Adventures In Day Trading

Anthony Mayer

At the intersection of gambling and legitimate investing lies day trading. Many amateurs claim to make money, a few professionals seem able to. With an endless stream of data, for as many stocks as one could want, and the potential for profit, day trading seems like an ideal arena for predictive modeling. It is easy to think that with the help of fast computers and sophisticated models that making money would be easy. But many believe (see the controversial efficient market hypothesis) that predicting future stock prices is impossible.

The data in day trading is sequential in time, therefore we are in the arena of time series forecasting. Stating the problem mathematically, given the data,

$$D = ((\underline{x}_t, y_t))_{t=1}^{n}.$$

What is $y_{t+1}, y_{t+2}, \ldots, y_{t+k}$ where y_k is the stock price at time $t = k$. One choice is polynomial regression in time. Model y_t as

$$y_t = \sum_{j=0}^{p} a_j t^j.$$

Where the coefficients a_j are found with least squares linear regression with at least $p+1$ data points in the past. There are immediate problems regressing on time. Since we want to forecast into the future we need to query the model outside the domain it was trained on. Polynomial models are good at interpolating data but quickly diverge outside of the domain they were fit to.

A better option may be to off-set the training data D. First augment the predictor vector $\underline{x}_t$ with y_t. Then stagger the data so $\underline{x}_k$ is paired with y_{k+1}. The new data set is

$$D' = ((\underline{x}_t, y_{t+1}))_{t=1}^{n-1}.$$

Now a model can be trained to predict the next price from the current data. Note that in this framework we are only predicting one time step ahead. To predict k time steps ahead we would change D to,

$$D' = ((\underline{x}_t, y_{t+1}, \ldots, y_{t+k}))_{t=1}^{n-k}.$$

I focused on predicting one time step out, typically on a one minute interval. Also time is no longer explicitly a predictor.

What the predictor vector $\underline{x}_t$ is, is largely a choice. In the language of machine learning that choice is called feature engineering. The information available from the IEX trading platform is the last sale price, ask price, bid price, sale volume, and a time stamp. After trying various predictors I settled on the predictor vector,

$$\underline{x}_t = [y_t - y_{t-1}, y_t - y_{t-2}, \ldots, y_t - y_{t-10}, y_t - y_{avg}]$$

Where y_{avg} is the current daily average price.

Since we have sequential data, and the patterns in the data may be quite complicated, a good choice of model is a recurrent neural network. Specifically an LSTM (long short term memory) network. LSTM networks store long term dependencies in a cell state. The network carefully regulates the flow of information in and out of the cell state. This stability makes LSTM networks much more effective than vanilla recurrent networks.

There are two distinct things the LSTM can be trained to do. One is to classify whether the price will increase, decrease, or stay the same. The other is to predict the next price directly. Of the two I found predicting the residual of the next price and current price most effective. So to get the prediction for the next price, add the output of the network to the current price. Code for both the classification and regression networks are at the end of this paper (code example 1). The implementation is in pytorch.

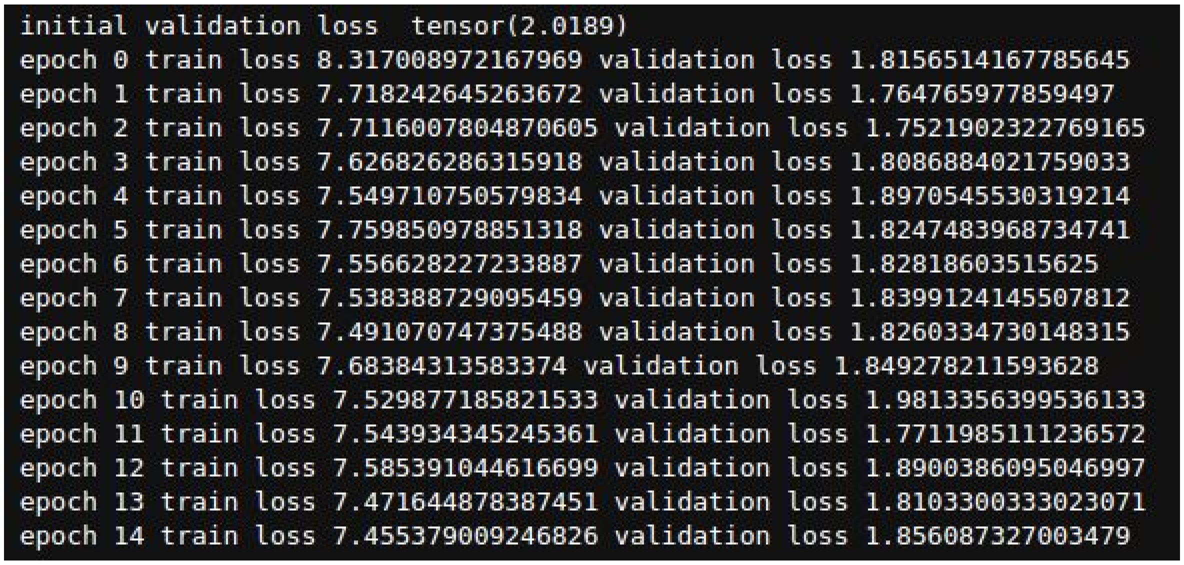
Neural networks need a lot of data. I used the previous ten days of data, for a given stock, to train the networks. This raises the question whether patterns persist in the stock market for that long. It’s hard to tell, but people claim that things like the difference between the current price and the average price over the day, as well as the momentum of the stock price, are very predictive. If that’s true the network should be able to learn those patterns given enough data.

The most important step is training the network. The Adam optimizer was used to train the neural network with a batch size of 32. The hidden and cell states were not kept between days, because after hours trading changes the dynamics of the stock price. The loss function for the classification networks was categorical cross entropy, and for the regression networks the loss was mean squared error.

Training Pseudo code:(Implemented in Code Example 1&2)

```
    Let the ten days of data be,
                    $D = \{D_i\}_{i=1}^{10}$
    Denote the network by
                    $F(.)$ .
    For i = 1:num_epochs:
        Shuffle(D)
        For $D_i \in D$:
            Hidden state -> 0
            Cell state -> 0
            For Batch $\in D_i$:
                $y_{hat}$ = $F(x_{batch})$
                $L$ = loss($y_{hat}, y_{batch}$)
                Grad = $\nabla L$
                Weights -= Adam Update
```

A validation set of one extra day was also used during training to detect overfitting. Implementation of the training loop (code example 1) is in the train method of the pytorch models. An example of a full training cycle,



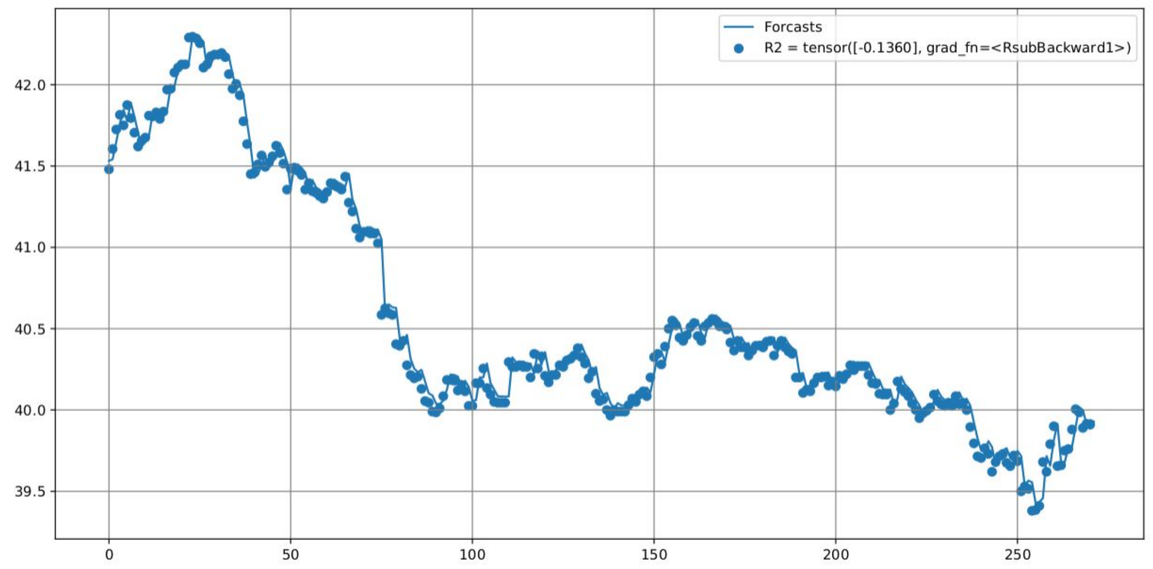
```
initial validation loss  tensor(2.0189)
epoch 0 train loss 8.317008972167969 validation loss 1.8156514167785645
epoch 1 train loss 7.718242645263672 validation loss 1.764765977859497
epoch 2 train loss 7.7116007804870605 validation loss 1.7521902322769165
epoch 3 train loss 7.626826286315918 validation loss 1.8086884021759033
epoch 4 train loss 7.549710750579834 validation loss 1.8970545530319214
epoch 5 train loss 7.759850978851318 validation loss 1.8247483968734741
epoch 6 train loss 7.556628227233887 validation loss 1.82818603515625
epoch 7 train loss 7.538388729095459 validation loss 1.8399124145507812
epoch 8 train loss 7.491070747375488 validation loss 1.8260334730148315
epoch 9 train loss 7.68384313583374 validation loss 1.849278211593628
epoch 10 train loss 7.529877185821533 validation loss 1.9813356399536133
epoch 11 train loss 7.543934345245361 validation loss 1.7711985111236572
epoch 12 train loss 7.585391044616699 validation loss 1.8900386095046997
epoch 13 train loss 7.471644878387451 validation loss 1.8103300333023071
epoch 14 train loss 7.455379009246826 validation loss 1.856087327003479
```

The next figure is a trained model forecasting Micron stock prices over the course of the day. It's important to note that since the model is predicting residuals not the prices directly, that the plots can look deceptively good. We are starting at the current price and adding a small number. So the plots will always look good. The real question is if the forecasts have predictive power. To test this I've introduced a modified R^2 statistic. Typically in the R^2 statistic, the baseline model is the average price y_{avg}. I've modified the statistic to use the previous price y_{t-1} as the baseline prediction.

$$\tilde{R}^2 = 1 - \frac{\sum_i (\hat{y}_i - y_i)^2}{\sum_i (y_{i-1} - y_i)^2}$$

If $\hat{y}_i$ is a better approximation to y_i than y_{i-1} is , then $\tilde{R}^2$ will be positive. If $\hat{y}_i$ is a worse than y_i, $\tilde{R}^2$ will be negative.

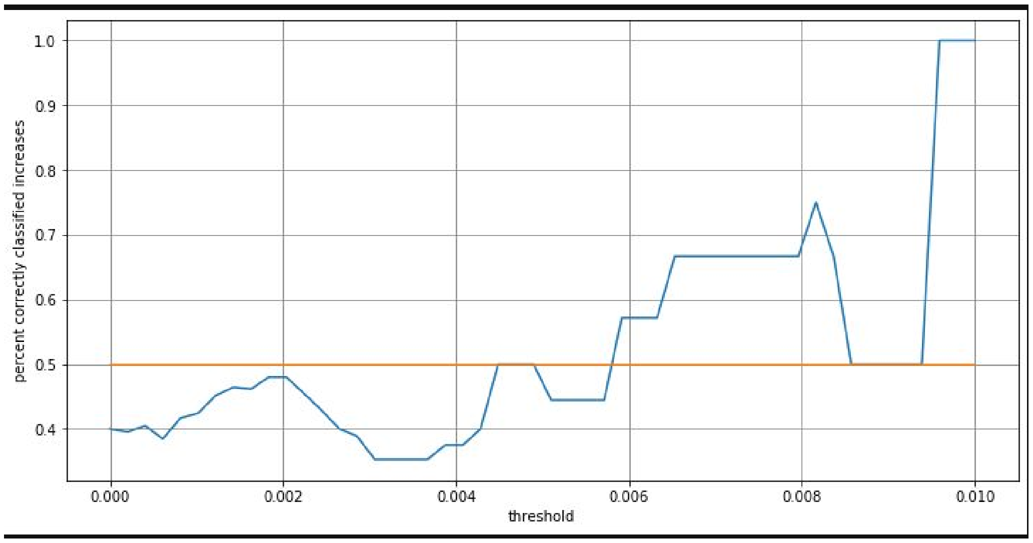
Plot from April 1st, (Code for stock trading class at end of the paper under code example 2.)



Despite the plot looking good, $\tilde{R}^2$ is negative, meaning the predictions had less predictive power than the previous price on average. Typically the $\tilde{R}^2$ values are around zero. This is not a good sign for the models. But maybe there is more value in some predictions than others. Stock prices may be random much of the time but in certain situations are predictable.

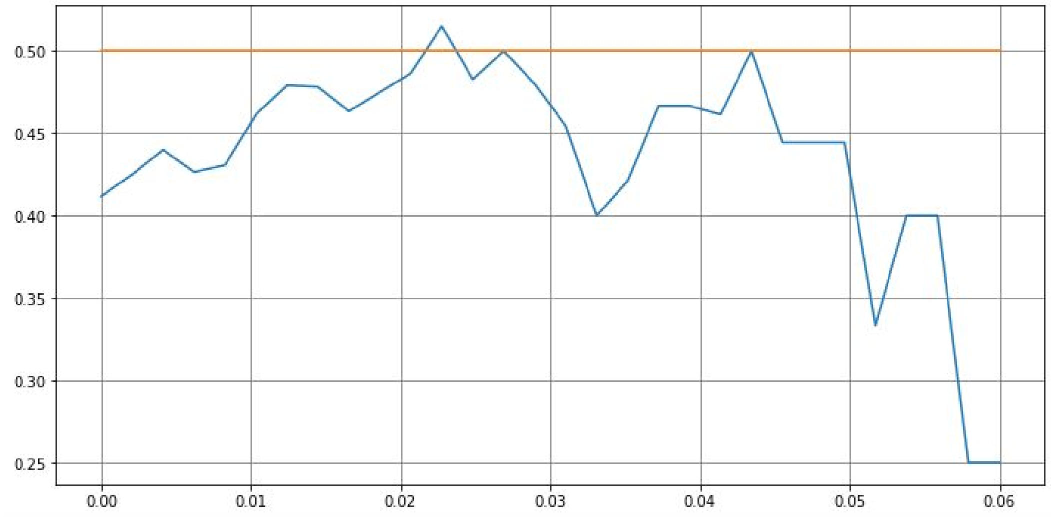
One hypothesis is; the more the model predicts the price to increase, the more confident the model is that the price will go up. To test this we can make a test set and set a threshold increase, t. Next we can run the model on the test set and calculate the percentage of times where the model predicts an increase $\geq t$, and the price actually increases. Next we can plot a range of thresholds and their corresponding accuracies, and see if there are any promising values for t.

The following is a plot that supports the hypothesis,



.

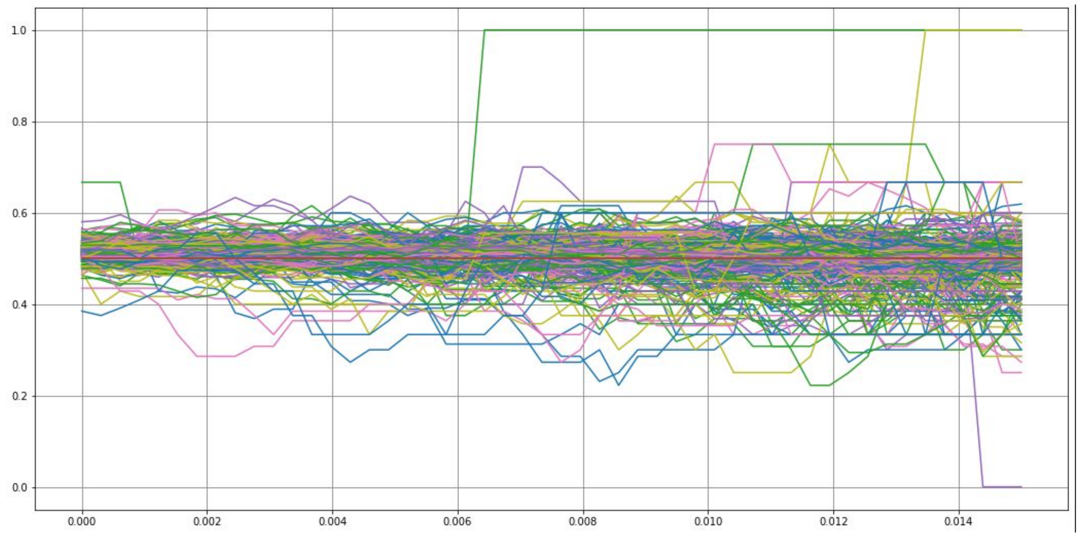
As the threshold increases the accuracy gets higher, suggesting to wait until the model predicts a large increase to buy. Unfortunately resplitting the train and test set and fitting a new model sometimes leads to plots like this,



where the model almost never reatures an accuracy of over 50%.

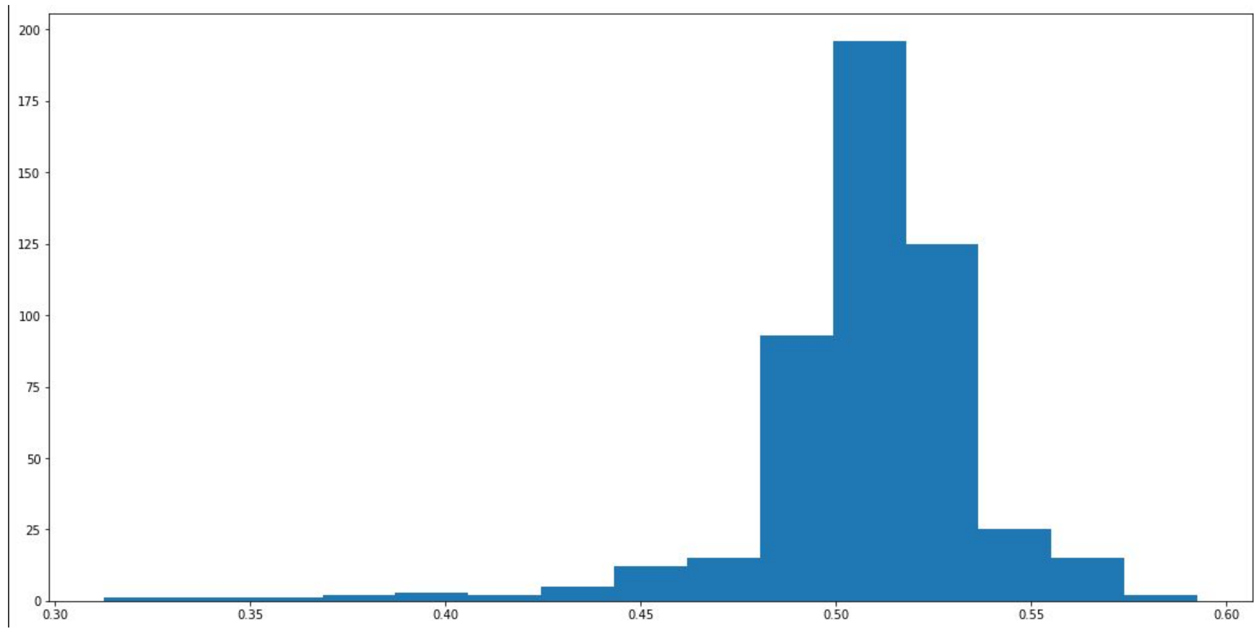
The only way to see which result is more common is to run a cross validation. I tried plotting the results of cross validation to see if there was clearly more density above the 50% line. But the results weren't clear.

Plot of cross validation:



It's hard to get information out of a plot like this. A better way is to pick a threshold, run cross validation and plot a histogram of the accuracies. Training that many models takes a lot of computation so I used a google cloud deep learning vm to train the models on gpu.

To generate the following histogram I ran a 500-Fold cross validation with a threshold of $.08.



From the different thresholds I tried this was the best result and still is not promising. Like all plots, histograms can be deceiving. It looks like there is a bin above 50% that contains the majority of the data, but really that bin contains 50% but is centered around 53%. The data is very tightly centered around 50% and with different bins the histogram would look quite different. With a mean value of 50.6% accuracy, this threshold approach does not seem like a good strategy for trading.

Between the evidence from $\tilde{R}^2$ statistic, as well as the cross validation results, this particular approach with LSTM networks does not seem to work. Perhaps ten days of trading data was too much and it would be better to just use the current days data. The dynamics of the stock market might change so fast only very recent data is relevant for learning patterns.

A new approach:

Instead of trying to predict stock prices, I propose a different problem. Can we predict when the price is close to a local minima or maxima? This may be an easier problem than predicting exactly what the next stock price is going to be. In the language of finance we are going to look for support and resistance lines.

Support lines have the property that as the price approaches them, the price tends to turn around and increase. Resistance lines have the opposite property where the prices turn around and decrease as they approach them. I propose looking for support and resistance lines by regressing on local minima and maxima. The only predictor I'm going to use is time steps. Hopefully the behavior of maxima and minima are less noisy than the rest of the data, allowing the regression lines to be useful for several time steps into the future.

The Resistance Line,

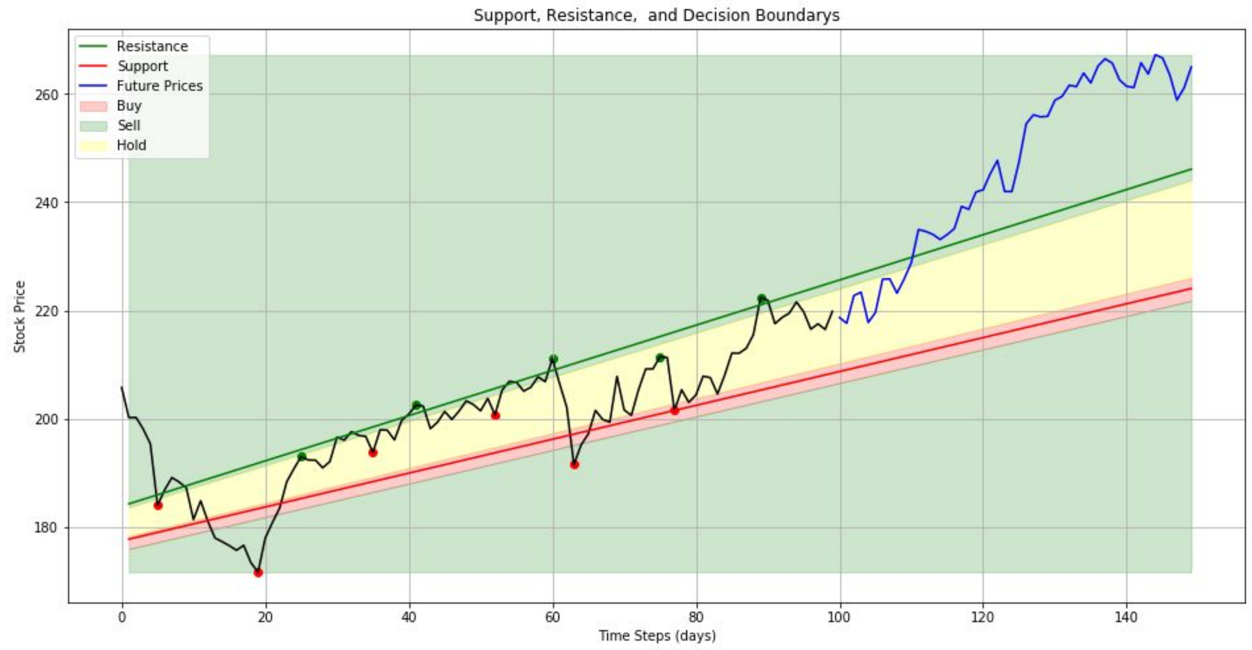
$$R(t) = \beta_0 + \beta_1 t.$$

The Support Line,

$$S(t) = \alpha_0 + \alpha_1 t.$$

The coefficients are found with the least squares estimator using data going 100 time steps in the past. The coefficients are recalculated every time step. The value of 100 was found trying many values and seeing which made the most money. It's also important to define what a local maxima or minima is. A local minima is defined to be a point with the lowest price in a region $\pm n$. Similarly a local maxima is a point with the highest price in a region $\pm n$. Through trial and error the best value of n is 5.

Here's an example of the support and resistance lines applied to apple data. (Code Example 3)



In this approach the decision boundaries for buying, selling, and holding have nice graphical representations. Let d_r be the distance of the price to the resistance line. Let d_s be the distance of the price to the support line.

Decision Rule:(Implemented in Code Example 6)

If the price is between the the support and the resistance lines the decision rules are as follows:

$$\text{If } \frac{d_r}{d_s} \le .1 \text{ sell}$$

$$\text{If } \frac{d_s}{d_r} \le .1 \text{ buy}$$

$$\text{Otherwise Hold}$$

If the price is below the support line,

$$\text{if } p_t \ge .98 * S(t) \text{ buy}$$

$$\text{Otherwise Sell}$$

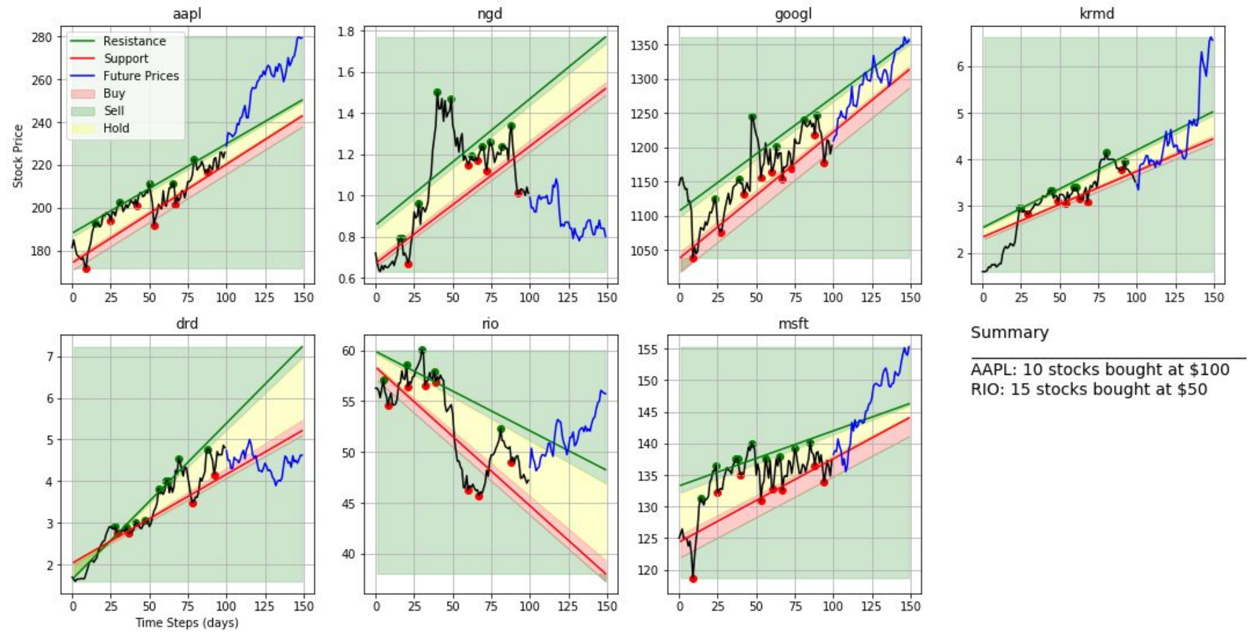
If the price is above the resistance line,

$$\text{Sell}.$$

All of the parameters were found through trial and error.

The reason the stock isn't bought whenever the price is below the resistance line is that we don't want to buy when our bounds are clearly not working. Since the buy region is small it makes sense to track many stocks at once and buy the stock that is within its buy region.

An example of a plot of many stocks, (Code Example 4)



Summary

AAPL: 10 stocks bought at $100
RIO: 15 stocks bought at $50

Here we can either trade ourselves or allow the algorithm to trade automatically. Code that makes these plots in real time is attached to the back.(Code Example 5)

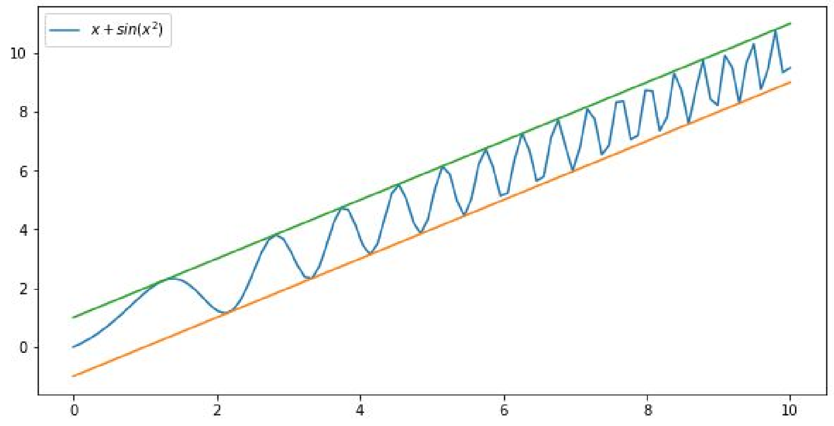
A nice feature of this approach is that it works well for any time step we select. To test the algorithm, last year's data was pulled from yahoo finance, and the algorithm traded with a time step of a day.(Code Example 6)

Results:

Stock	Percent Return
Apple	14.6
Google	12.1
G.E	9.6
Hp	-4.3
Microsoft	5.0
Tesla	10.3
Total	7.6

The results look promising, but it's important to note that 2019 was a very good year for the stock market. Code for the simulations are attached on the back. (Code Example 6)

There's another nice property of these resistance and support lines. Early in the class we tried to find the periods of local maxima and minima. That attempt failed, but consider the function $y = x + sin(x^2)$



Despite the period always changing, the local maxima and minima lie on a line. You can predict if you are near a local extrema purely by how close the point is to the resistance and support lines, without underlying knowledge about the period.

Conclusion

Two very different approaches were taken for day trading. The first was using LSTM neural networks for stock price prediction. I found no evidence that this is an effective strategy for predicting prices. There are, of course, countless other ways to try and use an LSTM for this problem. Different features could be used. Different amounts of training data. Better hyper parameters could be chosen. Trying to exhaust all of those possibilities is a very complex task for a brave person.

The second, more fruitful approach, changes the goal. Instead of trying to predict exact prices, we try to predict a price's proximity to local extrema. A simple approach was taken to do this; do a linear regression on local maxima and minima to find support and resistance lines. Then buy, sell, and hold based on proximity to those lines. Testing suggests this method is profitable. This method could become even better with more refinement. For example, various parameters could change in real time with the data. If the current price is no longer within the support and resistance lines, it could mean the model is looking at data too far in the past. In this case the model could shorten its view when making resistance and support lines. As with the neural network there are endless opportunities for improvement.

Pytorch Models (code example 1)

LSTM Regressor:

```
class LSTM_regressor(nn.Module):
    '''
    LSTM unit that tries predict the difference between the next price and
the current
    price
    '''
    def __init__(self,input_size,h_size,out_size=1):

        super(LSTM_regressor, self).__init__()

        #size of hidden state
        self.input_size = input_size
        self.h_size = h_size
        #initialize hidden state.
        self.hidden = torch.zeros(h_size)
        self.cell = torch.zeros(h_size)
        #define x matrices
        self.Wx_f = nn.Linear(input_size,h_size)
        self.Wx_i = nn.Linear(input_size,h_size)
        self.Wx_o = nn.Linear(input_size,h_size)
        self.Wx_c = nn.Linear(input_size,h_size)
        #define h matrices note that we don't need another bias
        self.Wh_f = nn.Linear(h_size,h_size,bias=False)
        self.Wh_i = nn.Linear(h_size,h_size,bias=False)
        self.Wh_o = nn.Linear(h_size,h_size,bias=False)
        self.Wh_c = nn.Linear(h_size,h_size,bias=False)
        #define y_hat matrix
        self.Wy = nn.Linear(h_size,out_size)

    def forward(self,x):
        h = self.hidden
        i = torch.sigmoid(self.Wx_i(x)+self.Wh_i(h))
        f = torch.sigmoid(self.Wx_f(x)+self.Wh_f(h))
        o = torch.sigmoid(self.Wx_o(x)+self.Wh_o(h))
        c_til = torch.tanh(self.Wx_c(x)+self.Wh_c(h))
        self.cell = f*self.cell + i*c_til
        self.hidden = o*torch.tanh(self.cell)
        y_hat = self.Wy(self.hidden).reshape(1)
        return y_hat

    def train(self,data_lst,epochs = 10,batch_size=32,validation_set =
None):
        '''
            The Data is in the list because data from different days
            need to be treated differently. Specifically the
            hidden state from the previous day probably should not
            be reused.
        '''
        criterion = nn.MSELoss()
        optimizer = torch.optim.Adam(self.parameters())
        loader_lst = [DataLoader(customDataset(data)) for data in data_lst]
        if validation_set != None:
            test_loader = DataLoader(customDataset(validation_set))
            self.hidden = torch.zeros(self.h_size)
            self.cell = torch.zeros(self.h_size)
            val_loss = 0
            for d,t in test_loader:
                y_hat = self.forward(d)
                val_loss += criterion(y_hat,t).data
            print('initial validation loss ',val_loss)
        for epoch in range(epochs):
            total_loss = 0
            np.random.shuffle(loader_lst)
            for data_loader in loader_lst:
                loss = 0
                i = 1
                self.hidden = torch.zeros(self.h_size)
                self.cell = torch.zeros(self.h_size)
                for d,t in data_loader:
                    if i%batch_size == 0:
                        loss.backward(retain_graph=True)
                        optimizer.step()
                        optimizer.zero_grad()
                        total_loss += loss.data
                        loss = 0
                    y_hat = self.forward(d)
                    loss += criterion(y_hat,t)
                    i+= 1
                loss.backward(retain_graph=True)
                optimizer.step()
                optimizer.zero_grad()
            if validation_set != None:
                self.hidden = torch.zeros(self.h_size)
                self.cell = torch.zeros(self.h_size)
                val_loss = 0
                for d,t in test_loader:
                    y_hat = self.forward(d)
                    val_loss += criterion(y_hat,t).data
                print('epoch {} train loss {} validation loss
{}'.format(epoch,total_loss,val_loss))

            else:
                print('epoch {} train loss {}'.format(epoch,total_loss))
```

Lstm classifier:

```python
class LSTM_classifier(nn.Module):
    '''
    LSTM that tries to classify whether the next
    stock price will be above, below,or the same as the current price.


    '''

    def __init__(self,input_size,h_size,out_size=2):

        super(LSTM_classifier, self).__init__()

        #size of hidden state
        self.input_size = input_size
        self.h_size = h_size
        #initialize hidden state.
        self.hidden = torch.zeros(h_size)
        self.cell = torch.zeros(h_size)
        #define x matrices
        self.Wx_f = nn.Linear(input_size,h_size)
        self.Wx_i = nn.Linear(input_size,h_size)
        self.Wx_o = nn.Linear(input_size,h_size)
        self.Wx_c = nn.Linear(input_size,h_size)
        #define h matrices note that we dont need another bias
        self.Wh_f = nn.Linear(h_size,h_size,bias=False)
        self.Wh_i = nn.Linear(h_size,h_size,bias=False)
        self.Wh_o = nn.Linear(h_size,h_size,bias=False)
        self.Wh_c = nn.Linear(h_size,h_size,bias=False)
        #define y_hat matrix
        self.Wy = nn.Linear(h_size,out_size)

    def forward(self,x):
        h = self.hidden
        i = torch.sigmoid(self.Wx_i(x)+self.Wh_i(h))
        f = torch.sigmoid(self.Wx_f(x)+self.Wh_f(h))
        o = torch.sigmoid(self.Wx_o(x)+self.Wh_o(h))
        c_til = torch.tanh(self.Wx_c(x)+self.Wh_c(h))
        self.cell = f*self.cell + i*c_til
        self.hidden = o*torch.tanh(self.cell)
        y_hat = self.Wy(self.hidden)
        return y_hat
    def train(self,data_lst,epochs = 10,batch_size=32):
        '''
            The Data is in the list because data from different days
            need to be treated differently. Specifically the
            hidden state from the previous day probably should not
            be reused.
        '''
        criterion = nn.CrossEntropyLoss()
        optimizer = torch.optim.Adam(self.parameters())
        loader_lst = [DataLoader(customDataset(data)) for data in data_lst]
        for epoch in range(epochs):
            total_loss = 0
            np.random.shuffle(loader_lst)
            for data_loader in loader_lst:
                loss = 0
                i = 1
                self.hidden = torch.zeros(self.h_size)
                self.cell = torch.zeros(self.h_size)
                for d,t in data_loader:
                    if i%batch_size == 0:
                        loss.backward(retain_graph=True)
                        optimizer.step()
                        optimizer.zero_grad()
                        total_loss += loss.data
                        loss = 0
                    y_hat = self.forward(d)
                    loss += criterion(y_hat,t)
                    i+= 1
                loss.backward(retain_graph=True)
                optimizer.step()
                optimizer.zero_grad()
            print('epoch {} loss {}'.format(epoch,total_loss))
```

Trading class that forecasts and plots in real time:
Code example 2:

```python
class trader:

    def __init__(self,model,stock,look_back):
        model.hidden = torch.zeros(model.h_size)
        model.cell = torch.zeros(model.h_size)
        self.model = model
        self.stock = stock
        self.look_back = look_back
        self.forcasts = []
        self.actual = []
        self.money = 1e4
        self.data = []
        self.shares = 0
        self.last_data = None


    def forcast(self):
        token ='pk_47eece3cb483468bb6728475f75b79ce'
        while True:
            while True:
                try:
                    html =
urllib.request.urlopen("https://cloud.iexapis.com/stable/tops?token="+token
+"&symbols="+self.stock)
                    lst = json.loads(html.read().decode('utf-8'))
                    if lst[0] != self.last_data:
                        self.last_data = lst[0]
                        break
                except:
                    pass
            d = lst[0]
            self.data.append(d['lastSalePrice'])

            if len(self.data)>look_back:
                if len(self.data)>look_back+1:
                    self.actual.append(self.data[-1])
                    avg = np.mean(self.data)
                    s_tot = sum([(self.actual[-1 - i]-self.data[-2-i])**2
for i in range(len(self.actual))])
                    s_reg = sum([(self.actual[i]-self.forcasts[i])**2 for i
in range(len(self.forcasts))])
                    self.R2 = 1 - s_reg/s_tot
                avg = np.mean(self.data)
                features = [self.data[-1]-self.data[-1-num] for num in
range(1,look_back+1)]
                features.append(self.data[-1]-avg)
                x = torch.Tensor(features)
                prediction = self.model.forward(x)
                self.forcasts.append(self.data[-1]+prediction)
                xx = [n for n in range(len(self.forcasts))]
                plt.figure('1',figsize=(14,7))
                plt.plot(xx,self.forcasts,label='Forcasts')
                if len(self.actual)>0:
                    plt.scatter(xx[:len(self.actual)],self.actual,label='R2
= {}'.format(self.R2))
                plt.legend()
                plt.grid(color='grey')
                plt.show()
            sleep(60)
```

Resistance and Support Code

Supporting Libraries:

```
import matplotlib.pyplot as plt
from yfinance.ticker import Ticker
import sys
import json
import urllib.request
import numpy as np
from time import sleep
import time
import pandas as pd
```

Plot single graph of boundaries:
Code Example 3:

```
def plot(n,delta,df,nbrh=5,key='price',thresh=.1):

    n1= n
    n2=n + delta
    n3=n + delta + 50

    series = df[key].values[n1:n2]

    minima,maxima = find_extrema(series,nbrhood = nbrh)
    y = [series[i] for i in minima]
    X = np.ones((len(minima),2))
    X[:,1] = minima

    beta = np.linalg.solve(X.T@X,X.T@y)

    xx = np.array([n for n in range(1,n3-n1)])
    yy = beta[0] + beta[1]*xx


    y = [series[i] for i in maxima]
    X = np.ones((len(maxima),2))
    X[:,1] = maxima

    beta_max = np.linalg.solve(X.T@X,X.T@y)

    xx = np.array([n for n in range(1,n3-n1)])
    yy_max = beta_max[0] + beta_max[1]*xx



    plt.figure('1',figsize = (16,8))
    plt.title('Support, Resistance,  and Decision Boundarys')
    plt.plot(series,color='black')
    plt.plot(xx,yy_max,color='green',label = 'Resistance')
    plt.plot(xx,yy,color='red',label = 'Support')


    plt.fill_between(xx,(yy+thresh*yy_max)/(1+thresh),(1-.01)*yy,alpha =
.2,color='red',label = 'Buy')

plt.fill_between(xx,(yy_max+thresh*yy)/(1+thresh),max(max(df[key].values[n2
:n3]),max(series),max(yy_max)),color='green',alpha=.2)

plt.fill_between(xx,min(min(yy),min(df[key].values[n2:n3]),min(series)),(1-
.01)*yy,color='green',alpha=.2,label='Sell')

plt.fill_between(xx,(yy+thresh*yy_max)/(1+thresh),(yy_max+thresh*yy)/(1+thr
esh),alpha = .2,color='yellow',label='Hold')
```

```
    plt.plot([n for n in
range(n2-n1,n3-n1)],df[key].values[n2:n3],color='blue',label = 'Future
Prices')
    plt.scatter(minima,[series[i] for i in minima],color = 'red')
    plt.scatter(maxima,[series[i] for i in maxima],color = 'green')
    plt.grid()
    plt.legend()
    plt.xlabel('Time Steps (days)')
    plt.ylabel('Stock Price')
    plt.savefig('fig.png',dpi=600)
    plt.show()
```

Several Subplots of Boundaries:
Code example 4:

```
def plot(n,delta,dfs,nbrh=5,key='Close',thresh=.15):

    n1= n
    n2=n + delta
    n3=n + delta + 50
    ndf = len(dfs)
    plt.figure(figsize=(18,9))
    own_dict = {'aapl':[10,100],'rio':[15,50]}#stock:[number of stocks,
bought price]
    for i,df in enumerate(dfs):
        series = df[key].values[n1:n2]
        minima,maxima = find_extrema(series,nbrhood = nbrh)

        y = [series[m] for m in minima]
        X = np.ones((len(minima),2))
        X[:,1] = minima
        beta = np.linalg.solve(X.T@X,X.T@y)

        xx = np.array([n for n in range(1,n3-n1)])
        yy = beta[0] + beta[1]*xx


        y = [series[m] for m in maxima]
        X = np.ones((len(maxima),2))
        X[:,1] = maxima

        beta_max = np.linalg.solve(X.T@X,X.T@y)

        xx = np.array([n for n in range(1,n3-n1)])
        yy_max = beta_max[0] + beta_max[1]*xx



        ax = plt.subplot(2,4,i+1)
        plt.plot(series,color='black',axes = ax )
        plt.plot(xx,yy_max,color='green',label = 'Resistance',axes = ax)
        plt.plot(xx,yy,color='red',label = 'Support',axes = ax)


        plt.fill_between(xx,(yy+thresh*yy_max)/(1+thresh),(1-.02)*yy,alpha
= .2,color='red',label = 'Buy',axes = ax)

plt.fill_between(xx,(yy_max+thresh*yy)/(1+thresh),max(max(df[key].values[n2
:n3]),max(series),max(yy_max)),color='green',alpha=.2,axes = ax)

plt.fill_between(xx,min(min(yy),min(df[key].values[n2:n3]),min(series)),(1-
.02)*yy,color='green',alpha=.2,label='Sell',axes = ax)

plt.fill_between(xx,(yy+thresh*yy_max)/(1+thresh),(yy_max+thresh*yy)/(1+thr
esh),alpha = .2,color='yellow',label='Hold',axes = ax)


        plt.plot([n for n in
range(n2-n1,n3-n1)],df[key].values[n2:n3],color='blue',label = 'Future
Prices',axes = ax)
        plt.scatter(minima,[series[i] for i in minima],color = 'red',axes =
ax)
        plt.scatter(maxima,[series[i] for i in maxima],color = 'green',axes
= ax)
        ax.grid()
        ax.set_title(stocks[i])
        if i == 0:
            ax.legend()
        if i == 4:
            plt.xlabel('Time Steps (days)',axes = ax)
        if i == 0:
            plt.ylabel('Stock Price',axes = ax)
    x = 1
    x = 1 + x
    x = 4+4

    ax = plt.subplot(2,4,i+2)
    ax.set_xticks([])
    ax.set_yticks([])
    s = 'Summary \n__________________________________\n'
    for stock in own_dict:
        num_stocks,bought_price = own_dict[stock]
        s += stock.upper() + ': '+str(num_stocks)+ ' stocks bought at $' +
str(bought_price)+'\n'
    ax.set_axis_off()

    ax.text(0,.7,s,fontsize=14)

    plt.savefig('fig.png',dpi=600)
    plt.show()


stocks = ['aapl','ngd','googl','krmd','drd','rio','msft']
tickers = [Ticker(stock) for stock in stocks]
dfs = [ticker.history(period='12mo') for ticker in tickers]
plot(10,100,dfs)
```

Plot and trade in real time:
Code example 5:

```
hyper_parameters = {

    'thresh':.15,
    'nbrh':5,
    'delta':100

    }
token ='pk_47eece3cb483468bb6728475f75b79ce'
stock_str ='aapl,ngd,googl,krmd,drd,rio'
def trade(stock_str,hyper_parameters,auto_trade = True):

    stock_lst = stock_str.split(',')
    data_dict = {stock:[] for stock in stock_lst}
    own_dict = {}
    thresh = hyper_parameters['thresh']
    nbrh = hyper_parameters['nbrh']
    delta = hyper_parameters['delta']
    money = 1e4
    buy = 1e3
    key = 'lastSalePrice'
    last_data_point = None #check for dubplicates
    n = 0
    while True:
        #make request
        html =
urllib.request.urlopen("https://cloud.iexapis.com/stable/tops?token="+token
+"&symbols="+stock_str)
        #list of dictionarys
        lst = json.loads(html.read().decode('utf-8'))
        for l in lst:
            data_dict[l['symbol'].lower()].append(l[key])
        n += 1
        if n > delta:
            dfs =
[pd.DataFrame({'lastSalePrice':data_dict[stock[-delta:]]}) for stock in
stock_lst]
            n1= 0
            n2=n + delta
            ndf = len(dfs)
            plt.figure(figsize=(18,9))
            own_dict = {'aapl':[10,100],'rio':[15,50]}#stock:[number of
stocks, bought price]
            for i,df in enumerate(dfs):
                series = df[key].values[n1:n2]
                minima,maxima = find_extrema(series,nbrhood = nbrh)
                y = [series[m] for m in minima]
                X = np.ones((len(minima),2))
                X[:,1] = minima
                beta = np.linalg.solve(X.T@X,X.T@y)
                xx = np.array([n for n in range(1,150)])
                yy = beta[0] + beta[1]*xx

                y = [series[m] for m in maxima]
                X = np.ones((len(maxima),2))
                X[:,1] = maxima
                beta_max = np.linalg.solve(X.T@X,X.T@y)

                xx = np.array([n for n in range(1,150)])
                yy_max = beta_max[0] + beta_max[1]*xx



                ax = plt.subplot(2,4,i+1)
                plt.plot(series,color='black',axes = ax )
                plt.plot(xx,yy_max,color='green',label = 'Resistance',axes
= ax)
                plt.plot(xx,yy,color='red',label = 'Support',axes = ax)



plt.fill_between(xx,(yy+thresh*yy_max)/(1+thresh),(1-.02)*yy,alpha =
.2,color='red',label = 'Buy',axes = ax)

plt.fill_between(xx,(yy_max+thresh*yy)/(1+thresh),max(max(df[key].values),m
ax(series),max(yy_max)),color='green',alpha=.2,axes = ax)

plt.fill_between(xx,min(min(yy),min(df[key].values),min(series)),(1-.02)*yy
,color='green',alpha=.2,label='Sell',axes = ax)

plt.fill_between(xx,(yy+thresh*yy_max)/(1+thresh),(yy_max+thresh*yy)/(1+thr
esh),alpha = .2,color='yellow',label='Hold',axes = ax)


                plt.scatter(minima,[series[i] for i in minima],color =
'red',axes = ax)
                plt.scatter(maxima,[series[i] for i in maxima],color =
'green',axes = ax)
                ax.grid()
                if i == 0:
                    ax.legend()
                if i == 4:
                    plt.xlabel('Time Steps (days)',axes = ax)
                if i == 0:
                    plt.ylabel('Stock Price',axes = ax)

                #calculate support val
                support = beta[0] + beta[1]*delta
                #calculate resistance val
                resistance = beta_max[0] + beta_max[1]*delta
                #apply decision rule
                curr_price = series[-1]

                if abs(support - curr_price)/abs(resistance -
curr_price)<=thresh:
                    if curr_price >= (1-.02)*support:
                        decision = 'buy'
                    else:
                        decision = 'cell'
                elif abs(resistance - curr_price)/abs(support -
curr_price)<=thresh:
                    decision = 'sell'
                elif curr_price >= resistance:
                    decision = 'sell'
                else:
                    decision = 'hold'

                if decision == 'buy':
                    if stock_lst[i] not in own_dict and money > 0:
                        invest = max([1000,money])
                        money -= invest
                        own_dict[stock_lst[i]] =
[invest/curr_price,curr_price]
                if decision == 'sell':
                    stock = stock_lst[i]
                    money += own_dict[stock][0]*curr_price
                    del own_dict[stock]


                ax.set_title(stock_lst[i] + ' Recommendation: '+decision)

            ax = plt.subplot(2,4,i+2)
            ax.set_xticks([])
            ax.set_yticks([])
            s = 'Summary \n__________________________________\n'
            for stock in own_dict:
                num_stocks,bought_price = own_dict[stock]
                s += stock.upper() + ': '+str(num_stocks)+ ' stocks bought
at $' + str(bought_price)+'\n' + 'curr price: $' + data_dict[stock][-1]
```

```
            ax.set_axis_off()

            ax.text(0,.7,s,fontsize=14)

            plt.savefig('fig.png',dpi=600)
            plt.show()

trade(stock_str,hyper_parameters)
```

Simulation code:
Code example 6:

```
def simulate(df,delta,threshold,threshold_m,nbrh,key='price'):
    total_money = 1e5
    buy = 1000
    shares = 0
    owns = False
    total_buys = 0


    for n1 in range(len(df)-delta):
        n2 = n1 + delta
        series = df[key].values[n1:n2]
        minima,maxima = find_extrema(series,nbrhood = nbrh)
        y = [series[i] for i in minima]
        X = np.ones((len(minima),2))
        X[:,1] = minima
        beta = np.linalg.solve(X.T@X,X.T@y)
        support_val = beta[0] + beta[1]*(n2-n1)
        y = [series[i] for i in maxima]
        X = np.ones((len(maxima),2))
        X[:,1] = maxima
        beta_max = np.linalg.solve(X.T@X,X.T@y)
        resistance_val = beta_max[0] + beta_max[1]*(n2-n1)

        ratio = abs((series[-1]-support_val)/(series[-1]-resistance_val))

        if series[-1]<=1.01*resistance_val and series[-1]>=
(1-.01)*support_val:
            in_bounds = True
        else:
            in_bounds = False
        if (ratio <= threshold and owns == False and in_bounds) or (owns ==
False and in_bounds and series[-1]<support_val):
            total_money -= buy
            shares = buy/series[-1]
            owns = True
            total_buys += 1


        if 1/ratio <= threshold_m and owns == True:
            total_money += shares*series[-1]
            shares = 0
            owns = False
        if in_bounds == False and owns:
            total_money += shares*series[-1]
            shares = 0
            owns = False
    if owns:
        total_money += shares*series[-1]
    return total_money-1e5,total_buys
```

```
stocks =  ['aapl','msft','googl','ge','hp','tsla']
total = 0
for stock in stocks:
    stck = Ticker(stock)
    df = stck.history(period='12mo')
    df = df[:-60]
    #plot(100,100,df,4,key='Close')
    profit,_ = simulate(df,100,.1,.1,4,key='Close')
    print(profit,stock)
    total += profit
print(total)
```

Helper Function:

```
def find_extrema(series,nbrhood=10):
    '''
    Return index of local minima in data frame.
    A point is a local minima if it is the
    smallest point in range +- nbrhood.
    '''
```

```python
    minima = []
    maxima = []
    for i in range(nbrhood,len(series)-nbrhood):
        canidate_minima = series[i]
        neihborhood = series[i-nbrhood:i+nbrhood]
        if sum(neihborhood < canidate_minima) == 0:
            minima.append(i)
        if sum(neihborhood > canidate_minima) == 0:
            maxima.append(i)
    return minima,maxima
```