THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF FINANCE

Performance of Semi-High Frequency Trading Algorithms in Python Based on Dark Pool Movements

MUTIAN FAN
SPRING 2022

A thesis
submitted in partial fulfillment
of the requirements
for baccalaureate degrees
in Computer Science and Mathematics
with honors in Finance

Reviewed and approved* by the following:

Mihail Velikov
Professor of Finance
Thesis Supervisor

Brian Davis
Professor of Finance
Honors Adviser

* Electronic approvals are on file.

ABSTRACT

Dark pools are hidden stock markets, which do not show trades before they occur as opposed to a transparent market such as the New York Stock Exchange. Not much research has been done in algorithmic trading based on dark pools; thus, the purpose of this thesis is to see if dark pools are able to predict movements in the market and generate a positive return in the market. This will be done using algorithmic trading done in the Python programming language, through TD Ameritrade's trading platform which allows foreign programs to access market information.

The trading program will be set up using a web scraper to gather live dark pool data, as there is a lack of historical information to back test an algorithm. Then, it will log this information to be analyzed later. An analysis through looking at the assets of the algorithm given a starting amount of $25,000 will be done and compared with the price movement on SPY during the period of data collection. Risk-based analysis will be done using a Sharpe ratio with the risk-free rate of the U.S. treasury yield. After the analysis, it was shown that the algorithm performed extremely well despite heavy limitations on how many shares it could buy at any given time. Although some assumptions were made for live market performance, it can be said that dark pools are a valid way to make a good semi-high frequency execution algorithm.

TABLE OF CONTENTS

LIST OF FIGURES iv

ACKNOWLEDGEMENTS v

Chapter 1 Literature Review 1

1.1 Introduction 1
1.2 Algorithmic Trading 2
1.2.1 History 2
1.2.2 Measures of Efficiency 4
1.2.3 Liquidity 6
1.2.4 Costs of Algorithmic Trading 7
1.2.5 Machine Learning 9
1.3 Dark Pools 9
1.3.1 Introduction 9
1.3.2 History 10
1.3.3 Types 11
1.3.4 Price Discovery 13

Chapter 2 Code 15

2.1 Introduction 15
2.2 Creating the Server 15
2.3 Scraping Data 16
2.4 Timing 19
2.5 Trading 21

Chapter 3 Connecting to the Market 25

3.1 Introduction 25
3.2 POST Request 26
3.3 Authentication 27
3.4 Placing Orders 29

Chapter 4 Analysis of Performance 31

4.1 Introduction 31
4.2 Price Charts 32
4.3 Risk Assessment 35

Chapter 5 Conclusions 38

5.1 Future Research – Limitation, Efficiency, Scalability 38
5.2 Conclusion 39

BIBLIOGRAPHY 41

ACADEMIC VITA 42

LIST OF FIGURES

Figure 1. Graph of the total money after each day of running the algorithm from the period 9/27/2021 to 2/2/2022 with money in dollars on the left axis and date on the bottom axis. 32

Figure 2. Graph of SPY price movement from the period 9/27/2021 to 9/2/2022 with dollars on the right axis and date on the bottom axis, retrieved on 2/8/2022.....33

Figure 3. The day over day percent change of the algorithm from 9/27/2021 to 2/2/2022 with the percent value on the left axis and date on the bottom axis.....34

Figure 4. The day over day percent change of SPY 9/27/2021 to 2/2/2022 with the percent value on the left axis and date on the bottom axis. 34

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Dr. Mihail Velikov and my advisor, Dr. Brian Davis for all the guidance they gave throughout the process. Thank you to James Qu, who provided excellent guidance in the financial aspect of this thesis – it would not have been possible without you. Thank you to Sharon Liu, who provided me a lot of support throughout the difficult times these last couple of semesters. Lastly, a thank you to my family and friends who have always supported me and made my time at Penn State one I will remember forever.

Chapter 1

Literature Review

1.1 Introduction

In the past decades, technology has revolutionized the way we go about our day to day lives. Instead of slow communication taking days to weeks we can now instantly send messages across the world. With a push of a button, we can get food delivered or have transportation waiting for us. Beyond our personal lives, technology has also drastically changed our industries, governments, and societies. This has been even more apparent during the ongoing Covid-19 pandemic during which our entire lives shifted online. With everything shifting online, it is to no surprise that the financial markets quickly embraced the ever-changing technology and evolved in every aspect.

Algorithmic trading was one of these evolutions. Although algorithmic trading is technically any trading done with an algorithm, in this paper we will specifically be referring to those which are done using a computer program and automated. This type of trading can have many forms – from low-frequency trades, which simply decide which stocks are the best to buy and hold over long periods of time, to high-frequency trades, during which many trades can be executed within fractions of a second. Algorithmic trades are applied to every market out there – from the public exchange to private equity to foreign markets; nowadays ubiquitous in the financial world. Unfortunately, the research surrounding algorithmic trading is mostly

proprietary and very often kept secret, so much prior research given will not apply directly to pythonic execution algorithms specifically.

1.2 Algorithmic Trading

1.2.1 History

Broadly speaking, algorithmic trading is a set of automated trading strategies which follow variables – such as time, price, volume, and historical/statistical patterns – in their decision-making process (Hilbert et al., 2020). Three main types of algorithmic trading are identified: execution algorithms, high-frequency trading, and algorithmic market making. The initial influence of algorithmic trading can be traced back to the mid-1990s (Hilbert et al., 2020).

After a few years, in 2003, the New York Stock Exchange (NYSE) introduced an automated system which automatically distributes new quotes in a system called “autoquote” (Hilbert et al., 2020). In following years, markets started providing automatic interfaces and trading systems; these still did not have the level of sophistication as today’s algorithms. However, from 2008 to 2013 although sell-side algorithms in foreign exchange markets did not increase by much, buy-side algorithms nearly tripled (Hilbert et al., 2020). In fact, even within the U.S. algorithmic trading was thought to be responsible for as much as 73% of the trading volume (Hendershott et al., 2011).

The actual algorithms themselves originated as aids for sell-side traders to help boost efficiency (Johnson, 2010). Around two decades ago, brokers realized that they can directly offer these aids to clients and the major brokerages were all offering algorithmic trading services as

well as giving frameworks to customers to help construct their own algorithms (Johnson, 2010). The very first algorithms followed naturally from order slicing that focused on specific measures, starting with Time Weighted Average Price (TWAP) proceeding to Volume Weighted Average Price (VWAP) (Johnson, 2010). VWAP was simpler and more accurately reflects price movement, thus it was used more widely (Johnson, 2010). Usually, both types of algorithms were statically driven, meaning as soon as an order was received a schedule for trading was created and executed (Johnson, 2010).

A second generation of algorithms followed, created due to an application of transaction cost analysis (TCA) which simply breaks down costs associated with trading (Johnson, 2010). Some of these costs come in the form of the effect of an order on price, timing risk, opportunity cost, and implementation shortfall (IS) – coined by Perold in 1988 to represent actual costs of trading which is represented by the difference between the market price when the decision to trade was made and the actual price during execution (Johnson, 2010). The emergence of TCA caused a reexamination of current algorithms and resulted in VWAP being replaced by decision price and algorithms moving away from minimal market impact to focusing on IS based algorithms (Johnson, 2010).

The third generation of algorithms came from two factors – the continuous search for liquidity and increasing order book transparency from the transition to electronic markets (Johnson, 2010). In previous generations, algorithms only looked at the best bid and offer quotes as there simply wasn't other information available; thus, as order data became more accessible more algorithms started utilizing this data for their placement decisions (Johnson, 2010). In this

third generation, algorithms started shifting from simplistic order routing systems to complex liquidity-based ones (Johnson, 2010).

Other classes of algorithms have also appeared over the years. One major example is off-market trading shifting to electronic venues such as dark pools or alternative trading systems (ATS) which algorithms now use to find liquidity at set prices to achieve best execution (Johnson, 2010). This behavior is now being incorporated into IS or VWAP algorithms, not just limited to dedicated liquidity-seeking algorithms, giving rise to hybrid strategies as well (Johnson, 2010). Another shift in algorithmic behavior occurred which focused on adaptability and customization to allow an offering of more client centered algorithms (Johnson, 2010).

Overall, algorithmic trading has had a large impact on the market. Algorithmic trading has allowed for much more complex and quicker trading strategies as well as varying improvements in efficiency, usability, and cost over the years. Now, a clear trend in the growth of algorithmic trading can be seen with a decline in other types of trading. In this paper, I will try to use algorithmic trading techniques and do some analysis on dark pools, which were mentioned briefly as one of the many factors which drove algorithmic trading development.

1.2.2 Measures of Efficiency

In the past decade, algorithmic trading – specifically, high frequency algorithmic trading – has had a large presence in the equity markets in the United States. Since the financial crisis of 2008 and the ‘flash crash’ of 2010, high frequency trading is said to account for over half of the volume in U.S. equity shares (Castura et al., 2010). In their paper “Market Efficiency and Microstructure Evolution in U.S. Equity Markets: A High-Frequency Perspective” Castura et al.

analyze the impact resulting from the integration of high-frequency trading inside equity markets, specifically from an efficiency standpoint.

Castura et al. (2010) give a couple measures of efficiency – the bid-ask spread which is expected to be driven down thus decreasing costs to investors, high liquidity which represents ability of investors to obtain desired investments, and a couple of other tests given by other authors such as looking at serial autocorrelation or the variance ratio test by Lo and Mackinlay.

Bid-ask spreads correlate to efficiency because smaller spreads imply better costs to trading; this also means that smaller spread market makers make less revenue because other participants are more competitive and shows a healthy market (Castura et al., 2010). Through their analysis, Castura et al. (2010) showed that the introduction of algorithmic trading corresponded with a decrease in spreads which was shown in many index funds such as the Russell 1000 and Russell 2000 with VIX-adjusted spread data. Liquidity, loosely defined as the ability of participants to trade at-will, can be measured by the amount offered for sale at any given point in time (Castura et al., 2010). Following this, available liquidity can be measured as the actual dollar amount available to buy or sell at any point in time (Casturea et al., 2010). Just like trends in bid-ask spreads, the trends in liquidity also showed increases after high-frequency trading was introduced in equity markets.

Other efficiency tests performed by Castura et al. (2010), noted above, also showed an increased market efficiency after the introduction of high-frequency algorithmic trading. Although the algorithm which I will be testing is not on such a macro scale, the look at measures of efficiency is valuable, especially with many similar factors to consider. For example, liquidity and slippage are a large concern with the type of algorithmic execution I will be attempting,

especially when the algorithm might look at stocks with less liquidity and be unable to execute trades at the volume desired. Thus, while the same measures of efficiency cannot be directly applied, the general ideas are a valuable starting point. The ideas that algorithmic trading evens the playing field for all market participants is another important driver for my research.

1.2.3 Liquidity

Since algorithmic trading has grown from its starting point in the 1990s, liquidity in markets has also improved. However, we cannot simply conclude that algorithmic trading is responsible, as it is not obvious that they are connected at all. Hendershott et al. (2011) state that algorithmic trading could either result in more competition in liquidity or have the opposite effect if they are used to demand liquidity. In fact, "AT could actually lead to an unproductive arms race, where liquidity suppliers and liquidity demanders both invest in better algorithms to try to take advantage of the other side, with measured liquidity the unintended victim" (Hendershott et al., 2011).

To establish some sort of causal relationship, Hendershott et al. (2011) studies an external event which only increases algorithmic trading of some stocks. Specifically, they look at the NYSE's autoquoting, which replaced specialists who previously handled the dissemination of quotes; this change was phased in non-uniformly, thus this can be used to identify causality (Hendershott et al., 2011).

Through their analysis, Hendershott et al. (2011) found that algorithmic trading does improve liquidity for large-cap stocks. This is shown by quoted spreads and effective spreads narrowing under autoquote, meaning a decline in adverse selection and consequently decrease in

price discovery associated with trades (Hendershott et al., 2011). Thus, algorithmic trading increases price discovery and implies quotes become more informative; however, this same instrument was less effective for smaller-cap stocks and no statistically significant effects were found (Hendershott et al., 2011). Finally, Hendershott et al. (2011) found that algorithmic trading increases measures of liquidity and supplier revenues which is surprising due to the expected mechanism for improving liquidity being competition between providers. The hypothesized explanation for this is that while algorithms only had temporary market power and over a longer time liquidity supplier revenues declined (Hendershott et al, 2011).

1.2.4 Costs of Algorithmic Trading

Domowitz et al. (2006) define algorithmic trading as "computer-based execution of equity orders via direct market-access channels, usually with the goal of meeting a particular benchmark." Included in this are smart routing, program trading, and rules-based trading. Like in the introduction, Domowitz et al. (2006) also recognize that literature in this topic area is diminished due to lack of available data. Thankfully, they took it upon themselves to bolster the research on algorithmic trading by looking into transactional costs of trading. Specifically, looking at 2.5 million shares consisting of almost ten billion shares from over forty institutions, with a portion traded algorithmically. This examination found that algorithmic trading is less expensive than alternative means (Domowitz et al., 2006).

To analyze the differences between types of trading, Domowitz et al. (2006) took algorithmic orders completed within a single day and compared them to randomly selected non-algorithmic trades. To compare the costs themselves, the Agency Cost Estimator (ACE) model

was used. This model doesn't associate a cost estimate for a trade; rather it views cost as a function of the strategy behind a trade, market conditions, and traits of the stock itself (Domowitz et al., 2006).

Through their analysis, Domowitz et al. (2006) found that on average, algorithmic orders generate costs of 14 bps. When compared to the control sample of non-algorithmic trades, there is a difference of 11 bps in performance. Across different algorithmic trading services, no difference in performance was found using the benchmark provided (Domowitz et al., 2006). After their analysis, Domowitz et al. (2006) conclude that algorithmic trading offers such advantages that most major brokers will offer these services; productivity is a major driver of this point, along with controlling costs and allowing more focus for difficult trades. The economic significance of algorithmic trades was also quantified, and a couple of key take-aways were given. Algorithmic trading is cost-effective regarding implementation shortfall and volume participation, algorithms are not sophisticated enough for large order sizes, and there is value considering other options in lieu of these imperfections.

Today, algorithmic trading is much more accessible and there has been a bit more literature on the topic over the years. However, it is still valuable to consider any downsides to algorithmic trading and lessons learned, particularly when trying to create an algorithm from scratch instead of utilizing preexisting services. While Domowitz et al. did a great job of analyzing extant algorithmic trading services against other methods, I hope to provide an effective way to create and test personal algorithms. Of course, some principles are still important to keep in mind, such as being wary of scaling an algorithm before fully testing all possibilities as well as relative costs to non-algorithmic methods. Although I will be mentioning

these points briefly, they will not be the focus of the research but could be a valuable avenue for further exploration.

1.2.5 Machine Learning

An incredibly popular avenue of algorithmic trading involves machine learning. Machine learning is a class of algorithms which can improve through experience and data and automate model building. The purpose of algorithmic trading is to make the local dynamic of traders more reliable and predictable. To this end, machine learning should reduce uncertainty even more through increasingly complex algorithms (Hilbert et al., 2020). In fact, Hilbert et al. (2020) have found that in foreign exchange markets, algorithms are associated with increasing predictability in bid-ask spread dynamics. However, this only holds true when looking at the market through a courser lens; upon looking at more fine-grained training, uncertainty has grown even larger (Hilbert et al., 2020).

1.3 Dark Pools

1.3.1 Introduction

Dark pools, as defined by Dizikes from MIT (2014), are “privately run stock markets that do not show participants’ orders to the public before trades happen.” They account for at least an eight of all trade volume inside the United States, and may even represent more (Dizikes, 2014). Despite being seemingly popular, dark pools are a controversial issue and one survey has shown

that 71% of finance professionals think dark pools are at least somewhat problematic in establishing stock prices (Dizikes, 2014).

Others say dark pools can aid price discovery. For example, one professor from MIT asserts that dark pools can help price discovery under the right circumstances. One example is attracting less-informed traders, which crowds out the better-informed ones and forces them back to public exchanges (Dizikes, 2014). Thus, there will be a higher concentration of traders on public exchanges who will contribute to price discovery. Unfortunately, literature on this secretive subject is scarce, just like with algorithmic trading, and the focus of this paper is on the algorithmic aspect so the review of dark pool literature will be shorter.

1.3.2 History

Dizikes (2014) states that dark pools are thought to have originated in the 1980s but have recently been gaining more popularity. In a study by the Securities and Exchange Commission in 2009, they estimated that 32 dark pools represent around 8% of trades; a couple year later in 2011, the Tabb Group and Rosenblatt Securities estimated that dark pools represent around 12% of trades, matching with the "eighth of trades" statistic earlier (Dizikes, 2014).

In "Do Dark Pools Harm Price Discovery?" Zhu (2014) discusses the market structure and regulatory framework for dark pools in the United States. Before 2005, they had a low market share and were primarily used to trade shares without revealing intentions to avoid being front run (Zhu, 2014). However, after the adoption and implementation of the Regulation National Market System (Reg NMS) new electronic trading centers were encouraged to compete

with existing ones. Many “dark venues” appeared in this period alongside the transparent “lit venues” exchanges (Zhu, 2014).

Besides regulations which increase competition, dark pools have gained market share due to other reasons as well. For example, in recent years there has been an increase of need to trade large amounts of shares secretly; in the meantime, the orders on exchanges have declined (Zhu, 2014). Another factor is that dark pools can offer price improvements over the best bid-ask spread on the exchange which attracts more investors (Zhu, 2014). A third factor is the ability for dark pools to match customer orders internally, without a need for trading fees on exchanges and other trading centers. This incentivizes broker-dealers to set up their own dark pools (Zhu, 2014).

1.3.3 Types

There are a couple of different types of dark pools which all derive execution prices from lit venues, albeit through different ways (Zhu, 2014). These types include classical dark pools, “midpoint” dark pools, bounded dark pools, and electronic market makers.

For reference, normal markets can be classified via two main important factors: trading mechanism and frequency of trading (Johnson, 2010). Trading mechanisms can be further divided into two different categories: quote-driven and order-driven. A quote-driven trading mechanism only allows traders to have transactions with a market maker who gives buy and sell price quotes (Johnson, 2010). On the other hand, an order-driven trading mechanism allows traders to all place orders on an order book, which are then matched using a consistent set of rules (Johnson, 2010). A major difference between the two mechanisms is how prices are set – in quote-driven markets, a trader can choose to negotiate a quote and then execute, thus having a

guaranteed execution at that price (Johnson, 2010). However, in order-driven markets, the best bid and sell prices are ostensibly based on the best respective orders available, and trades may only occur when a bid and sell match (Johnson, 2010).

Trading frequencies determine when matches turn into actual executions and can generally be split between three types – continuous, periodic, and request-driven (Johnson, 2010). As its name suggests, continuous trading is convenient and efficient but can lead to price volatility (Johnson, 2010). Periodic trading is normally scheduled, thus allowing for better price formation, whereas request-driven has the pest price formation but is less convenient and not as efficient while achieving this price (Johnson, 2010). Finally, combining the mechanisms and frequencies together we can get a table of differing market types.

Dark pools come into the picture when transparency is considered, which is a term used to represent the amount of available information in the market before and after a trade (Johnson, 2010). From the order types discussed above, order-driven markets tend to offer higher visibility (i.e., more transparency) because quote-driven markets only show the broker's best bid and offer (Johnson, 2010). A fully transparent market would imply an order book on full display – with no hidden volume – however, some, if not most, of these users might not prefer such transparency (Johnson, 2010). This leads into the appeal of dark pools. To reduce the impact of large orders on the market, institutional traders often prefer an anonymous order book and hidden orders (Johnson, 2010). Thus, opaque venues such as dark pools were created. However, these tend to be most successful when fair market prices can be determined from a visible market, as without that price discovery would be incredibly difficult (Johnson, 2010).

1.3.4 Price Discovery

The relationship between dark pools and price discovery is highly debated. The European Commission, International Organization of Securities Commissions, and various finance professionals all worry that dark pools could inhibit price discovery by hiding otherwise public trades (Zhu, 2014). Zhu (2014) investigates the validity of these worries through selections between "informed traders" who "hope to profit from proprietary information regarding the value of the traded asset" and "liquidity traders" who "wish to meet their idiosyncratic liquidity needs." A couple of other assumptions are made as well: traders will make optimal choices between public exchanges and dark pools, exchanges execute orders at the bid or ask and dark pools match orders at the midpoint of the exchange bid/ask, and the dark pool has no market makers to absorb excess order flow (Zhu, 2014). This means that dark pools are essentially a trade-off between improved prices and an order not executing (Zhu, 2014).

In an analysis of this framework, Zhu (2014) concludes that dark pools will in fact improve price discovery on average, although he concedes that they may occasionally mislead inferences of the asset value. Zhu (2014) also notes it is important to separate price discovery and liquidity, as they may not always have a proportional relationship; in fact, "More informative orders tend to worsen adverse selection on the exchange." Despite these concessions, because the matching of orders depends on availability of counterparties, larger orders will not get executed (Zhu, 2014). It is also important to note that more informed orders are correlated with larger ones, thus they will suffer from lower completion rates (Zhu, 2014). Zhu (2014) finally claims that this execution risk will push informed traders into the exchange while uninformed ones will

populate the dark pools implying that under natural conditions, dark pools increase price discovery.

Regardless of whether this framework is applicable in the real world, I think this was valuable to look at, especially for my research. A lot of concepts, such as information of the traders and trade sizing, are ones I will be using in the actual decision portion of the algorithm. Although I will not be trading in dark pools themselves, as I will be looking at dark pool trades and try to apply them within public exchanges, these ideas will still be important to keep in mind. Even if price discovery is negatively affected, if my algorithm works as intended it will enable people on the public exchange to leverage dark pools as well, thus negating some of the advantages that dark pools hold over more transparent venues.

Chapter 2

Code

2.1 Introduction

The process to create a working algorithm was unique and technically involved from a programming standpoint. To start, I was not able to find any historical dark pool data which means simply writing a couple lines of code to calculate an execution price point would not be sufficient. Instead, I had to go through a process of scraping live dark pool data, calculating optimal buy prices, forking processes to issue sell orders without interrupting the rest of the program, and finally logging the data.

2.2 Creating the Server

To start automation, a server is required to constantly run the algorithm. Although this would work on any computer, it is best that this be held on one that was able to run consistently and uninterrupted. This means that personal computers are often not the best choice, unless a dedicated computer could be set up. However, I did not have the resources available for that to be possible, thus I decided to host my algorithm on a server on the cloud.

The cloud computing service I ended up choosing was Amazon Web Services (AWS), which has a feature called Elastic Compute Cloud (EC2). EC2 allows clients to rent virtual computers to run applications and offers many tiers including a very simple windows virtual machine (VM) that could run continuously for free. The algorithm and surrounding program are

very lightweight; all they require are Python, an internet connection, and some dependencies for web scraping and multithreading.

Monitoring the VM can be done from any other computer with a series of steps. To connect to the VM, you must know the internet protocol (IP) address of the VM as well as have a password which Amazon will give to anyone holding a private key. The VM holds the public key, which can be used to encrypt a message (e.g., the password), which the private key holder can then decrypt. This extra cryptographic step is used for security, ensuring that nobody who is unauthorized – one who does not have access to the private key corresponding to the public key held by the VM – can have access. Although the safest way would be to hold my own server and not allow foreign inbound connections, security concerns were not too heavily considered for the purposes of this thesis; however, security of execution algorithms could be an interesting avenue for future exploration.

2.3 Scraping Data

Due to the lack of available historical dark pool data, I had to find a website to view this data in real time. After searching across many different financial websites and products, I settled on BlackBoxStocks which seemed like the cheapest option given the information I needed. Specifically, I was only searching for dark pool prints and blocks, which BlackBoxStocks outputs in an updating scrolling list format along with time of activity, ticker, type of activity (such as dark blocks, unusual options activities, etc.), and price per stock. Once you select a specific ticker, you can search for all opaque ticker-specific activity over the given day which is

important to be used in my execution algorithm. This includes time of order, expiration, strike price, type of option, total value of contract, and IV.

Web scraping extracts data from websites on the internet. In my case specifically, the only way I can automate the collection of dark pool data and perform calculations that would otherwise be impossible for a human in real time would be to collect data from the website using web scraping. For this thesis, I will be scraping the HyperText Markup Language (HTML) for relevant information. HTML, along with JavaScript and Cascading Style Sheets (CSS) are the building blocks of the internet and are standardized ways to define and display web pages. HTML performs the function of marking up pages (such as organizing and annotating images, text, and other information on the webpage). On the other hand, CSS assists HTML by adding stylistic elements to a webpage, whereas JavaScript adds scripting capabilities which can dynamically change the webpage based on certain events. The latter two are less important for this thesis but should be noted as general background information.

Most things you see on a webpage can be found within the HTML using the tag system – although it should be noted that some can be hidden from web scrapers using JavaScript, which isn't always available to the user. This is how I can automate a web scraper to find data within a webpage. Even though the HTML can be updated dynamically, this shows up in real time within the browser, so when the website receives new information on dark pool trades the HTML can be dynamically updated and extracted. Further research could be done in this area, though, to see the delay between the actual dark pool order and scraping the HTML; there probably is also delay between when the dark pool trade is executed to when it is displayed on the website for everyone to see.

To find the specific reference to certain elements, I manually stepped through the website in search of individual identifiers of the elements. Normally, HTML will be static on a webpage thus we only have to send a request to the server for the raw HTML and don't have to open an actual browser. Unfortunately, some of the data on the website is updated dynamically so passive scraping without a browser isn't possible. For example, as the table of dark pool data is updated, the rows shift down as the new rows of data appear at the top. Because of this, it is not possible to simply request the raw HTML from the server and I had to resort to having a bot constantly monitor the website as it is updated in real time. To do this, I decided to use the selenium, a tool which allows automation of browsers.

The last issue with automatically scraping data was unpredictable changes to the website that halted the data gathering process due to introducing unknown HTML elements. This was an unavoidable feature of the website I chose to get data from. One example of this is during some mornings, the website generates a popup window informing users about daily updates and news. The popup covered all existing elements, effectively hiding them in the background, thus causing the program to try to find a nonexistent element and fail in the process. This, in turn, lead to the program erroring out which stopped all data collection. Although this wouldn't be an issue if the automation wasn't reliant on having a browser open, to gather the core dark pool data I had to make the tradeoff of automating a browser instead of having browser-less scraping. Instead, as a temporary workaround I manually started the program every morning and closed the popup which would interfere with the rest of the data collection. Although this meant I could not have a perfectly automated program, this would be an area of future research.

2.4 Timing

As evidenced by the emphasis on creating a system to pull dark pool data, a lot of thought was put into the various time related aspects of the algorithm. This mainly occurred in three places – gathering data from the website to use in calculations, finding current execution prices and placing the "order" (in this case, logging the event as a paper trade), and when to start and stop the algorithm due to market hours.

The most obvious part timing comes into play is pulling the dark pool data. It was mentioned above that the program tries to pull the data as quickly as it arrives on the user-end; although, there might be some improvements to be made in efficiency given further analysis. After data is gathered, it is immediately used to calculate whether the ticker will be bought. For now, I only consider buying stock and selling it later, so no calculations go toward shorting a stock or buying options as that adds an extra layer of complexity into the calculation.

To find a buy price and subsequently a sell price, live data was pulled from TD Ameritrade (TDA). This broker was chosen mainly because TDA has a good application programming interface (API), which is simply software that offers services for other users to connect to and use. This interface governs all interactions the algorithm has with the New York Stock Exchange; without it, the algorithm would have no information on live market data and would instead be monitoring the performance of trades with a bit of delay. Other than simply querying current stock prices, the API allows much more control of a user's TDA account such as placing actual market orders with as much precision and detail you could have, as well as managing a portfolio and any other action you could do from the website. This will be important because I will discuss how to connect the algorithm to the market, so it performs more than paper trades, albeit with higher stakes.

To test the efficacy of the algorithm, only the current market price of the stock at the time of a buy or sell, as determined by the algorithm, is needed. Thus, limited use of the powerful API offered by TDA was utilized. To query the current stock price, special API calls needed to be made. These take the form of HyperText Transfer Protocols (HTTP) requests, which are protocols for fetching information from internet resources. It was briefly mentioned above that normally web scrapers could just fetch raw HTML pages and get information from there. Indeed, another way to implement this data collection which can be directly targeted is through HTTP requests. When a client wants to request certain information from a webpage, it will find the specific uniform resource identifier (URI) which effectively corresponds to the internet address of the server. Once this URI is identified, a specific type of request is generated. For querying the stock prices, only a GET type of HTTP request is needed.

A GET request is one of the simpler HTTP requests, only allowing fetching data from the server; thus, it cannot contain any data in the body of the message or affect any data on the server itself. It was briefly mentioned that using the API would allow a program to automate anything that could be achieved on the TDA website, but these requests are more complex and will be discussed in a later section. Using the GET request, the program is now able to query data from the source TDA servers, so the quotes should be fairly accurate with the current prices in the market. Note that this would be bottlenecked only by the speed of the client's internet connection, which will affect how quickly the request gets sent and received. For the purposes of this thesis, it's assumed that this connection would be consistent and fast, showing no noticeable disruption to the price quotes.

Unfortunately, there were a couple of issues with getting the quote prices that I discovered a bit later into the data collection process. It seems like there is a slight delay on

TDA's end, where the current market price for a stock, as displayed on something like Google or Yahoo Finance, would be a couple minutes ahead of the queried price received after the HTTP GET request. Although I wasn't sure exactly why this issue occurred, as that has a significant effect on any automated trading algorithm which queries prices through TDA's API. I investigated this issue slightly, but I didn't notice until after most of my data had already been collected so this could be an area of future exploration.

The last area related to timing was checking whether the market is open and closed. Because dark pool information can come in at hours where the market is not open to normal activity, it is important to make sure the algorithm doesn't log data or make paper trades while the market is closed. To do this, we simply must check the current time of the server, adjust the time to the NYSE, and pause any scraping of the website while the market is closed. There is one special thing to consider, which is that the algorithm must pause before the market closes for good so that the "sells" have time to execute as well. This is because I don't want to hold any stock overnight and don't want to have mid-frequency trades. To accomplish this, I will explain my sell simple strategy later.

2.5 Trading

The actual execution of the paper trade is straightforward. After the algorithm determines a stock is worth buying, it simply logs the price of the stock and the time it bought that stock. In turn, stock is sold by simply logging the price when a specific sell time is reached. Logging can be accomplished by sending data in real time to another server, or simply writing to a file and

sending that file over the internet. For this thesis, I chose to have all log entries written to a file, and have daily files emailed to me so the data is backed up and I can analyze it at will.

There is nothing that technically involved with the buy. After each new update in dark pool table on the website is scraped by the program, it is checked for whether it should be bought. If it is decided the stock is worth buying, a simple log entry is made in the file. However, the process of creating a sell entry is more difficult. The more complex issues arise when waiting for the stock to be sold because the program must keep running. However, it must also keep constant calculation of when to sell the stock, and a single program is not able to do two things at once. Especially when looking at even more stocks, it must somehow keep track of trying to buy new stock and when to sell the old ones. To balance all these considerations, multithreading must be used to spawn new processes to individually keep track of when to sell stock.

As the name suggests, normally a program only runs as one “thread” in the processor on a computer. However, if multithreading is applied then multiple threads can use processor resources at the same time. This concurrent utilization allows for a program to do more than one thing simultaneously, letting the algorithm continuously scrape the website looking for new stocks to buy while also keeping track of when to sell each stock. Specifically, in this program, a new thread will be created once each stock is determined to be bought a separate thread is spawned to determine when to sell that stock and the logs the relevant data. Thus, the only thing required by the “main” thread of the program is to continuously scrape the website and figure out which stocks to buy.

Logging the data doesn’t require as much finesse as buying and selling stock, but it does require some knowledge of other internet protocols. Writing the log file is not difficult, as all programming languages have built in functions to read and write files. However, sending this

data over the internet to be received in an email, as explained earlier, is more difficult. To do this, an email account is needed to send and receive emails. Furthermore, a Simple Mail Transfer Protocol (SMTP) server needs to be set up to format emails and create them. This can be done easily within Python using libraries such as email, smtplib, and ssl. This last library is important because at its core, SMTP is not a secure protocol. That means that any message sent over an SMTP channel can tampered with by external actors. To prevent this, another layer is needed on top of SMTP called the Secure Sockets Layer (SSL). The SSL layer establishes an encrypted link between the program and receiver, which the SMTP message (i.e., the actual email) can then be sent through. With all this set up, the log file can finally be constructed in a Comma Separated Value (CSV) format, for better readability, and sent out and received at the end of every day to be analyzed.

There is one assumption that is made when determining which stocks to buy – that the website will update the dark pool entry tables slowly enough that the program has enough time to scan the data and determine whether to buy the stock before the next entry comes in. If the program is in the middle of running calculations and more than one entry arrives, the program would only be able to get data from the newest arrival at the time of scraping. Through observation, this assumption and limitation should not have a large effect on the validity of the data and conclusions drawn. The minimum window for new dark pool activity seems to be on the order of a couple seconds, which is more than enough time for the program to run the calculations it needs. However, it is important to note that this is an assumption made for the purposes of the thesis.

Adding these fail safes to make sure no dark pool entries are missed is one possible avenue of future research. This would cause the above assumption to be unnecessary and allow

for a more perfect test of the actual algorithm; although, the assumption is reasonably sound in the first place. Another avenue of possible research is to further improve the efficiency of the code itself, which I didn't take too long to analyze and optimize because I was mainly interested in the collection of the dark pool data. Furthermore, a good future pursuit would be to save all the dark pool data as well, not just log the stocks picked out by the algorithm. This would allow for back testing and lead to the creation of better algorithms based on dark blocks in the future.

Chapter 3

Connecting to the Market

3.1 Introduction

Although connection to the market was not strictly necessary for the purposes of this thesis, it provided some valuable insight into some flaws such as the timing issues with TDA's API calls as discussed in the previous section. This also serves as a good proof of concept for future research, as well as provides a framework for scaling up the algorithm and having it interact with the actual market instead of just making paper trades. More will be discussed in the analysis of performance later, but our current assessment of the effectiveness of the algorithm may not show the exact same result in the real world.

To start, a TDA application needs to be created, along with a TDA account. The application serves a couple purposes. First, the application can be referenced by any computer with its identifier (if proper authentication is provided). The application then provides a way to utilize TDA's API and connect to a TDA user account, which can then be used to interact with the stock market. The creation process of such an application is simple. Instead of going on the proper TDA website, the developer website is needed. Here, a different account must be created which only has access to the developer tools and applications. Note that this account and website does not have any assets, nor can it perform any trades. This account is simply used to create applications as mentioned before, which can be used as a tool to connect code and trading accounts.

3.2 POST Request

Once the account is created and connected to the code, important information can now be sent and received from TDA. Previously, it was mentioned that HTTP GET requests can be used to query data such as getting stock quotes; however, these requests cannot contain any information in their bodies and thus cannot send data to TDA. To authenticate the application to the trading account, as well as to perform trades, the availability of requests containing sensitive data is required. HTTP POST requests provide this functionality.

POST requests send a request to a web server which asks the server to accept the data enclosed within the body of the request. This is starkly different from a GET, which simply cannot send data to the server in the first place. POST requests have a slightly differing anatomy, consisting of a header, and a payload consisting of the body, where the data to be sent will be stored, and other miscellaneous data fields. The header specifies things like what type of data is being sent, as well as any authentication being done. In the payload, things such as the destination URL (to send data to), extra parameters and data which won't be necessary for this program, and data of the specified format from the header can be added.

In the case of TDA's API, all data sent and received needs to be in JavaScript Object Notation (JSON) format. JSON is a standard file format that stores data in nested dictionaries – keys with associated values that are arrays. This is both easy to create and parse, especially for programming languages, so it is an efficient choice for sending and receiving data. Once the JSON is filled with the required values, the request can be sent through the API. A response is then received in JSON format as well. This response can then be parsed to check whether the POST request was received successfully, along with any data that is returned, or whether there was an error which occurred while the server is processing data.

HTTPS errors are common and occur all the time. Each error is associated with a different number which tells the program how to respond when one occurs. To start, the most common error codes are HTTP error 500 (Internal Server Error), 403 (Forbidden), 404 (Not Found), 400 (Bad Request), and 401 (Unauthorized). By far the most common errors that will be encountered by this program are 400 and 401. HTTP error 400 will appear when a trade is requested but there is something wrong with the order being placed. For example, when you try to buy a nonexistent stock or try to sell stock in which you own no shares. However, the request is still a valid HTTP request and thus the server itself will deny the request while processing it rather than not receiving the request in the first place. Similarly, for HTTP error 401, the server would have received the POST request but found that whichever client sent that request is not properly authenticated. These errors are important to handle and let fail gracefully; this way, the program doesn't crash when errors occur and can continue collecting data and executing trades.

3.3 Authentication

One of the most important concerns regarding allowing an external program to access a TDA trading account is security. This means that some sort of authentication step must occur to let TDA know that an account gives permission to a specific external program. The way TDA handles this is giving the user a couple of tokens. These tokens can then be included within the HTTP requests that the user sends to the server through the program. However, there are a couple of quirks with the authentication process which are important to cover.

The first step of authentication is to allow an application through the API to be trusted by a trading account. This can be done manually, which is the simplest way. This step does not need

to be automated because the first time an app is authorized, it will be issued both an authentication token and a refresh token. As suggested by its name, an authentication token can be attached to a request to tell a server that whomever sending the request is authenticated to make changes to the account, also specified in the request. If the authentication token wasn't issued by the account in question, authentication fails, and an HTTP error occurs.

Normally, authentication tokens will have a set lifetime. This is to ensure when a client uses an authentication token, they should still be allowed access at the time in the future. However, once one an authentication token is sent along with an HTTP request, there are a couple of security concerns including an unintended person getting access to the token. Thus, TDA places a thirty-minute timer on authentication tokens. The refresh token from when the accounts were linked manual comes into play here. This refresh token can be used to generate new authentication tokens once an authentication token becomes unusable. This will ensure that authentication tokens are fresh and solve the slight security risk and stale authentication issue from above.

For this thesis, a new authentication token is generated every time a request requiring proof of authentication is sent. This is to avoid the overhead of keeping track of previous authentication tokens to generate them on expiration. However, there is a tradeoff of slightly more time needed to send a request, as a new authentication token must be fetched. Although this will most likely not affect the efficiency of the algorithm, this is an area that can be researched in the future. Moreover, as mentioned above it will also avoid some other calculations so it might not have any negative effect in the first place. Furthermore, this sacrifice of efficiency for some security is common within many areas of computer science, and thus should not be thought of as an overall detriment.

3.4 Placing Orders

The first type of order to be placed is a buy order. This step will be straightforward and will occur concurrently with the logging of data for the paper trade. Once the algorithm determines whether a particular ticker is worth buying, it will start the process of sending an HTTP POST request. As mentioned above, the first step is to acquire a new authentication token. This is then embedded in the header of the POST request. Then, a couple of things are tested to determine if the stock should be traded. For this algorithm specifically, no actual trades with stocks below ten dollars were conducted. Although these stocks satisfy the criteria, there is a concern with penny stocks not having enough volume to fill the order before slippage occurs; thus, all stocks below ten dollars will not be traded. Another important consideration is adjusting for slippage for normal stocks as well. Placing simple market orders is not ideal, because, even with large-cap stocks, an order might not get filled at the price desired. Thus, a slight couple cents adjustment is used to make a limit-buy, ensuring that a stock is bought cheaply enough for the algorithm to prove effective.

A stop loss must be placed along with the buy order. This is to cut losses in the event the price on the stock ever drops drastically. A weird quirk with TDA is that this stop loss technically counts as a sell order, so it must be cancelled before the sell order can be placed. This occurs because the website thinks the stocks are in the process of being sold, thus they cannot be sold again in case the stop loss is executed. Therefore, the stop loss part of the buy must be cancelled if the buy went through. Here, HTTP error codes can be used to cancel these orders more effectively.

Before a sell order is placed, a cancel order will be placed to first try to cancel the buy order if it has not been filled. In this case, a sell order is ostensibly not needed as there are no

stocks to sell. In the case the buy order has been executed, an HTTP error 400 will be returned to the program, as TDA will have found no order to cancel. Then, the stop loss is available to be cancelled. Thus, the program must wait for an HTTP 400 error at this time. In most cases, the stop loss will be successfully cancelled and then the sell order should be executed on time. In the event the stop loss is triggered, another 400 error will be received from the server and then a sell order need not be sent. Finally, in the case of any other unexpected error codes the program should handle these failures gracefully and check which orders are currently active, then try the process again.

Once the sell order is required, it should not be too complex with all the other considerations made above. As explained in the previous section, multithreading is used to allow all sells to occur independently of the main program. This also enables feeding the sell order specific data from the time of the buy. Whereas, if threading was not used the program would also have to keep track of many different variables from each buy, leading to even more overhead. Another important aspect to keep in mind is slippage. Although for the purpose of this thesis only one share is bought meaning only one share can be sold, if more than one share is sold significantly affecting the total volume traded, the sell price might drop. The slippage concern for this program were less significant. An attempt to try and correct for slight market movements was made – the sell price in the actual order was adjusted by a couple cents so that the sell order fills within a reasonable price range, allowing the order to be filled while maintaining profits expected by the algorithm.

Chapter 4

Analysis of Performance

4.1 Introduction

Although the actual market ready algorithm was not run for long, it was interesting to see and gave some interesting insights into slippage and scalability, which will be discussed more later. Note that because the market ready algorithm was not run for long and does not have too much data, the rest of the analysis will focus on the algorithm performing paper trades. Another important thing of note in this analysis is only one share was "bought" or logged per stock that the algorithm decided to trade. Although this quantity can easily be scaled up by simply multiplying by a factor by the number of times shares should be bought, this is might not directly translate into the real world.

The first problem with simply scaling up is that the amount of money left over will be in constant flux, so calculations must be made for how much money is currently free to buy things. This calculation gets trickier after the timing of the buys and sells are considered, because at some points in time there might be a lot of trades executed where during other times there might be no trades executed. Although modeling this is possible, there is no need to go through such lengths for this thesis if the assumption is made that only one share can be traded per stock. This will practically ensure that there will always be available capital to buy a share, especially when applied to the live algorithm because if trading on margin is considered the account would start at 25,000 dollars and be allowed to trade as if it had 100,000 dollars (though 25,000 dollars would probably never be exceeded).

4.2 Price Charts

Considering the assumption, the proper analysis can be started. First, without any mathematical analysis, we can simply look at the price charts for SPY from the period 9/27/2021 to 2/2/2022, which is where the data collection of the algorithm started and ended for the purposes of this thesis. During this period, SPY started at $442.64 with a high at $477.71 and a low of $428.64 and ended this period at $457.35. On the other hand, from the assumption that my algorithm started with $25,000 to day trade with, the algorithm ended with $26,329.73 and had a low of $24,963.64 with a high of what it ended with. The more interesting part of this analysis is the graph of the price movement.

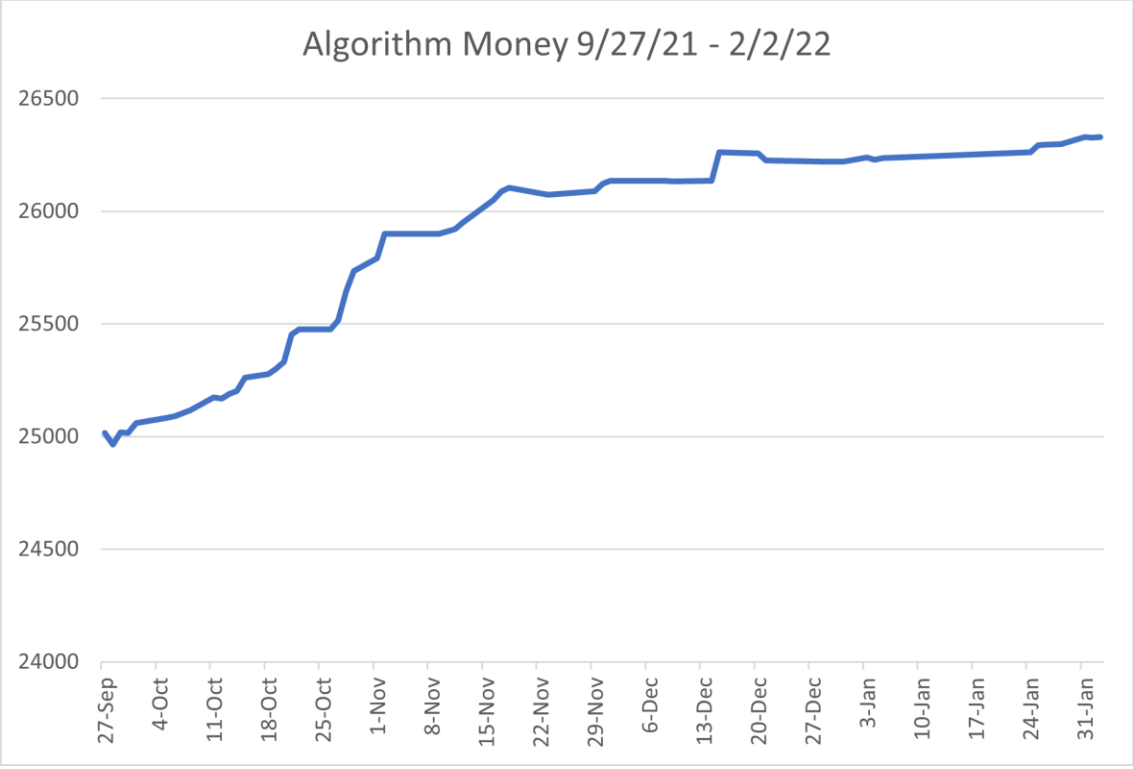

Figure 1. Graph of the total money after each day of running the algorithm from the period 9/27/2021 to 2/2/2022 with money in dollars on the left axis and date on the bottom axis.

Figure 2. Graph of SPY price movement from the period 9/27/2021 to 9/2/2022 with dollars on the right axis and date on the bottom axis, retrieved on 2/8/2022.

These two figures are especially interesting because we can see the constant flux of SPY, which ended ($457.35-$442.64)/$442.64 = 3.3% up, while at one point it was up at a high of ($477.71-$442.64)/$442.64 = 7.9% and was down at a low of ($428.64-$442.64)/$442.64 = 3.2% during the specified period. On the other hand, my algorithm had much less constantly flux, seemingly almost always increasing in value consistently. The algorithm ended up ($26,329.73-$25,000)/$25,000 = 5.3%, which was also its highest point, while it was only ever down at a low of ($24,963.64-$25,000)/$25,000 = .14%. Although at the highest point the algorithm did not perform as well as the highest point of SPY, it did beat out SPY over the five-month period. Another important thing to note is that as said above, this performance is predicated on only trading one share per stock. If we simply traded two shares, we could already see a doubling in performance to 10.6%. Although this may not scale up exactly as we trade more stocks, we can see that this algorithm is very promising.

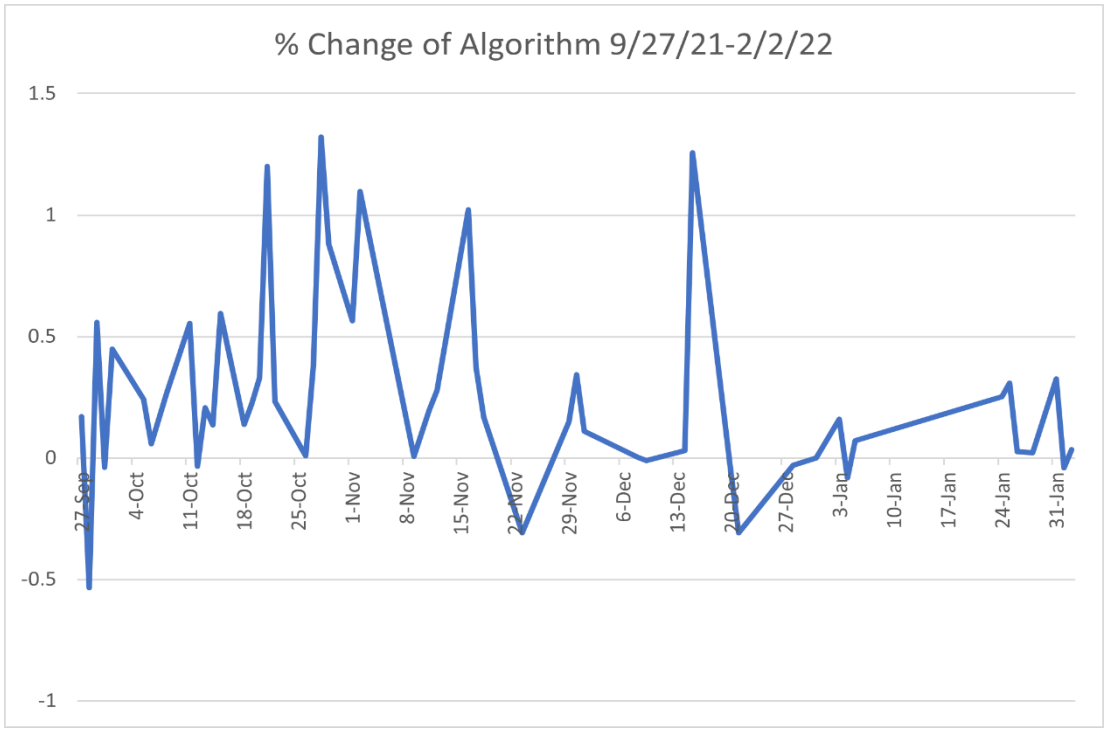

Figure 3. The day over day percent change of the algorithm from 9/27/2021 to 2/2/2022 with the percent value on the left axis and date on the bottom axis.

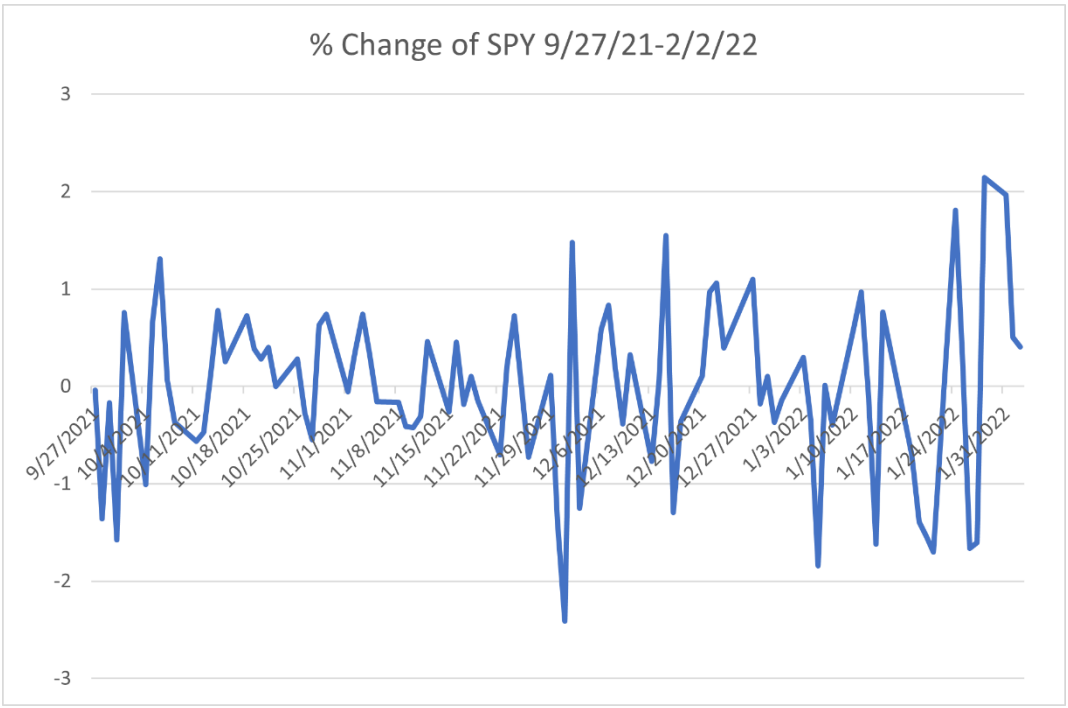

Figure 4. The day over day percent change of SPY 9/27/2021 to 2/2/2022 with the percent value on the left axis and date on the bottom axis.

Looking at the day over day percent change, we can see the trend of the algorithm usually making good trades and increasing in value every day. In Figure 3, we can see only a couple of days where the algorithm had a negative percent change, whereas in Figure 4 it is obvious that the SPY has a lot more fluctuations as well as larger fluctuations. The fluctuations in SPY look consistently seem to be within the bands of around +/- 2% whereas the fluctuations in the algorithm seem to be within a smaller band, only ever ranging between -.5 and +1.5%. However, most of the change in the algorithm seems to be above zero, showing general growth almost every day. One thing to note is that these percent change values would also be affected if we scaled up the trades, however this should only affect the magnitude of the perturbations, not the general trend of growth.

4.3 Risk Assessment

Although the growth looks good, some formal analysis needs to be done which shows whether the algorithm is performing will and worth the risk as opposed to investing in a risk-free asset.

To perform this assessment, several different metrics can be used, but we will be looking at a Sharpe ratio, which will compare the average return more than a risk-free rate in a portfolio, or in this case the algorithm. For the risk-free rate, I will be looking at the U.S. treasury yields, which are generally accepted as risk-free investments. Specifically, both the calculations with the one-year and two-year U.S. treasury yields will be shown and compared against the Sharpe ratio for SPY, which is 1.29. To make an actual conclusion on whether the excess returns are based on good investments versus risky ones, a higher Sharpe ratio would be expected. There are some

limitations to this metric – it operates under the assumption that returns are normally distributed, which is not necessarily true in the financial markets as returns are skewed away from the mean. Also, it assumes that both positive and negative price movements are equally risk, which is not necessarily the case. Despite these limitations, the Sharpe ratio still provides a useful metric and can give us tangible numbers to work with.

To start, we look at the one-year U.S. treasury yield which is .76% as of 2/2/2022, and the 2-year yield which was 1.16% as of the same date. The calculation of the standard deviation of excess return, telling us how much the algorithm's return deviates from the expected return, gives us a value of 3.88% by simply taking the standard deviation of the percent change, which is the excess return. For my data, I started at a value of $25,000 and ended at $26,329.73, giving us a return of ($26,329.73-$25,000)/$25,000 = 5.3%, which was also calculated above. Thus, we get our final calculations for the Sharpe ratio at (5.3-.76)/3.88 = 1.17 for the one-year U.S. treasury yield. For the two-year yield, we get a ratio of (5.3-1.16)/3.88 = 1.06.

From this formal analysis, we see that the algorithm indeed has a good risk-free assessment performance, with ratios well above 1 for the one-year yield while a slightly worse but still decent ratio for the two-year yield. However, both ratios are below the Sharpe ratio of SPY, which may be indicative that the risk of investment in the algorithm is greater than the risk of investment in SPY. Although this conclusion does not have to be mutually exclusive with a better raw performance from the price charts above, it is still curious why although the algorithm's performance is better it is deemed as a "riskier investment." This can most likely be explained through the drawbacks of a Sharpe ratio calculation mentioned above. Because it is based on the standard deviation of percent change, any high positive change will also have a large effect on the Sharpe ratio thus lowering it. This was mentioned briefly as the Sharpe ratio

doesn’t discriminate between positive and negative differences between expected return, and thus can look at a large positive difference and deem it risky.

Another important factor to keep in mind, which has been reiterated before, is that the algorithm currently only considers one share traded per stock. Taking an example used above, if this was assumed to be even one more share, making it two shares per trade, the Sharpe ratio would almost double because the percent return would double. Although this extreme growth might not be seen when scaling up and in a live test run, it still shows us that the algorithm has an incredible amount of potential and is most likely a safe way to beat the market.

Chapter 5

Conclusions

5.1 Future Research – Limitation, Efficiency, Scalability

Throughout this thesis, there were various limitations and areas of future research mentioned. These can be separated into areas of current limitations for the program, improving and testing the efficiency, and scaling up the program.

For current limitations of the program, it was mentioned previously that there are daily popup messages which prevented fully automating the program. This caused there to be some gaps in the days, so the program didn't collect data or trade for some days, at random, during the five-month period, simply because I forgot to start it in the morning. In the future, this issue would be a good avenue to start looking into. It is important to note that because this was random, it most likely didn't affect any conclusions drawn for this thesis. Another limitation was the lack of back testing methods available for dark pools based on the nonexistent historical data. Thus, documenting and saving all dark pool data would be a good avenue of future work to improve on similar research ideas.

Although no actual metric of efficiency was shown for the algorithm, and efficiency of code is hard to quantify, these could also be interesting areas for future improvements. There are a couple of immediate areas for checking efficiency that were identified in this thesis. In the web scraper, there might be a lot of small adjustments to be made, especially with the live-updating nature of the website used to collect data. Here, it could be important to analyze whether any data collection was missed and whether there is a delay between the actual dark pool order and when the algorithm picks up on it. Any timing issues with the TDA API should also be analyzed.

This includes querying the TDA website to get live prices and placing orders; these are crucial to get right if the algorithm goes live, so it is an important avenue of future research. The last issue is the tradeoff between security and efficiency, during which sometimes efficiency might have to be sacrificed – however, this may be worth it after careful future analysis.

Finally, as mentioned many times throughout the thesis, the algorithm only considered trading one share of a stock at a time. Scaling this up is crucial in the future, especially if any significant amount of money wants to be made from the algorithm. Depending on how carefully a large-scale algorithm is created will drastically change the performance metrics. Important things to consider in this case are current liquidity held which might make specific stocks bought at variable quantities. For example, only buying 5 AMZN shares as opposed to 70 T shares. Another consideration is slippage. This must be kept in mind especially for small-cap stocks, so special calculations should be made there. With that being said, if the algorithm proves to have good scalability, then as shown in the analysis, the returns generated might be multiple times more than the returns seen in this thesis, which already proved to be good.

5.2 Conclusion

Based on both the analysis of the price movements between SPY and the algorithm, as well as the Sharpe ratio, I can conclude that this algorithm which creates execution orders based on dark block data is indeed efficient and would be effective in an actual market. Although I was not able to create any hard numbers for the efficiency itself, as the calculations that go into that from the literature review are very complicated, it has beat the SPY in performance over the five-month period it was active and collecting data. Although no data was shown in this thesis, it is

useful to note that the live version of the algorithm also did make money however it had much more restrictions as explained, so was not included in the analysis. As for the riskiness of the investments made, it is shown that the algorithm should prove to be better than risk free, especially considering changes which would be made to implement it in the real world.

Altogether, this means that algorithmic trading in Python using dark pools is a viable way to be competitive against market index returns.

BIBLIOGRAPHY

Castura, Jeff & Litzenberger, Robert & Gorelick, Richard & Dwivedi Yogesh. (August 30, 2010). Market Efficiency and Microstructure Evolution in U.S. Equity Markets: A High-Frequency Perspective.

Dizikes, Peter. “How 'Dark Pools' Can Help Public Stock Markets.” *MIT News | Massachusetts Institute of Technology*, MIT, 4 Feb. 2014, https://news.mit.edu/2014/how-dark-pools-can-help-public-stock-markets-0203.

Domowitz, Ian & Yegerman, Henry. (2006). The Cost of Algorithmic Trading: A First Look at Comparative Performance. The Journal of Trading. 1. 33-42. 10.3905/jot.2006.609174.

Hendershott, Terrence & Jones, Charles M. & Menkveld, Albert J. (2011). Does Algorithmic Trading Improve Liquidity? The Journal of Finance. 66.

Hilbert, M., & Darmon, D. (2020). How Complexity and Uncertainty Grew with Algorithmic Trading. *Entropy*, *22*(5), 499. doi:10.3390/e22050499

Johnson, B. (2010). *Algorithmic trading & Dma: An introduction to direct access trading strategies*. 4Myeloma Press.

Zhu, Haoxiang. (March, 2014). Do Dark Pools Harm Price Discovery? *The Review of Financial Studies*, 27(3), 747-789.

ACADEMIC VITA

EDUCATION:

Pennsylvania State University, *Schreyer Honors College* Grad: May 2022

B.S. Computer Science, B.S. Mathematics

BIOVIA, San Diego CA June 2020 - Present

Software Engineering Intern, Pipeline Pilot Team

- Created new tests; improved accuracy, robustness between operating systems for existing ones using **Python**.
- Fixed an external XML entity injection vulnerability, in an Excel reader software component, using **Java**.
- Satisfied a user request by improving speed and parsing capabilities of a Jupyter component using **Python**.
- Developed under **Scrum** and used **Jira** for project management and **Perforce** for version control.

Qualcomm, San Diego CA

IT Intern, Cyber Security Microservices and Automation May-August 2020

- Developed and deployed a microservice for password auditing using **C++**, **Bash**, and **Docker/Kubernetes**.
- Delved into **Windows Active Directory** and **Linux** system internals to harden the host machine.
- Automated data gathering, processing, threat detection for critical IP access using **Python** and **Splunk ES**.
- Operated under a **Kanban** methodology, used **Jira** for project management and **Git** for version control.

Ultra-Communications, Inc., Vista, CA

Software Engineering Intern May 2019 - January 2020

- Wrote scripts and documented software in **Python** to help automate the quality assurance process.
- Trained DNN and K-Means in **Keras/Tensorflow** to recognize patterns within faulty parts and classify them.

Google Computer Science Summer Institute, CSUSM

Teacher's Assistant July-August 2019

- Mentored incoming CS students in web dev, app dev, good coding practices, and college in general.
- Educated students in **Javascript**, **HTML**/**CSS**, **Python**, and **Google App Engine**.
- Acted as a project manager for and guided a team which created a simple game-type web application.