Simple Stock Price Prediction with ML in Python—Learner's Guide to ML

Alec Cunningham Follow
Nov 9 · 10 min read

Introduction

One of the most prominent use cases of machine learning is "Fintech" (Financial Technology for those who aren't buzz-word aficionados); a large subset of which is in the stock market. Financial theorists, and data scientists for the better part of the last 50 years, have been employed to make sense of the marketplace in order to increase return on investment. However, due to the multidimensional nature of the problem, the scale of the system, and inherent variation with time, it

has been an overwhelmingly tough challenge for humans to solve, even with the assistance of conventional data analytics tools. However, with the onset of recent advancements in machine learning applications, the field has been evolving to utilize non-deterministic solutions the "learn" what is going on in order to make more accurate predictions.

In this article I will demonstrate a simple stock price prediction model and exploring how "tuning" the model affects the results. This article is intended to be easy to follow, as it is an introduction, so more advanced readers may need to bear with me.

Step 1: Choosing the data

One of the most important steps in machine learning and predictive modeling is gathering good data, performing the appropriate cleaning steps and realizing the limitations.

For this example I will be using stock price data from a single stock, Zimmer Biomet (ticker: ZBH). Simply go too finance.yahoo.com, search for the desired ticker. Once you are on the home page of the desired stock, simple navigate to the "Historical Data" tab, input the range of dates you would like to include, and select "Download Data." I chose 5 years, but you can choose as far back as you would like.

Now that we have out data, let's go ahead and see what we have. Simply open the file in Excel.

	1	2	3	4	5	6	7
1	Date	Open	High	Low	Close	Adj Close	Volume
2	11/11/13	88.519997	89.559998	88.410004	89.230003	85.394432	1151800
3	11/12/13	88.779999	89.309998	88.349998	89.25	85.413544	1102700
4	11/13/13	89.07	89.650002	88.519997	89.639999	85.786797	743000
5	11/14/13	89.470001	90.75	89.400002	90.300003	86.418427	1068800

Looks like we have some goodies here. You may notice that all of the fields are numerical values, except that pesky date value… We need to fix this. The values that we are going to pass into our model need to be in a format that can be most easily understood. So, we need to perform some "data preprocessing" steps. In our case we are going to insert a new column after 1, name it "Date Value," and copy all of the dates from column 1 into column 2. Then select all of the data and change

the type from "Date" to "Text." The results should look like the following:

	1	2	3	4	5	6	7	8
1	Date	Date Value	Open	High	Low	Close	Adj Close	Volume
2	11/11/13	41589	88.519997	89.559998	88.410004	89.230003	85.394432	1151800
3	11/12/13	41590	88.779999	89.309998	88.349998	89.25	85.413544	1102700
4	11/13/13	41591	89.07	89.650002	88.519997	89.639999	85.786797	743000
5	11/14/13	41592	89.470001	90.75	89.400002	90.300003	86.418427	1068800

Ok, so now save the file as "choose_a_name.csv" (make sure it is a ".csv "and not one of the excel default formats).

Before we start, lets talk about limitations. You will notice that the only data we feed this model is date and price. There are many external factors that affect the price outside of the historical price. Highly robust models might utilize external data such as news, time of the year, social media sentiment, weather, price of competitors, market volatility, market indices, etc. This model is very basic, but in time you can learn the skills to build a model that is more "aware" of the overall marketplace. That being said, let's move one.

Step 2: Choosing the model

So now that we have data cleaned up, we need to choose a model. In this case we are going to use a neural network to perform a regression function. A regression will spit out a numerical value on a continuous scale, a apposed to a model that may be used for classification efforts, which would yield a categorical output. In this situation, we are trying to predict the price of a stock on any given day (and if you are trying to make money, a day that hasn't happened yet).

To build our model we are going to use TensorFlow… well, a simplified module called TFANN which stands for "TensorFlow Artificial Neural Network." In order to do this, we are going to use Google Colab. If you are not familiar with Colab, simply navigate to colab.research.google.com, it is a *free* virtual python notebook environment. (For those of you that will be following along and don't know what you are doing, just copy paste the code below into a "cell" and then hit run before creating a new one and copying more code).

Step 3: Building the Model

First we need to install TFANN. Open a new Colab notebook (python 3). Colab has numerous libraries which can be accessed without installation; however, TFANN is not one of them so we need to execute the following command:

```
!pip install TFANN
```

Now let's import our dependencies:

```
import numpy as np
import matplotlib.pyplot as mpl
from sklearn.preprocessing import scale
from TFANN import ANNR
from google.colab import files
```

NumPy will be used for our matrix operations, Matplotlib for graphs, sykit-learn for data processing, TFANN for the ML goodness, and google.colab files will help us upload data from the local machine to the virtual environment.

Now we need to import the data that we have already processed. To do this we will execute the following command, which will provide us with a window to upload the .csv file.

```
files.upload()
```

Easy, right?

You now have a virtual folder that contains this file. Execute the following command if you don't believe me, it will print the names of the files in the current directory.

```
!ls
```

Now we can finally get to the meat of this project. Execute the following commands:

```
#reads data from the file and ceates a matrix with only the
dates and the prices
stock_data = np.loadtxt('ZBH_5y.csv', delimiter=",",
skiprows=1, usecols=(1, 4))
#scales the data to smaller values
stock_data=scale(stock_data)
#gets the price and dates from the matrix
prices = stock_data[:, 1].reshape(-1, 1)
dates = stock_data[:, 0].reshape(-1, 1)
#creates a plot of the data and then displays it
mpl.plot(dates[:, 0], prices[:, 0])
mpl.show()
```

You should get a nice graph that looks like this:

Note, that the scale is no longer in dollars on the y-axis and those arbitrary integer-date values on the x-axis. We have scaled the data down to make the learning process more effective. Try writing some code to return the scale of the y-axis back to dollars and the x-axis to years!

Now, we need to construct the model. In this case we will use *one* input and output neuron (input date, output price) and will have *three* hidden layers of 25 neurons each. Each layer will have an "tanh"

activation function. If you do not understand these concepts, feel free to google it and come back, understanding the basics for neural network principals will be very helpful as you progress.

```
#Number of neurons in the input, output, and hidden layers
input = 1
output = 1
hidden = 50
#array of layers, 3 hidden and 1 output, along with the tanh
activation function
layers = [('F', hidden), ('AF', 'tanh'), ('F', hidden),
('AF', 'tanh'), ('F', hidden), ('AF', 'tanh'), ('F',
output)]
#construct the model and dictate params
mlpr = ANNR([input], layers, batchSize = 256, maxIter =
20000, tol = 0.2, reg = 1e-4, verbose = True)
```

We have now initialized the model and are ready to train!

Step 4: Training the Model

```
#number of days for the hold-out period used to access
progress
holdDays = 5
totalDays = len(dates)
#fit the model to the data "Learning"
mlpr.fit(dates[0:(totalDays-holdDays)], prices[0:(totalDays-
holdDays)])
```

Once the training is complete, we can execute the following commands to see how we did.

```
#Predict the stock price using the model
pricePredict = mlpr.predict(dates)
#Display the predicted reuslts agains the actual data
mpl.plot(dates, prices)
mpl.plot(dates, pricePredict, c='#5aa9ab')
mpl.show()
```

Not too bad! But we can do better.

Let's think about some ways in which we can increase the fidelity of the model. We can think of about this as "what knobs can we turn" to tune our model. Well the first is to simply decrease the error tolerance.

The first trial, the error tolerance was set as .2; however, we can lower this to a smaller number, say .1, lets give that a try!

Simply make the following changes. Note that I am also updating the name of the variables so that the values we already created/observed do not change. Certainly not the most effective method here, but I am sure you can create a better one!

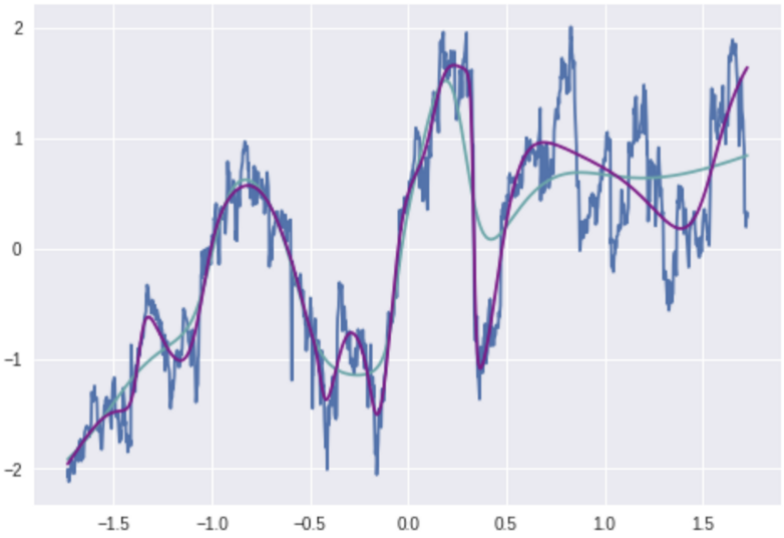
```
#Number of neurons in the input, output, and hidden layers
input2 = 1
output2 = 1
hidden2 = 50
#array of layers, 3 hidden and 1 output, along with the tanh
activation function
layers = [('F', hidden2), ('AF', 'tanh'), ('F', hidden2),
('AF', 'tanh'), ('F', hidden2), ('AF', 'tanh'), ('F',
output2)]
#construct the model and dictate params
mlpr2 = ANNR([input2], layers, batchSize = 256, maxIter =
10000, tol = 0.1, reg = 1e-4, verbose = True)
```

Run the model again with the following commands and we get new results:

```
holdDays = 5
totalDays = len(dates)
mlpr2.fit(dates[0:(totalDays-holdDays)], prices[0:
(totalDays-holdDays)])
```

Once it has finished training:

```
pricePredict2 = mlpr2.predict(dates)
mpl.plot(dates, prices)
mpl.plot(dates, pricePredict, c='#5aa9ab')
mpl.plot(dates, pricePredict2, c='#8B008B')
mpl.show()
```



Looking better! As you can see, lovering the error tolerance… well… lowered the error. So you might be wondering "why not just set the error to a really small number?" and that would be a great question. Go ahead and try it for yourself, re-execute the code you just ran with the tolerance set at .05. What you will observe is that the maximum number of iterations you use will stop the execution before it reached the desired level of error. Ok then, why not just increase the maximum

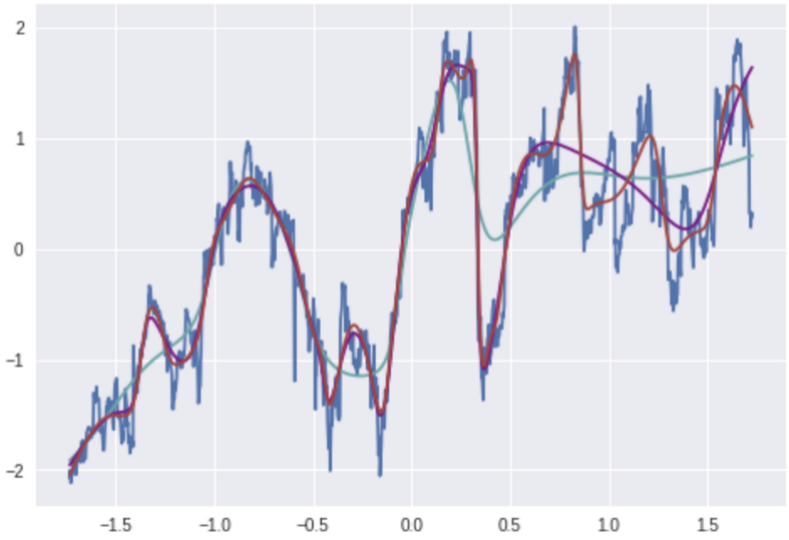
number of iterations? Well, the problem lies in the given model parameters. The model itself has limitations, the lowest achievable error for the model we constructed may only be .8 (I have not checked exactly for this for this model). In this situation, it does not matter how many more iterations you add, the structure of the model will not yield better results, no matter how many iterations are run. It is simply capped out.

The next logical question to ask here would be "how can we change the model to achieve greater error?" and that is what we are going to explore!

Models have what are known as "hyperparameters." These are the parameters that govern the model, they define how the model is created. Altering these can give us better (or perhaps worse) results. Examples include: number of neurons in each hidden layer, the number of hidden layers, the activation function, etc.

Our goal here is to "tune" these hyperparameters to achieve a lower error tolerance than was possible with our first model. The simplest way to do this, in my opinion, is do increase the number of neurons in the hidden layers. I am by no means a leading source of knowledge on this topic, but I will venture far enough to say that increasing the number of neurons and/or the number of hidden layers increases the level of abstraction with which the model can represent the given data. So lets give that a try!

Increasing the number of neurons in each hidden layer from 50 to 100 and setting the tolerance to .075:



Much much better! The orange line is out newest prediction. Notice how much better it tracks the more recent prices than the last model did.

I think we have created a good model, and I am satisfied with the results! But this project can be continued to learn more about hyperparameters. Try changing the activation function to something besides "tanh", or perhaps adding an additional layer.

To add another layer, reference this line of code:

```
layers = [('F', hidden), ('AF', 'tanh'), ('F', hidden),
('AF', 'tanh'), ('F', hidden), ('AF', 'tanh'), ('F',
output)]
```

Add one additional layer by adding another ('AF', hidden), ('AF', 'tanh') before the output node. This adds the layer and the activation function applied to it before it is fed into the next layer.

```
layers = [('F', hidden), ('AF', 'tanh'), ('F', hidden),
('AF', 'tanh'), ('F', hidden), ('AF', 'tanh'),('F', hidden),
('AF', 'tanh'), ('F', output)]
```

Or perhaps you want a different number of neurons at each hidden layer, tapering them down is a common method. The example below tapers from 100 down to 25 nodes before the output:

```
layers = [('F', 100), ('AF', 'tanh'), ('F', 50), ('AF',
'tanh'), ('F', 25), ('AF', 'tanh'), ('F', output)]
```

So, there you have it! An easy introduction to machine learning and neural networks that you can do at home for free in about an hour!

I would like to recognize Nicholas T. Smith, whose model influenced the creation of this post.

Finally, if you have any questions, comments, suggestions, or concerns, feel free to reach out!