Cry0 Final Report:
Cryptocurrency Algorithmic Trading (Arbitrage)

Team Code: CRY0

Date: 3-16-2018

Team Members:

- Timothy Bramlett, email: bramlett@oregonstate.edu; Github: TimothyBramlett
- Patrick Mullaney, email: mullanep@oregonstate.edu; Github: mullaneyp
- Matt Nutsch; email: nutschm@oregonstate.edu; Github: mnutsch

You can find a short video introducing the project here:
https://media.oregonstate.edu/media/t/1_0vcs7gjq

I. Introduction

For our capstone, we developed a cryptocurrency wallet application with the ability to search for arbitrage opportunities. The application was written in python and does not need a graphical user interface as it can run from a command line and is currently running on a Linux server to maximize its arbitrage opportunity search.

Through its queries, the application is capable of finding single currency arbitrage opportunities, where a price difference may exist for the same currency at different exchanges. Additionally, it is also capable of detecting arbitrage opportunities for multiple currencies across a chain of up to three transactions. For example, there may be some situations where no single currency arbitrage opportunities exist between currency A between different exchanges, however, there may be a chain of transactions such that price differences between currency A and currency B or currency C at different or the same exchange could yield a profit.

While running, the application repeatedly queries at the various exchanges rates between cryptocurrencies at recurring intervals. If it finds a situation where the exchange rates are not consistent, depending on configuration, the application will notify the user of the opportunity via email or SMS text message. For example, the app check the USD exchange rate for Bitcoin Core (BTC) at the exchange "GDAX" and also the exchange rate for Bitcoin Core (BTC) at exchange Gemini. If the difference in price exceeds a preset threshold, then the application will send a notification either by SMS text message or email.

II. Setup & Usage

Setup a web server:

Set up a Linux web server. We recommend hosting the application on Amazon Web Services' (AWS) Elastic Cloud Compute (EC2) service. You can find instructions to do so here:

https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_Tutorials.WebServerDB.CreateWebServer.html

Download the project code:

A repository on Github.com has been used to store the application's code. Navigate to the following URL and download the contents. Upload the extracted contents to the application web server.

https://github.com/mnutsch/Crypto_Algorithmic_Trading

Install Python

Install Python on the web server. This application was built and tested with Python version 2.7.5. You can find instructions for installing the latest release of Python here:

https://www.python.org/

Install Python libraries:

From the Linux command prompt run the following command. It will install the libraries utilized by the application:

pip install -r requirements.txt

Set up Cryptocurrency exchange accounts:

In order to transact in cryptocurrencies you will need to register accounts with the cryptocurrency exchanges that the application interfaces with. Note that it may take several days for the exchanges to process your account requests.

1. Create a user account at the GDAX exchange. Follow this URL for more information: https://www.gdax.com/
2. Create a user account at the Gemini exchange. Follow this URL for more information: https://gemini.com/
3. Add GDAX credentials to the file settings in the following files:
    a. checkAccountBalanceGDAX.py
    b. execTransactionsGDAX.py
    c. withdrawToCryptoAddressGDAX.py
4. Add Gemini credentials to the file settings in the following files:

    a. checkAccountBalanceGemini.py
    b. execTransactionsGemini.py
    c. withdrawToCryptoAddressGemini.py
5. Update the exchange fee amounts.
    a. Look up the latest fee amounts from the exchanges' websites.
    b. Edit the following files and update the fee amounts to the latest numbers:
        i. currency.py
        ii. exchange.py

Set up a Tropo account for text (SMS) notifications:

In order to receive notifications with text (SMS), you will need to register an account with Tropo. Note that it may take a couple of days for Tropo to process your account request once you register a payment method.

1. Create an account with the service Tropo. Follow this URL for more information: https://www.tropo.com/
2. Follow the instructions at this URL to install and configure the Tropo component of the application: https://github.com/mnutsch/Crypto_Algorithmic_Trading/blob/master/tropo/readme.txt
3. Edit the file configuration.py in a text editor. Input the phone number where you want to receive text (SMS) messages. You can enter one phone number in the format:

    PHONE_NUMBERS = ['15551234567']

    Alternatively, you can enter multiple phone numbers in the format:

    PHONE_NUMBERS = ['15551234567', '15551234568']
4. Edit the file .bashrc to include the API token and SMS authentication code for the Tropo app. The file .bashrc is a hidden file which resides in a Linux user's home directory. Add the following lines to the file. Update the values based on those of the app in Tropo.

    export TROPO_API_TOKEN="12345abcdef"
    export TROPO_SMS_AUTH="12345"

Set up an email account for email notifications:

1. Register an account with an email service of your choice (i.e. Gmail).
2. If needed, enable third party email clients in the email service's settings.
3. Edit the file .bashrc to include the username and password for the email service. This is a hidden file which resides in a Linux user's home directory. Add the following lines to the file. Enter the email address and password for the email account where appropriate.

    export GMAIL_EMAIL="yourEmail@gmail.com"
    export GMAIL_PW="enterYourPasswordHere"

Users can also adjust settings to suit their needs in the configuration.py file:

```
1
2  # Set time interval(seconds) to check for opportunities.
3  INTERVAL = 300 # seconds
4
5  # Set phone numbers for text/sms notifications.
6  PHONE_NUMBERS = ['1234567890']
7  # Multiple phone numbers are possible.
8  # PHONE_NUMBERS = ['1234567890', '9876543210']
9
10 # Set email for email notifications.
11 EMAIL_LIST = ['sampleEmail@gmail.com']
12 # Multiple emails are possible.
13 #EMAIL_LIST = ['sampleEmail@gmail.com', 'sampleEmail2@gmail.com']
14
15 # Set a threshold amount for opportunities you would
16 # like to recieve notifications/execute trades on.
17 THRESHHOLD = 0
18 # Set maximum cost range of transactions (in dollars).
19 # Application will search ranges from 0 to MAX_COST.
20 MAX_COST = 500
21 # Set the minimum profit range amount (in dollars) to return opportunities on.
22 # Note: losses are possible, to see negative profit/loss opportunities,
23 # set MIN_PROFIT to a negative value.
24 MIN_PROFIT = 0
25 # Set true to send notifications via SMS.
26 NOTIFY_SMS = False
27 # Set true to send notifications via email.
28 NOTIFY_EMAIL = False
29
```

After appropriate setup and configurations:

1. Enter command to run on server:

   python main.py

2. Enter ‘ctrl-z’ to quit.

A sample of the program polling for opportunities:

```
Checking single currency opportunities...
Buy: ETH @ gemini: $601.85
Sell: ETH @ gdax: $602.13
Amount of coins: 1
Profit/Loss: -0.93

Checking single currency opportunities...
Buy: ETH @ gemini: $598.60
Sell: ETH @ gdax: $599.29
Amount of coins: 1
Profit/Loss: -0.51

Checking single currency opportunities...
Buy: ETH @ gemini: $597.24
Sell: ETH @ gdax: $597.50
Amount of coins: 1
Profit/Loss: -0.94
```

```
Checking single currency arbitrage opportunities...
Currently no profitable single currency opportunities.

Checking multiple currency arbitrage opportunities...
There are 30 possible transactions...
Checking 24360 possible transaction permutations...
Checking 480 possible transaction chains...
Currently no profitable multiple currency opportunities.
```

In this interval, no profitable opportunities existed

III. APIs, Libraries, Tools, etc.

GDAX and Gemini Cryptocurrency Exchange APIs:

Exchange APIs were perhaps the most essential tools we worked with. Calls to GDAX's and Gemini's APIs enabled us to retrieve pricing on the various cryptocurrencies we selected, which in turn enabled us to calculate arbitrage opportunities. Furthermore, calls to the API enabled implementation of some of the application's wallet functionality, specifically the ability to check a user's account balances and withdraw funds from a user's account.

- GDAX cryptocurrency exchange API info: https://docs.gdax.com/?python#introduction
- Gemini cryptocurrency exchange API info: https://docs.gemini.com/rest-api/?python#introduction

Python programming language & libraries:

hashlib library:

Python's hashlib library provides secure hash algorithms that are useful in developing functions to send data more securely and was used in the development of the files to access and transfer account data at the various exchanges, specifically, checkAccountBalance[exchange].py and withdrawCryptoAddressFrom[exchange].py. More information on the module can be found here: https://docs.python.org/2/library/hashlib.html.

Itertools library:

The itertools library was used primarily for the algorithms behind checkForSingleCurrencyOpps.py and checkMultipleCurrencyOpps.py file. A moderate amount of mathematical research went into the algorithms, predominantly using wikipedia articles on permutations and pairwise comparison (https://en.wikipedia.org/wiki/Pairwise_comparison). Itertools provides a series of functions for efficiently looping over lists, including permutations function which was of great use in all algorithms. For each of the algorithms, the application first builds a list of all relevant currencies via calls to the Gdax and Gemini APIs. Itertools' permutations function then returns a list of all permutations of currency pairs, representing a list of transactions. Each pair represents a transaction for currency A to currency B. For checkMultipleCurrencyOpps.py, itertools' permutations function is then used again via calls to functions in the conversions.py and chains.py files, this time to return a list of permutations of up to three transactions from the transaction list, representing a possible transaction chain up to three transactions (it is possible to explore longer transaction chains, however this could be financially risky due to time constraints). Documentation on itertools can be found here: https://docs.python.org/2/library/itertools.html.

NumPy library:

The NumPy library was used in some calculations. As development progressed and it became apparent that there were not a large amount of arbitrage opportunities and some of those that did exist, existed only for currencies which had a significantly high value (i.e. Bitcoin), we began implementing our algorithms with capability for fractional amounts of cryptocurrencies. Unfortunately python's native range function does not support float data types and this was causing either errors or truncation of calculations. NumPy's arange function was able to to address this. After some trial and error, calculations are run on coin denominations as small as 0.001. Smaller fractions significantly slowed down calculations for large lists. Documentation on NumPy can be found here: https://docs.scipy.org/doc/.

requests library:

The requests library contains various functions to assist with HTTP requests and was primarily used for authentication purposes in development of wallet functionality files, checkAccountBalance[exchange].py and withdrawToCryptoAddressFrom[exchange].py. Further information can be found here: http://docs.python-requests.org/en/master/.

smtplib library:

The smtplib library was used to in development of the sendEmail.py file. As referenced in the documentation, it is a handy module for creating an SMTP client session object for sending emails. The various functions in that are called as appropriate by main to send notifications of arbitrage opportunities that have met a profitability threshold specified in main. Further information can be found here: https://docs.python.org/2/library/smtplib.html.

time library:

The time library was used predominantly in main and for some test files, however it served a simple, yet important purpose, that of intervals. The configuration file offers an interval setting in seconds to specify the time intervals between polling the APIs for current rates and evaluate arbitrage opportunities. This was implemented with the use of time's sleep function. It should be noted however that most exchange APIs actually set a call rate limit for requests, typically a limit per seconds. For our purposes however, we frequently set this for increments of 60 seconds due to email and sms notifications being on and did not want to get flooded with messages. Further information on the time library can be found at https://docs.python.org/2/library/time.html.

urlib and json libraries:

Python's urlib and json libraries were used primarily to retrieve pricing data from the exchange APIs. Specifically, urlib's urlopen function was used to fetch content the various exchange's particular product page while json.loads was used to extract the contents. This functionality was used in development of the readExchangeRates[exchange].py files in order to retrieve live currency pricing, which was then in turn evaluated by the checkForSincleCurrencyOpps.py and checkForMultipleCurrencyOpps.py files. Further information on urlib and json can be found at https://docs.python.org/2.7/library/urllib.html and https://docs.python.org/2/library/json.html respectively.

Amazon Web Services (AWS) EC2:

We set up a dedicated project server running Linux using Amazon Web Services (AWS) Elastic Cloud Computer (EC2). This was necessary due to limitations in scheduling and keep alive permissions on the Oregon State University "Flip" server.

Git and GitHub:

We used Git and GitHub as our version control system and code repository respectively. We designed our application to be as modular as possible, which made development of individual files simple. Consequently, Git/GitHub served us well as we did not have a lot of issues with merge conflicts, etc. but it still assisted us greatly in sharing code throughout development. The repository can be found at this URL: https://github.com/mnutsch/Crypto_Algorithmic_Trading

Tropo (text / SMS service):

We used Tropo's API and service for the text/SMS notification portion of our application. While the application is running on the server, when an arbitrage opportunity exists that is above the profit threshold set in the configuration file, a notification is sent to the user or phone numbers specified in the configuration file via a call to Tropo's API messaging service. More information on Trop may be found here: https://www.tropo.com/docs/rest.

Stack Overflow

While not heavily relied upon and appropriately cited when referenced, we would be remiss in the capstone of our academic career to not provide an honorable mention to the wonderful resource and reference that is Stack Overflow.

Useful Links:

While not a software development tool, a useful reference for arbitrage calculations can be found at https://99bitcoins.com/bitcoin-arbitrage/. This reference was listed in our project proposal and while it was not the basis for all our algorithms, its detailed calculation and consideration of associated fees beyond the potential profits due to the price inconsistencies was a great resource that was taken into consideration during development of calculations.

IV. Teammate Contributions

Matt Nutsch:

Matt got us started by setting up the Linux server with Amazon Web Services (AWS) EC2 and writing the files to obtain pricing data through the exchange APIs. These first steps were essential for gettings us started by enabling us to poll the data we needed to explore algorithmic trades and opportunities. Matt also set up the Tropo account and wrote sendSMS.py file necessary for text/SMS notifications of arbitrage opportunities. He further

developed files necessary to implement wallet functionality, specifically the files capable of checking account balances and withdrawing funds and the

Summary:
Setup the Linux server for hosting the project application.
Wrote the readExchangeRates[Exchange].py files.
Wrote the checkAccountBalance[Exchange].py files.
Wrote the withdrawFromCryptoAddress[Exchange].py files.
Wrote the logCryptoPrices.py script.
Wrote the sendSMS.py files.
Setup the Tropo account (for text SMS) and wrote the tropoSendSMS.py file.

Patrick Mullaney:

Patrick implemented the algorithmic components of the application in the checkSingleCurrencyOpps.py and checkMultipleCurrencyOpps.py files. Relying on mathematical research on combinations and permutations, he explored various algorithms to build transaction chains to evaluate for profitability. He developed different algorithms that built transaction chains both conservatively through cost minimization and aggressively through profit maximization, optimization algorithms via dynamic programming to improve performance while still working with the live data provided by the API calls. He also implemented to the email notification functionality for arbitrage opportunities in the sendEmail.py file.

Summary:
Wrote the sendEmail.py file.
Wrote the currency.py file.
Wrote the exchange.py file.
Wrote the checkSingleCurrencyOpps.py file.
Wrote the checkMultipleCurrencyOpps.py file.
Wrote the calc.py file.
Wrote the conversions.py file.
Wrote the chains.py file.
Wrote the testArb.py file.
Modifications to configuration.py.
Modification to main.py.

Timothy Bramlett:

Tim brought us home by pulling the various pieces together. He wrote the main.py file and conceived the skeleton for how the various pieces would integrate. He also implemented the transaction capability of the application in the execTransactions[exchange].py files. He further wrote the .bashrc file, useful in pulling in necessary passwords and api keys from the environment that we would not want accessible in a public code repoistory.

Summary:
Wrote the main.py file.
Wrote .bashrc file to import certain passwords and api keys in from the environment
Wrote execTransactionsGemini.py
Wrote execTransactionsGDAX.py

All team members:
Registered accounts at the cryptocurrency exchanges.
Tested application code and functionality.
Wrote instructions for setting up and using the application.

V. Deviations from Original Plan

There were several changes from our original plan, much of which did not have a significant impact on the original concept. The first change was that of the server. While originally intended to run the server on OSU, limitations with the flip environment for students led us to reconsider and find a more flexible server in AWS as this would allow us to run Cron scripts and schedule tasks with more flexibility than the OSU server.

Another deviation from the original plan was that we found that Bitcoin Cash (BCH) and Litecoin (LTC) were not available for “single currency” opportunities because they were not supported by the Gemini exchange API, however this was not a significant issue as there were other single currency opportunities and the currencies remained relevant for multiple arbitrage opportunities.

Perhaps a more significant deviation from the plan was in the algorithmic approach to the checking for multiple currency arbitrage opportunities. The basic principle of our idea was to identify price inconsistencies across cryptocurrencies and exchanges and determine if transactions would be profitable after various fees were considered. On a deeper level however, this proved quite challenging as the potential risks became apparent quite early on. There are fees to consider for all transactions. Fees not only vary in value between exchanges, but they vary also in the methodology in which they are calculated by the exchange. Consequently, we had to explore different ways of calculating the same fees for different exchanges and currencies. When factoring in these various fees across exchanges, they very often negate any profitability of a transaction. Given the same, we were forced to rethink our algorithms

several times, taking a trial and error approach while attempting to balance the optimization of the application from a technical perspective against the correctness and profitability of the application.

Initially, we had proposed checking for potential multiple currency opportunities via checking the transaction costs across a chain of transactions, seeking the lowest cost of the transaction. The code for this was written and functions, however, after running for some time, we simply weren't getting many profitable results. After further research, we were able to determine that this approach was rather conservative. Its risk averse nature is effective in preventing a user from losing money, however it did not yield many profitable results.

Consequently, we revisited this matter and developed a more aggressive algorithmic approach. The second attempt, is focused on maximizing profitability rather than minimizing cost or exposure. Conceptually, the algorithms share some common structure in the sense that they both form a chain of transactions based on permutations of potential currency transactions, however as the logic driving the formation of the transaction chains was different, it did necessitate writing different functions. Similar to the first algorithm based on conversion costs, the algorithm focuses on building a chain of up to three links, each link representing a transaction from currency A to currency B or currency A at exchange #1 to currency A at exchange #2. When iterating through the various permutations of these three transactions, the algorithm calculates the profit/loss value of executing the entire transaction chain, and builds a list of profitable chains of transactions. This approach yielded significantly more profitable opportunities. The profitability of the opportunities was not always significant, however, it was certainly more frequent than the algorithm focused on costs first and considered profitability secondary. In the end, to verify profitability potential, we actually had to implement best case scenario fee structure for high volume traders (simulated, as we realistically lack the resources to trade at that volume), which did return profitable values.

VI. Conclusion

Cryptocurrency arbitrage is a potentially profitable, yet challenging and risky venture. We entered into the beginning of this project enthusiastically and perhaps a bit too optimistic, but overall, the challenge itself was a great learning experience. On a high level, the philosophy behind our concept was quite simple, develop an application to query cryptocurrency exchange APIs for currency pricing, determine if there are any

pricing inconsistencies, and calculate the profit/loss value of any potential transactions to obtain a profit.

Another aspect that was challenging was that as the majority of of our project relied on information provided by APIs, we were often constrained by the limitations of those APIs and the challenges of trading on cryptocurrency exchanges as a whole. For example, in order to successfully profit off an arbitrage opportunity, one must execute the series of trades in the opportunity as close to real time as possible. The longer the time it takes to execute the transactions, the the more risk that a fluctuation in either price or volume (and in turn, fees) could change a profit into a loss. This had a huge impact on our strategies as there were instances where we could technically implement something, but it may not be viable in real world transactions. Take for example our check for multiple arbitrage opportunities files. We did a bit of optimization via dynamic programming to reduce time obtaining conversion costs, however we capped the potential chain of transactions at three transactions. While it would have been technically feasible to implement the function up to N transactions, the potential risk associated with executing more transactions than that before the opportunity had gone stale increases.

Overall, this application was a learning experience. While development of the application failed to make us rich, it did succeed in other valuable ways. We relied on and applied all of the skills we have developed throughout this program. We conceived an application on our own and implemented it. When our initial ideas or strategies were not successful, we developed our own solutions. When the initial tools we anticipated on using did not suit the application's needs, we sought out and integrated tools more appropriate for our purposes. When we were constrained by the APIs we were working with and the application's performance was inhibited, we implemented programming solutions to overcome these obstacles. We set out to develop an application which would identify potential arbitrage opportunities and determine if they were profitable. While profitable opportunities exist, they are not as frequent as they are perceived and the investment required to turn a profit is often quite significant. In short, you need to invest a lot of money to profit significantly off of arbitrage opportunities. Regardless, our application functions and prevents us from executing trades in which profit would be negated by fee structures despite the initial price discrepancy. We also developed the application and overcame roadblocks by applying the skills and concepts we learned in this program and in that sense, we succeeded.

Appendix A - File Descriptions:

main.py = This is the main file which is run by the Cron job and/or user. It remains a work in progress as to further incorporation of files that remain in development however is currently running on the server.

configuration.py = This file contains configuration information. It defines API keys for different cryptocurrency exchanges. Modifications may define minimum or maximum thresholds when checking for arbitrage opportunities and limits on the amounts of transactions for perform.

readExchangeRates[Exchange].py = This is a series of files. Each file contains functions that read data from a cryptocurrency exchange and then parse the data into a common object format. Replace [Exchange] with the name of the exchange. The application reads rates from two exchanges, GDAX and Gemini. Functions from these files have been imported into main.py, checkForSinglecurrencyOpps.py, and checkForMultipleCurrencyOpps.py.

checkForSingleCurrencyOpps.py = This file contain function(s) which check for arbitrage opportunities from trading a single currency between different exchanges. The file remains a work in progress from an optimization standpoint to generate better results.

checkForMultipleCurrencyOpps.py = This file contains function(s) which check for arbitrage opportunities from exchanging a chain of currencies between each other. It remains in development at this time. Currently, it evaluates a potential chain of currency transactions by first calculating the cost of converting $[amount] currency A to $[amount] of currency B, $[amount] of currency B to $[amount] of currency C, and $[amount] of currency C to $[amount] of currency A. A function will then iterate through all permutations (currently up to 1,320 permutations) of these cost combinations and return a list of potentially profitable chains of transactions.

calc.py = This file contains functions for calculating profit/loss, fees, etc. These functions are accessed by functions called in both checkForSingleCurrencyOpps.py and checkForMultipleCurrencyOpps.py for determining the true cost of an arbitrage opportunity.

chains.py = This file contains functions for creating transaction chains to be evaluated by checkForMultipleCurrencyOpps.py. It is capable of building transaction chains up to three consecutive transactions, however we determined that longer transaction chains unfortunately would increase the risk of a profit turning into a loss as the lengthy cryptocurrency transaction times could cause the data on transactions toward the end of the chain to be stale by the time the trade is executed.

conversions.py = This file contains functions for calculating conversion costs for transactions which can be called by checkForMultipleCurrencyOpps.py. It was used heavily in our first attempt to find arbitrage opportunities by building transaction chains based on the lowest cost for fees, etc. Performance was improved via a dynamic programming implementation of conversion cost tables. This approach was the most safe and risk averse, however it frequently did not return any transactions that were profitable and consequently led us to explore more aggressive approaches.

currency.py & exchange.py = These files contains functions for returning 'currency' and 'exchange' objects which store data about particular currencies and exchanges to be calculated and updated throughout various files used by checkForSingleCurrencyOpps.py and checkForMultipleCurrencyOpps.py.

sendEmail.py = This file contains functions for generating an e-mail to the user(s). Functions from this file will be imported into main.py.

sendSMS.py = This file which contains functions for generating a text (SMS) message to the user(s). Functions from this file have been imported into main.py.

tropoSendSMS.py = This script runs on the Tropo (text SMS) service. It handles incoming calls from the sendSMS.py file and translates those connections into SMS messages.

logCryptoPrices.py = This script logs crypto-currency exchange rates in a CSV file. The log it creates was intended to be used to help us determine what is a safe margin of error for executing an arbitrage transaction. However, throughout the course of development, we learned that majority of price inconsistencies between currencies and exchanges were not profitable when fees were considered.

withdrawToCryptoAddressFrom[Exchange].py = This is a series of files which transfer cryptocurrency from a wallet at one exchange to a wallet at another exchange. Although not in the original project plan, we determined that this is needed to fully automate a full arbitrage trade.

execTransactionsGDAX.py and execTransactionsGemini.py = This is a series of files which purchase or sell cryptocurrencies at the exchanges.

requirements:txt = A file essential for installing the libraries necessary to run the application. Run with the following command: *pip install -r requirements.txt.*