(df, forecasts, scaler, n_test + 2)
print(forecasts[0])
```

```
[array([ 0.99817699]), array([ 0.65313024]), array([ 0.29598473]), arra
y([-0.04950019]), array([-0.38339851]), array([-0.72059298]), array([-
1.05661804]), array([-1.4186112]), array([-1.81074842]), array([-2.1404
5308]), array([-2.52078262]), array([-2.8620052]), array([-3.2056572
5]), array([-3.53801565]), array([-3.8539204]), array([-4.19843525]), a
rray([-4.56431409]), array([-4.91703769]), array([-5.22010239]), array
([-5.57080846]), array([-5.93119894]), array([-6.28682809]), array([-6.
63209229]), array([-6.9775768]), array([-7.34700905]), array([-7.702373
79]), array([-8.05753066]), array([-8.43813442]), array([-8.7708901]),
array([-9.09494999])]
```

Get the actual test target and transform the data to their original scale

In [15]:
```
actual = [row[1:] for row in test]
actual = inverse_transform(df, actual, scaler, n_test + 2)
```

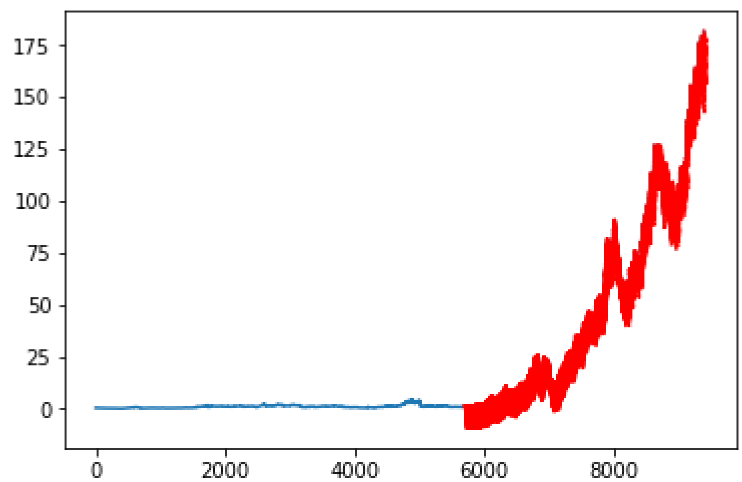
Evaluate the RMSE for each forecast time step

In [16]:
```
from math import sqrt
from sklearn.metrics import mean_squared_error

def evaluate_forecasts(test, forecasts, n_lag, n_seq):
    for i in range(n_seq):
        actual = [row[i] for row in test]
        predicted = [forecast[i] for forecast in forecasts]
        rmse = sqrt(mean_squared_error(actual, predicted))
        print('t+%d RMSE: %f' % ((i + 1), rmse))

evaluate_forecasts(actual, forecasts, 1, 30)
```

```
t+1 RMSE: 1.213634
t+2 RMSE: 1.647081
t+3 RMSE: 2.128069
t+4 RMSE: 2.626768
t+5 RMSE: 2.991953
t+6 RMSE: 3.371821
t+7 RMSE: 3.812751
t+8 RMSE: 4.222598
t+9 RMSE: 4.781010
t+10 RMSE: 5.132431
t+11 RMSE: 5.510405
t+12 RMSE: 5.948555
t+13 RMSE: 6.316946
t+14 RMSE: 6.722795
t+15 RMSE: 7.058245
t+16 RMSE: 7.463258
t+17 RMSE: 7.826017
t+18 RMSE: 8.209595
t+19 RMSE: 8.592708
t+20 RMSE: 9.047438
t+21 RMSE: 9.449130
t+22 RMSE: 9.877070
t+23 RMSE: 10.298055
t+24 RMSE: 10.730791
t+25 RMSE: 11.114181
t+26 RMSE: 11.577285
t+27 RMSE: 12.035677
t+28 RMSE: 12.392909
t+29 RMSE: 12.724031
t+30 RMSE: 13.123339
```

Plot the forecast aginst the original dataset

In [17]:
```
def plot_forecasts(series, forecasts, n_test):
    # plot the entire dataset in blue
    plt.plot(series.values)
    # plot the forecasts in red
    for i in range(len(forecasts)):
        off_s = len(series) - n_test + i - 1
        off_e = off_s + len(forecasts[i]) + 1
        xaxis = [x for x in range(off_s, off_e)]
        yaxis = [series.values[off_s]] + forecasts[i]
        plt.plot(xaxis, yaxis, color='red')
    # show the plot
    plt.show()

plot_forecasts(df, forecasts, n_test + 2)
```



As shown in the graph above, there is large fluctuation in the forecast data compare with the actual data. However, the model is generally good as it can predict the correct trend of the stock price

2. DecisionTreeRegressor Model

Second, let's train a DecisionTreeRegressor and see how it performes

In [18]:
```
from sklearn.tree import DecisionTreeRegressor
modelDTR = MultiOutputRegressor(DecisionTreeRegressor(random_state=0)).f
it(X,y)
```

In [19]:
```
forecastsDTR = make_forecasts(modelDTR, test)
print(forecastsDTR[0])
```

```
[0.024823655524463875, 0.026401758620267824, 0.025649068894848217, 0.02
7576245486498598, 0.023918246144613187, 0.023256460975407824, 0.0231001
05138725593, 0.025754518180052251, 0.024598212225062741, 0.024078238163
543739, 0.026263583694828684, 0.023703711391957573, 0.02527454212326771
6, 0.025307267763498651, 0.025656341259345763, 0.02764533294921704, 0.0
25678158352837058, 0.025267269758768581, 0.023460087181317223, 0.027845
322972879016, 0.027078088518467033, 0.025176365202558625, 0.02445276493
5128532, 0.026699925564635272, 0.023278278068894816, 0.0252236355717894
02, 0.028067130090029892, 0.027398072556327681, 0.023798252130415861,
0.027565336939754659]
```

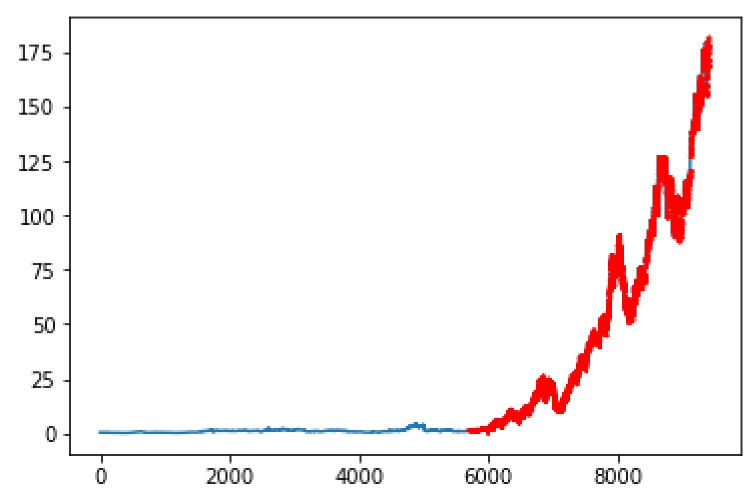
In [20]:
```
forecastsDTR = inverse_transform(df, forecastsDTR, scaler, n_test + 2)
print(forecastsDTR[0])
```

```
[array([ 1.25502622]), array([ 1.26457715]), array([ 1.26808207]), arra
y([ 1.28706709]), array([ 1.27666914]), array([ 1.26095539]), array([
1.24398571]), array([ 1.24833766]), array([ 1.24340155]), array([ 1.234
28875]), array([ 1.24272978]), array([ 1.23060858]), array([ 1.2311051
1]), array([ 1.23186451]), array([ 1.23542785]), array([ 1.25496781]),
array([ 1.2587064]), array([ 1.25914451]), array([ 1.24506639]), array
([ 1.26621278]), array([ 1.28119634]), array([ 1.28090426]), array([ 1.
27479985]), array([ 1.2867458]), array([ 1.2712073]), array([ 1.2712949
2]), array([ 1.29422298]), array([ 1.31177681]), array([ 1.30041501]),
array([ 1.3193124])]
```

In [21]:
```
evaluate_forecasts(actual, forecastsDTR, 1, 30)
```

```
t+1 RMSE: 1.059251
t+2 RMSE: 1.498298
t+3 RMSE: 1.824373
t+4 RMSE: 2.108605
t+5 RMSE: 2.375138
t+6 RMSE: 2.601160
t+7 RMSE: 2.798409
t+8 RMSE: 3.011196
t+9 RMSE: 3.228060
t+10 RMSE: 3.406364
t+11 RMSE: 3.572429
t+12 RMSE: 3.723011
t+13 RMSE: 3.868358
t+14 RMSE: 3.999081
t+15 RMSE: 4.143604
t+16 RMSE: 4.284882
t+17 RMSE: 4.421200
t+18 RMSE: 4.557767
t+19 RMSE: 4.699522
t+20 RMSE: 4.835123
t+21 RMSE: 4.990988
t+22 RMSE: 5.107230
t+23 RMSE: 5.230035
t+24 RMSE: 5.348993
t+25 RMSE: 5.456401
t+26 RMSE: 5.570900
t+27 RMSE: 5.660230
t+28 RMSE: 5.782262
t+29 RMSE: 5.896631
t+30 RMSE: 6.028059
```

In [22]:
```
plot_forecasts(df, forecastsDTR, n_test + 2)
```



Conclusion:

Both SVR and DecisionTreeRegressor model can predict the trend of the stock price. However, DecisionTreeRegressor seems to perform better than SVR by comparing both the RMSE and the plot.