Machine Learning Engineer Nanodegree

Capstone Project

Pinzhi Chen

Feb 21st, 2020

I. Definition

Project Overview

Investment firms, hedge funds and even individuals have been using financial models to better understand market behavior and make profitable investments and trades. Machine learning engineers and fund managers are therefore working together to find financials model that can properly describe the behaviors of the stock markets.

However, statistical analysts once argued and (some of the researchers) proved empirically that stock returns (especially in U.S. Equity markets) are weak-form efficient. Therefore, historical prices bear no information on future stock price movements. They argued that stock prices are unpredictable martingale processes.

Recent rises in big data analysis, machine learning and deep learning have brought us to new horizons. With proper usage of these algorithms, we can probably be able to find reasonable models that can indicate the movement of stock prices.

Problem Statement

- Use machine learning / deep learning models to build the stock returns against a list of explanatory variables, each of them properly normalized.
- Analyze the stock price (return) predictability among the provided factors.
- Deploy the model to website or any platform through Amazon Sagemaker, such that users can input a stock symbol and get the prediction of movements tomorrow.

Metrics

I will not simply model the prices of the stocks because it has been observed that stock prices are empirically not stationary process and linear regression on the prices against explanatory variables are likely spurious. The machine learning models associated with these problems are likely to be spurious as well.

A more reasonable approach is to predict the stock returns instead of the prices. Stock returns are empirically proved to be stationary processes and therefore reasonable

linear regression can be modeled. Complex machine learning and deep learning models can also be implemented. An extra benefit of stock return is stock returns are normalized to some number close to zero, while stock prices have different scales for each stock throughout the times.

Throughout the analysis, I will use MSE as the loss function to train the returns of stock prices.

II. Analysis

Data Exploration

I collect the stocks and economics related data, from 1/1/2000 to 12/31/2019. All data are in the daily level, so it contains about 5000 rows of data.

The following are the sources of the datasets:

- Yahoo Finance: Web API are used to get the stock prices.
- FRED: Web API are used to get the economic data. For example, interest rates, dollar index, etc.
- Professor Kenneth R. French's Website: Fama-French factor models used Fama-French factors. The up-to-date factors are freely available in the website.

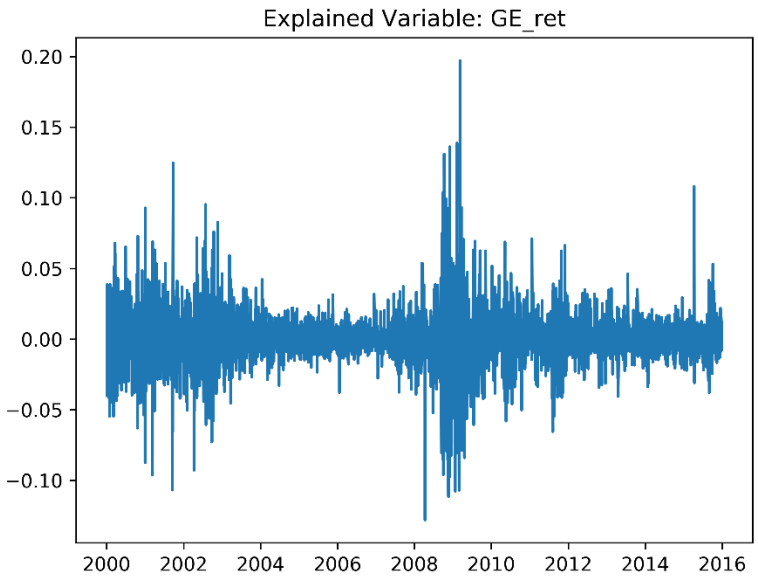
Especially, I used incorporated python APIs from pandas data reader to get the datasets. To install this, please type the following commands in the command line (I used anaconda to install this):

```
conda install -c anaconda pandas-datareader
```

Exploratory Visualization

The Explained variable that we want to forecast (suppose this is the stock returns of the GE equity) is in the following graph. As can be shown, the stock returns of GE are stationary and a random walk model might be the best linear predictor that forecast its future movements. So, our machine learning model will still try to answer the question: is the random walk the best model amongst all the possible linear and non-linear models?

Figure I. Visualization of Explained Variable



Algorithms and Techniques

I will use a simple neural network model to train the RMSE loss of the predictor that forecasting the stock returns of GE stock. I find the results of this simple neural network performs to be no better than the random walk, so I want to evaluate whether some series models, e.g. deep AR RNN or LSTM model can be working.

My steps are:

- Data preprocessing and normalization.
- Train and evaluate a simple neural network model.
- Train and evaluate a LSTM deep learning model.
- Explain potential problems and possible future steps.

Benchmark

The *martingale* model, or so-called *random walk* that predicts the future returns are always 0, with possibility of going up to be 0.5, and possibility of going down to be 0.5 is the most simple, but sometimes ridiculously, most powerful model in the linear form, sometimes possibly in both linear and non-linear form, when it comes to forecasting the future stock returns.

It's very likely that both the simple and the complicated models are worse than the benchmark model. This is because financial data are very noisy and generally it cannot be repeatable. Unlike natural experiments. Many researches are focusing on this area but up-to-now there is no best solutions.

III. Methodology

Data Preprocessing

The stock prices are what I want to forecast. For these datasets, we can get daily stock values such as Open, High, Low, Close, Volume and Adjusted Close. The returns of Adjusted Close are what I am trying to predict. For other two auxiliary dataset, the FRED dataset is all about macroeconomic data, and the Fama-French factor data are all about the market factors that are well-believed to picture the risk premium of individual returns.

First, I will briefly describe (most of) the columns in my dataset, and explain how I normalize each of them:

- Normalize 12-Month London Interbank Offered Rate (LIBOR), based on U.S. Dollar (USD12MD156N): divide by 100 to get the real value (percentage)
- Normalize Trade Weighted U.S. Dollar Index: Major Currencies, Goods; Index Mar 1973=100, Not Seasonally Adjusted (DTWEXM): DTWEXM (new) = (DTWEXM - 100) / 100: 100 is like the benchmark index
- Normalize Moody's Seasoned Aaa Corporate Bond Yield, Percent, Not Seasonally Adjusted (DAAA): divide by 100 to get the real value (percentage)
- Normalize VIXCLS (CBOE Volatility Index: VIX; Index, Not Seasonally Adjusted): divide by 100 to get the real value (percentage)
- Normalize The Fama-French Five Factor Model data (Mkt-RF, SMB, HML, RMW and CMA): divide by 100 to get the real value (percentage)

Then, I merge the dataset to get an overall data, fill the N.A. values with the values of the last trading date, and add an extra column as the returns of the next date as the explained variable (y). For the purpose of fair evaluations, I chose to forecast the returns of GE. I chose to forecast its returns because in the entire dataset the mean of GE's returns are closest to 0, so the accuracy-based analysis are the most fair in this explained variable. It's very convenient to switch to other predictors, so there is no loss of generality.

Third, I list a descriptive data analysis on the distributions and the scales of the normalized dataset. As is shown, the standard deviations of all the normalized data are similar in the scales. Machine learning algorithms can be properly trained and evaluated thereafter.

Table I. Descriptive Statistics of Variables

	mean	std	min	25%	50%	75%	max
GE_retNext	0.00%	1.97%	-12.79%	-0.83%	0.00%	0.83%	19.70%
GE_ret	0.00%	1.97%	-12.79%	-0.84%	0.00%	0.83%	19.70%
GE_loglogVol	2.91	0.04	2.80	2.89	2.91	2.94	3.07
GE_retInday	-0.05%	1.66%	-11.17%	-0.78%	-0.03%	0.70%	14.97%
GE_rangeRatio	2.26%	1.76%	0.32%	1.18%	1.78%	2.77%	20.45%

Other stocks				…			
USD12MD156N	2.43%	1.77%	0.53%	1.02%	1.81%	3.23%	7.50%
DTWEXM	-14.02%	10.94%	-31.99%	-23.74%	-14.76%	-7.78%	13.10%
DAAA	5.00%	1.19%	2.81%	3.94%	5.11%	5.67%	8.12%
VIXCLS	19.47%	8.45%	9.14%	13.48%	17.30%	22.90%	80.86%
Mkt-RF	0.03%	1.20%	-8.95%	-0.48%	0.07%	0.58%	11.35%
SMB	0.01%	0.59%	-4.31%	-0.32%	0.02%	0.35%	4.49%
HML	0.01%	0.66%	-4.24%	-0.29%	0.00%	0.28%	4.83%
RMW	0.02%	0.51%	-2.92%	-0.23%	0.01%	0.25%	4.40%
CMA	0.01%	0.42%	-5.94%	-0.19%	0.00%	0.20%	2.41%
RF	0.01%	0.01%	0.00%	0.00%	0.00%	0.01%	0.03%

Implementation

Using the data created from the data preprocessing step, here I am going to build a simple benchmark model (simple neural network) to evaluate the stock return predictability.

The neural network is fairly simple. It inputs the 46 features, performs a linear transformation (from input_dim = 46 to hidden_dim = any number to be specified) to get the hidden layer, feeds to a sigmoid activation function, then I manually convert it to scaled from -1 to 1 to preserve some forms of unbiasedness, and does a Dropout, and finally feeds to the second linear nodes to get the output values (with output_dim = 1).

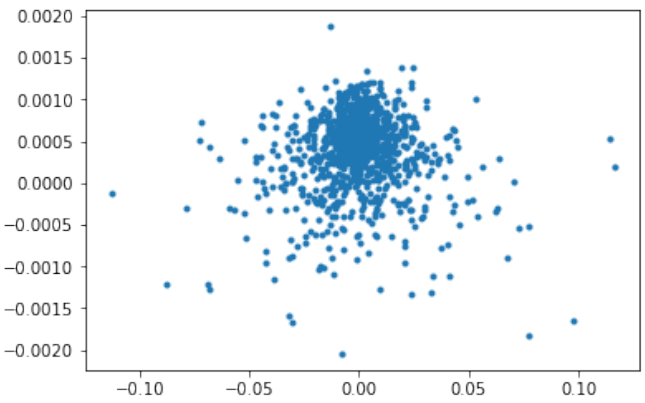
The structure of the code of the simple neural network I trained is:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class SimpleNet(nn.Module):

    def __init__(self, input_dim, hidden_dim, output_dim):
        '''Defines layers of a neural network.
           :param input_dim: Number of input features
           :param hidden_dim: Size of hidden layer(s)
           :param output_dim: Number of outputs
         '''
        super(SimpleNet, self).__init__()
        # defining 2 linear layers
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, output_dim)
        self.drop = nn.Dropout(0.1)
        self.sig = nn.Sigmoid()

    def forward(self, x):
        '''Feedforward behavior of the net.
```

```
        :param x: A batch of input features
        :return: A single, sigmoid activated value
    '''
    out = self.fc1(x)
    out = self.sig(out)
    # convert to from -1 to 1
    out = 2 * out - 1
    out = self.drop(out)
    out = self.fc2(out)
    return out
```

The following figure shows the predicted next returns against actual next returns of GE. If we evaluate whether the direction of the movements are correctly valued, we conclude that the accuracy is actually 50.05%. This is in fact no different from random walk so may further investigations can be applied.

Figure II. Actual Returns (x-axis) against Predicted Returns (y-axis) for GE



The second model I used is a simple LSTM, and it did have improved the performance in some forms. The input parameters of the simple LSTM is the same as the simpleNet I trained before, to preserve generality. The model puts the nn.LSTM layer in the hidden cell, and performs a final linear transform to predict the final outcome from the hidden LSTM layer.

The code of the structure of the model is defined as follows:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

# Here we define our model as a class
class SimpleLSTM(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super().__init__()
        self.hidden_layer_size = hidden_dim
```

```
        self.lstm = nn.LSTM(input_dim, hidden_dim)
        self.linear = nn.Linear(hidden_dim, output_dim)
        self.hidden_cell = (torch.zeros(1, 1, self.hidden_layer_size),
                            torch.zeros(1, 1, self.hidden_layer_size))

    def forward(self, input_seq):
        lstm_out, self.hidden_cell = self.lstm(
              input_seq.view(len(input_seq), 1, -1), self.hidden_cell)
        predictions = self.linear(lstm_out.view(len(input_seq), -1))
        return predictions
```

Refinement

The codes and models that I have provided have already witnessed some forms of refinement that I have not mentioned.

For the simple Net model, I used a dropout layer. The dropout ratio is a hyperparameter that can be tuned. The dropout ratio may not be set to be too high because we do not have too many training data to use. For the optimizer, I used Adam algorithm to fasten gradient descent. The hidden dim is another hyperparameter that can be tuned. Hyperparameter tuning can be probably applied to improve the performance. But I want to switch to the better LSTM model.

For the LSTM model, I did not use a dropout layer. This can be potentially added and the hyperparameter associated with it can be tuned as well. For the optimizer, I used Adam algorithm to fasten gradient descent. The hidden dim is another hyperparameter that can be tuned. Hyperparameter tuning can be probably applied to improve the performance. But the hyperparameter tuning costs too much time.

Cross-folder validation and introducing validation set and be applied to improve the performance. But the model is time-series data and future data are not allowed to use. So traditional k-fold validation is inappropriate. Rather, a rolling-window machine learning estimation can be estimated. This is perhaps a good future step to work on.

IV. Results

Model Evaluation and Validation

By tests that are not listed in the notebook nor here, if we try to forecast the entire test set (forecasting 4 years based on this simple model), then the performance is really poor: the accuracy is only about 48% to 52%, and our model is always predicting negative numbers, so the accuracy actually only depends on how many returns are really negative.

One of the underlying problems, is that LSTM does not learn from the predicted output against the real output aggregately. In that way, the actual returns, essential for the data, did not get integrated with our model. Therefore, it's unlikely that our model still preserves

long-term dependency. The model must learn from the output of each iteration of forecasting.

So, our model may only be useful for short-term forecasting. Here I presented my methods when only forecasting 22 days, which is roughly the trading dates of a month. As we can see, my model predicts negative signs for all the returns in GE, and the actual accuracy is about 60%. This is a sign that my model says that this stock is not a good investment, and therefore it's always predicting negative signs. In fact, the actual returns have 60% of the times being negative.

Figure III. Predicted Returns (y-axis) for GE in LSTM Model

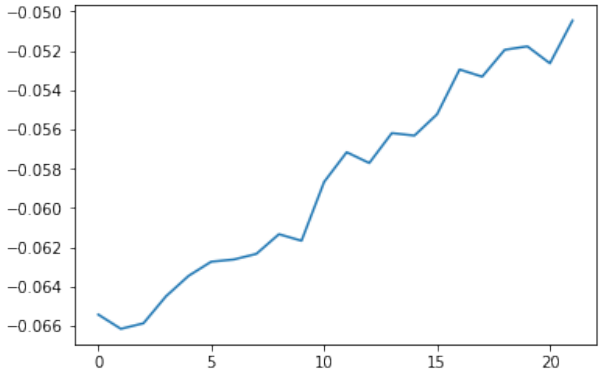
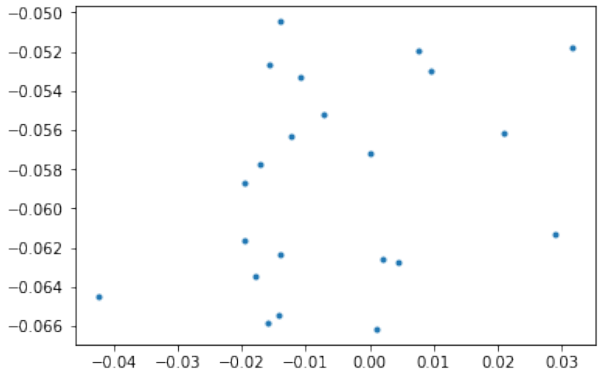


Figure IV. Actual Returns (x-axis) against Predicted Returns (y-axis) for GE in LSTM Model (Horizon: 22 trading days)



It is therefore possible that the performance of my model has enhanced. However, more further steps need to be done for better analysis. For example, more high-frequency and up-to-date data, and better implement of model that can learn real-time, so the model can be useful in predicting long-term and it can keep learning.

Justification

Our benchmark model (martingale) states that the predicted returns should be always 0 and there is actually no point to predict anything. This assumption is still pretty strong because none of any machine learning models that we used could really learn from the

patterns and predict something with accuracy significantly greater than 0.5 after a long period of time. Sequence-based models, like LSTM, can be possibly working for forecasting a short-term horizon. But the LSTM memories might still got lost when it comes to the real long-term memory. So machine learning models must learn real-time, adjusting parameters and give reasonable estimations. More works must be done before a really practical predictor is working.

V. Conclusion

Statistical analysts have argued for a long time and (some of the researchers) proved empirically that stock returns (especially in U.S. Equity markets) are weak-form efficient. Therefore, historical prices bear no information on future stock price movements. They argued that stock prices are unpredictable martingale processes.

Our machine learning models do not have enough evidence to reject this null hypothesis that stock markets are completely unpredictable. For example, a simple net could not beat the random walk at all. For another example, unadjusted LSTM cannot beat the random walk. Other models, e.g. XGBoost, LSTM, or even simple net, with adjustment in real-time might be appliable.

Moreover, more data can be added to improve the performance and justify the hypothesis. For example, real-time news data can be incorporated to study how investors behave in real-time. We may learn that they have been overreacted and can perform strategies to dig into these profits. Many more works can be done to deal with highly noisy financial data. Therefore, sometimes it can be an art instead of a science.