Cryptocurrency Trading Using Linear Regression Models & Feed-Forward Neural Networks

Abe Leite, Dante Razo[1]

Abstract
This project is a cryptocurrency trader that is trained on historic Bitcoin data. There are multiple "players" that use different models to predict when to buy and sell their holdings. The goal is to maximize profits; the player with the most money at the end of a session is the winner. By pitting multiple models against each other, the most effective one can be determined by seeing who won. Previous projects have focused solely on using neural networks to model price trends, but real-world performance is also dependent on the ability to successfully place and execute orders. This project's full-market framework is able to achieve this with the ability to measure real-world performance. A full high-level training framework was created for the TensorFlow library to trail these models.

Keywords
linear regression — feed-forward — neural networks — tensorflow — bitcoin — cryptocurrency

[1] *Computer Science; School of Informatics, Computing and Engineering at Indiana University, Bloomington, IN, USA*

Contents

1. Background & Description

Bitcoin, arguably the best-known cryptocurrency, was created in 2009 with the goal of decentralizing financial transactions. In recent years, it has gained notoriety as a tool to circumvent law enforcement and as a quick way to get money. For this project, it is analyzed simply as a digital currency.

The open-ended nature of the project prompt ("create an AI") allowed a great degree of freedom. This project aims to compare and contrast different modeling techniques. There are multiple "player" classes containing unique models. Players are given a set amount of fiat currency (in this case, US dollars) and are tasked with buying and selling their holdings. The winner is the player with the most assets at market value after a set time period.

This project was done entirely in Python and R. In all diagrams and figures, **red** represents the linear regression model and **blue** the feed-forward neural network.

1.1 Constraints, Limitations & Restrictions

The biggest constraint met during the development of project is the time and expertise required to train models. In addition, quality readily-available cryptocurrency data was sparse. We elected to combine multiple sets to create a conglomerate Bitcoin dataset. As a result, historical data only goes back to April 2013, as sets had to be cut to align with each other.

1.2 Challenges

Due to the complexity of the TensorFlow library, it had to be studied thoroughly. This wasn't an easy task and took the entirety of Spring Break and a few weeks after to learn. In addition, the chaotic nature of the "super" dataset made it hard to work with.

1.3 Variations

Different cryptocurrencies have different histories. Bitcoin was chosen because it was the best documented currency. Ethereum was briefly considered, but was not the focus of this project given the time constraint. Importing the **ethereum.csv** dataset is easy in the *loader* class should we choose to analyze it.

2. Data Preprocessing

Most data preprocessing was conducted in R. Modified datasets were then exported as **.csv** files for easy referencing in Python.

2.1 Handling Missing Values

There were only 15 missing values in the Bitcoin dataset. Generally, missing data is taken care of by imputing their value or by removing them. Due to the small percentage of missing points, the rows containing them were simply removed.

2.2 Scaling

Feature scaling is a type of normalization that transforms values such that $\{x \in \mathbb{R} \,|\, 0 < |x| \leq 1\}$. The formula is as follows, with X being the point to be normalized:

$$X' = \frac{X - X_{min}}{X_{max} - X_{min}}$$

Finding the inverse function helps us undo the scaling. The formula for this reversal is as follows:

Let R be the point to be reversed
Let m_1 be the maximum of r's column
Let m_2 be the minimum of r's column

$$R' = R(m_1 - m_2) + m_2$$

2.3 Instance Creation

Training instances for one-day delay predictions were created by collating three preceding days' data columns with the current day's low-high price columns. Training instances for two-day delay predictions, similarly, were created by collating the three preceding days' data columns with the following day's low-high price columns.

3. Framework

The project is modular in the sense that models can be easily swapped for comparisons. For the purpose of this assignment, only linear regression, manual (human-controlled), and feed-forward neural network models were implemented.

The *loader* function loads any **.csv** file into the program. The directory is defined but can easily be changed to accommodate differing file hierarchies.

Model is perhaps the most important class. Its subclasses have several responsibilities, including defining variables, models, and the training step. The class has utility functions to handle training the defined models and applying them to data. Finally, accepts a loss factory function to train based on several different loss functions. One particular feature of the Model class is its *state* interface. The state interface allows the values of trained variables to be easily extracted and saved as a numpy array, and later substituted in at will. It allows the same tensorflow Graph and Session instances to easily service multiple optimization aims. Its primary advantages over the tensorflow Saver methods are its speed and its transparency.

The *Agent* class acts as a wrapper for models. There are a few types, but the most important are *informed_agent* and *tf_model_agent*. The informed agent has access to the whole dataset at once, and is aware of future price trends. This allows it to maximize profits and make smart trades. It acts as a ceiling for potential performance. The Model agent uses "blind" models that have access to only the data provided by the Market framework.

Market is responsible for feeding data to Agents that require regular updates and don't have unlimited access to data and executing their orders. The class also contains the demo which will be featured at the course symposium.

4. Algorithms and Methodology

4.1 Training

Our models were trained on the period from April 2013 to July 2016, and validated on the period from July 2016 to August 2017.

A training framework, which we have termed the 'Jack strategy', was developed in which every *frequency* training steps the loss would be evaluated on the validation dataset. The top *threshold* scores were kept in a heap, and if no top scores were registered within *patience* validations, the training program would terminate.

The training step utilized a TensorFlow AdamOptimizer in addition to an Adagrad optimizer. In order to get the best estimates, mean squared error was used when calculating loss. For lower bound estimates, a mean squared error was used that multiplicatively scaled positive squared errors at a ratio of 4 times that of negative errors. For upper bound estimates, we symmetrically used a mean squared error that multiplicatively scaled negative errors at a rate of 4 times that of positive errors.

4.2 Agent Strategy

Agents first estimated the difference between the next day and the following day's low and high prices in order to determine whether it is a good time to buy or sell.

If it is favorable to sell, they take a low estimate of the next day's high price (so that their sell will execute) and sell a ratio (confidence) of their crypto holdings at that price. If it makes more sense to buy, they symmetrically take a high estimate of the next day's low price and buy at that price.

4.3 Linear Regression

The linear regression model used in this project is as follows:

$$y = W * x + b$$

4.4 Feed-Forward Neural Network

The feed-forward neural network contained a hidden layer of 10 neurons. The model is as follows:

$$h = sigmoid(W_1 * x + b_1)$$
$$y = W_2 * h + b_2$$

4.5 Time & Space Complexity

The R script containing the code for data preprocessing has a time complexity of $O(n)$. This is due to the fact that every custom defined function only utilizes one for-loop at most. The models defined are highly efficient, executing in constant time. Model training, on the other hand, is a search optimization problem that has no guaranteed optimal solution. Theoretical time complexity for the search problem approaches the intractable $O(2^n)$.

Space complexity depends almost entirely on the optimizer class implemented by the specific model subclass. The AdamOptimizer used in this project's implementation is very efficient, storing less than 10 megabytes in RAM. It calculates the loss gradients with respect to each of the model variables and keeps track of a truncated history of these gradients. Because the dataset was relatively small, it was also stored in RAM.

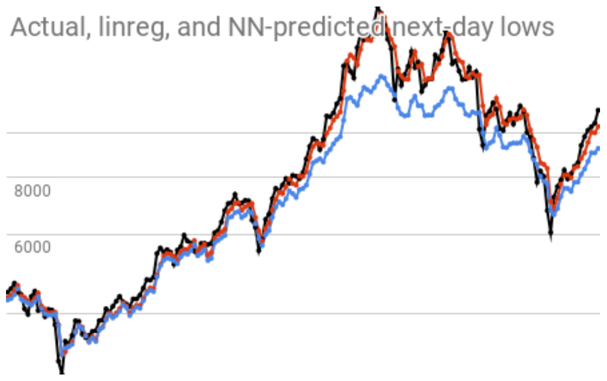
4.6 Alternative Approaches

An alternative model that attempted to predict the price change between the next and and following days was also considered, in addition to the raw prices of those two days. Based on the errors with the current results, that seems to have been the better strategy to take.

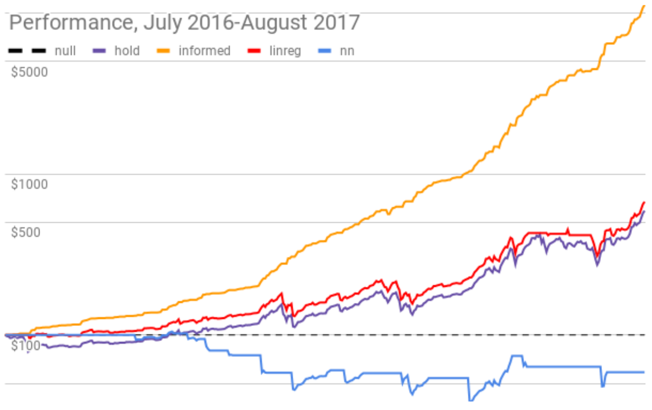
5. Experiments and Results

5.1 Model Performance

One interesting observation is that both models seemed to agree more often with each other than with the price trends. The agents performed well when capturing general trends in price data. However, the predicted price changes could have been more accurate.



The buy and sell prices represent the upper estimate of the next-day low and the lower estimate of the next-day high, as these are the optimal prices at which to buy and sell respectively.



5.2 Bitcoin Trading

The benchmarks for agent performance were a completely informed model and a model that simply held all its assets in cryptocurrency. The former enjoyed incredible success by netting over a thousand times the fiat currency it was given to start out with over the testing period. The latter expectedly performed worse and at times ended the simulation with less money than it started with.

The agents performed reasonably well, with the linear regression model tending to outperform the 'buy-and-hold' model over the July 2016-August 2017 validation data. Possibly due to the scaling model employed, the agents were unable to adapt to unforeseen high prices during the August 2017-February 2018 test period. Different training outcomes for their $n+1$ and $n+2$ predictions (with n being the current day) resulted in their predictions becoming virtually meaningless. In order to implement the code on unseen price fluctuations, the changes detailed in the next section were implemented and/or considered.

The linear regression model performed slightly better to the "hold" model, in which the starting fiat is immediately traded for BTC, and only traded back in the last step. The neural network model dramatically underperformed compared to our expectations. In the simulations, it was more beneficial to simply hold the money you start out with than to buy/sell based on predictions.

6. Expansion & Improvements

In order to improve future performance, there are a few things that could be changed or improved upon. First, trying new regularization methods that are more resistant to novel price ranges would have made for a better dataset to train the models on. Second, having instance generation more sensitive to the performance aspects of the models would have given them more insight into how the price would change from tomorrow to the next day. More time for model training and tuning would have been appreciated and is a must for future revisions of this project.

The biggest weakness of this project was the overgenerality of the training process. Different targets were trained separately even though the relationship between them was the operant factor. For example, the n, $n+1$, and $n+2$ price predictions with n being the current day. The neural network should have achieved significantly higher performance, and it is believed that this disconnect between the targets is one of the reasons why it didn't meet expectations. Linear regression models generally serve as the baseline for model performance, so the fact that the feed-forward network performed worse shows that there is a lot of work that could be done to improve it.

Using a recurrent neural network would have achieved more favorable results than the feed-forward network implemented in this project. Due to the nature of RNNs, it would have been more resistant to sudden price fluctuations.

7. Conclusion

In conclusion, the project's **linear regression** model and particularly its **feed-forward neural network** model didn't perform as expected. Given more time to tune and train the models, and a higher-quality dataset, the models would have performed much better, or at least as expected.

The modular nature of the project allows for easy swapping of models in the future, which means updates are easy to implement. This makes the search for the best performing model more streamlined.

8. Libraries

8.1 Python

- TensorFlow
- NumPy
- pandas

8.2 R

- caret
- dplyr
- e1071
- zoo
- ctv
- readr
- clusterSim

References

1. *TensorFlow* by Google Brain Team
2. *Normalization (statistics)* on Wikipedia

Special thanks to Mathias Legrand and Vel for creating this LaTeX template.