New York City Taxicab
Transportation Demand Modeling for
the Analysis of Ridesharing
and Autonomous Taxi Systems

ANDREW JAMES TOULOUKIAN SWOBODA
ADVISOR: PROFESSOR ALAIN L. KORNHAUSER ’71

SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
BACHELOR OF SCIENCE IN ENGINEERING
DEPARTMENT OF OPERATIONS RESEARCH AND FINANCIAL ENGINEERING
PRINCETON UNIVERSITY

JUNE 2015
I hereby declare that I am the sole author of this thesis. I authorize Princeton University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Andrew James T. Swoboda

I further authorize Princeton University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Andrew James T. Swoboda
Abstract

A major contributor to the development of society is its capacity for rapid transportation. As this technological capability improves and spreads, the world shrinks and becomes increasingly accessible. Countries, governments, economies, and demographics all change as a result of a more efficient and connected society. The apparent fact that daily transportation systems appear to have plateaued in their expeditiousness begs the question of whether there are more efficient methods of transportation within the confines of the current system.

It is important to intelligently analyze the existing transportation network to determine manners in which automobiles can optimally coexist and commute. By implementing more sophisticated and efficient transportation systems on today’s existing network of roads, the United States will come closer to achieving various national and societal goals that have been in sight for decades: first, an improved national transportation system will reduce the amount of unnecessarily wasted fuel, shrinking both the country’s dependence on foreign oil and its carbon footprint; second, a more intelligent network will improve the quality of life of Americans, as it will decrease congestion and increase access to mobility as a result of lower costs for personal transportation; lastly, a “smarter” and decreased amount of congestion will result in a safer transportation experience.

Within the Operations Research and Financial Engineering department at Princeton University, Professor Alain Kornhauser has been conducting ongoing research to accurately model national transportation demand. Upon establishing an accurate model of daily trips, the feasibility and potential applications of ridesharing on a national scale can be determined in an attempt to reduce the total number of vehicle miles required to carry out everyday life. Unlike other solutions to congestion such as the lane-expansion of roads, ridesharing does not increase linearly with the number of additional individuals looking to commute and is therefore a much more sustainable form of transportation. Additionally, under the assumption that autonomous vehicles are to be road-ready in the near future, Kornhauser’s research looks at the feasibility of replacing personal transportation with an autonomous taxi (aTaxi) network that has vehicle stations placed in a dense grid layout, allowing individuals to walk to a nearby station and hop in a waiting vehicle.

The research that follows takes the national modeling problem out of a theoretical representation of demand to an actual subset of real-life transportation demand by considering the New York City Taxicab and Limousine Commission (TLC) trips taken during 2013. Modeling transportation demand using an actual, albeit smaller, data set instead of a synthetic data set, provides validation for, and a basis of extrapolation to, a national ridesharing model.

Using all recorded taxi trips taken in Manhattan and the surrounding boroughs of New York City, this thesis examines current trends and characteristics of the NYC taxicab service. It then investigates how various ridesharing systems, similar to that in Kornhauser’s research, would perform if they were implemented to replace the current network system. Furthermore, the research in this thesis lends itself to the possibility for what would happen if the existing NYC TLC fleet were replaced with autonomous taxis, anticipating that the current progress of autonomous car development will continue.
Acknowledgements

First and foremost I would like to thank my advisor, Professor Alain Kornhauser. Over the years we have known each other, he has served as an incredible teacher and mentor. His knowledge, passion, and rich curiosity have motivated, and will continue to motivate, my academic pursuits. Additionally, the help provided by Tsering Shawa invaluably progressed this research; without him, this thesis would not have been possible. I would also like to thank Jake Gao, Talal Mufti, and the innumerable students in past ORF 467 courses who, over the years, have helped progress transportation demand and ridesharing research within the ORFE department. Specifically I would like to acknowledge and thank a close friend and former classmate, Hill Wyrough, for the willing guidance and consultation he provided as it related to research, school, the workforce, and everything in between.

I would also like to thank my family. To Mom and Dad: I cannot begin to describe how much of an impact your love and support have had on my life up to this point. There is no manner in which I can truly show how grateful I am to always have you just a phone call or a plane ride away. To Matt: Observing the manner in which you lead your life always inspires me to push harder with mine. I could not possibly ask for a better brother, mentor, confidant, and best friend. Although our family has spread out geographically over the past four years, I have never felt closer with all of you.

Lastly, I must thank all of my friends who have contributed, in large or small ways, to my time at Princeton and made it such a valuable experience. To Scott: it has been an incredible four years, thank you for being a best friend. To Kristen: I can’t wait to continue our journey, one onomatopoeia and bananagram at a time. To all the members of the Squad: you have added, and will continue to add, so much to my life. Thank you.
Contents

Abstract ................................................................. ii
Acknowledgements ......................................................... iii
List of Tables ............................................................. vi
List of Figures ............................................................. vii

1 Introduction ............................................................ 1
1.1 Purpose ............................................................. 1
1.2 Transportation Demand Modeling within ORFE ....................... 2
1.3 Motivation ........................................................... 3
  1.3.1 Potential as Ridesharing Proof of Concept ......................... 5
  1.3.2 UberPool ......................................................... 8
  1.3.3 Self-Driving Cars ................................................ 10
  1.3.4 Replacement of NYC TLC Fleet with Autonomous Taxis .......... 12
1.4 Re-Focusing ......................................................... 13

2 2013 NYC TLC Data .................................................... 15
  2.1 Background on 2013 NYC TLC Data .................................. 15
  2.2 New York City TLC Service .......................................... 16
  2.3 Source Data Description ............................................. 17
  2.4 Source Data Alterations ............................................. 18

3 Assorted Analyses ...................................................... 21
  3.1 Taxicab Pickup Activity, by Week ..................................... 21
  3.2 Taxicab Trip Distance ............................................... 24
    3.2.1 By Borough ..................................................... 24
    3.2.2 By Daypart ..................................................... 26
  3.3 Taxicab Usage in 2013 and Impact of Precipitation .................. 27
  3.4 Airport Activity .................................................... 29
  3.5 Taxicab Fleet Analysis .............................................. 30
    3.5.1 Fleet Usage .................................................... 30
    3.5.2 Per-Medallion Information ..................................... 33
  3.6 Concluding Remarks ................................................. 34

4 Pixelization of NYC Trips ........................................... 36
  4.1 Overall Grid Structure ............................................. 36
  4.2 Yearly Pickup and Drop-off Trip Demand Distributions ............... 37

5 Ridesharing Introduction ........................................... 41
  5.1 Analysis Clarification and Terminology .............................. 42
  5.2 Methodology of Rideshare Departure Creation ....................... 43
    5.2.1 Geographic Boundaries ........................................ 45
    5.2.2 Common Destination Terminology ................................ 45
  5.3 Overall NYC Taxicab Inspection ..................................... 46
  5.4 Top-Five Pixel Investigation ....................................... 47
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Pixels with the Most Pickups</td>
<td>47</td>
</tr>
<tr>
<td>5.2</td>
<td>Pixels with the Most Drop-offs</td>
<td>47</td>
</tr>
<tr>
<td>6.1</td>
<td>((CD = 1)) Ridesharing Policy at Two Service Levels — Transit Hubs vs. All of NYC</td>
<td>56</td>
</tr>
<tr>
<td>6.2</td>
<td>((CD = 3, CIR = 0.2)) Ridesharing Policy at Two Service Levels</td>
<td>63</td>
</tr>
<tr>
<td>6.3</td>
<td>((CD = 3, dCIR = 0.2)) Ridesharing Policy at Two Service Levels</td>
<td>68</td>
</tr>
<tr>
<td>6.4</td>
<td>((CD = 3, dCIR = 0.1)) Ridesharing Policy at Two Service Levels</td>
<td>69</td>
</tr>
<tr>
<td>6.5</td>
<td>((CD = 5, dCIR = 0.1)) Ridesharing Policy at Two Service Levels</td>
<td>71</td>
</tr>
<tr>
<td>6.6</td>
<td>((CD = 5, dCIR = 0.2)) Ridesharing Policy at Two Service Levels</td>
<td>72</td>
</tr>
</tbody>
</table>
## List of Figures

1.3.1 New York City Visualization .................................................. 6
1.3.2 UberPool Ridesharing Concept ................................................. 9

2.2.1 Types of TLC Services .............................................................. 16
2.2.2 Yellow Taxi vs. Boro Taxi Service ............................................. 17

3.1.1 Per-Minute Trip Demand during 2013 by Weekday .......................... 22
3.2.1 Trip Lengths Split By Pickup Location ....................................... 25
3.2.2 Trip Lengths Split By Daypart .................................................. 27
3.3.1 All Trip Demand + Precipitation .............................................. 28
3.5.1 Per-Day View: Proportion of Fleet that is Active ......................... 31
3.5.2 Per-Minute View: Proportion of Fleet Generating Revenue .............. 32

4.2.1 Distribution of 2013 NYC TLC Trip Demand ................................ 38
4.2.2 3D Visualization of 2013 NYC TLC Pickup Demand ...................... 39
4.2.3 3D Visualization of 2013 NYC TLC Drop-off Points ..................... 40

5.2.1 MP300 Rideshare Framework Sample Execution ............................ 44
5.2.2 Visualization of Pixel Sizes .................................................... 45
5.3.1 Cumulative Distribution Function of Taxicab Pickups .................... 46
5.4.1 Pixel #51412 Pickup Analysis ................................................... 48
5.4.2 Pixel #51120 Pickup Analysis ................................................... 49
5.4.3 Pixel #52586 Pickup Analysis ................................................... 49
5.4.4 Pixel #49355 Pickup Analysis ................................................... 50
5.4.5 Pixel #51714 Pickup Analysis ................................................... 50

6.1.1 Major Transportation Hubs Visualization ................................... 53
6.2.1 Logic of $(CD = 1)$ Ridesharing Policy ...................................... 54
6.2.2 Taxicab Vehicle Mile Reduction ............................................... 57
6.2.3 Effectiveness of $(CD = 1)$ Policy Under Different Frameworks ........ 57
6.3.1 Scenario of Common Destination and Circuity Logic ....................... 60
6.3.2 Taxicab Vehicle Mile Reduction ............................................... 64
6.3.3 Effectiveness of $(CD = 3, CIR = 0.2)$ Policy Under Different Frameworks ... 64
6.4.1 Directional Circuity Logic ....................................................... 66
6.4.2 Taxicab Vehicle Mile Reduction ............................................... 67
6.4.3 Effectiveness of $(CD = 3, dCIR = 0.2)$ Policy Under Different Frameworks ... 68
6.5.1 Taxicab Vehicle Mile Reduction ............................................... 70
6.5.2 Effectiveness of $(CD = 3, dCIR = 0.1)$ Policy Under Different Frameworks ... 70
6.6.1 Taxicab Vehicle Mile Reduction ............................................... 73
6.6.2 Effectiveness of Policies Under Different Frameworks .................... 74

A.0.1 Self-Driving Car Advertisement from 20th Century ....................... 84
A.0.2 Average Number of Taxis Driving on the Road by Time of Day [10] ........ 84
A.0.3 Pixelization of Five Boroughs of Manhattan .................................. 85
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.0.4</td>
<td>Sampling of Original 2013 NYC TLC Trip Data</td>
<td>86</td>
</tr>
<tr>
<td>A.0.5</td>
<td>Sampling of Modified 2013 NYC TLC Trip Data</td>
<td>87</td>
</tr>
<tr>
<td>A.0.6</td>
<td>Per-Minute Trip Demand during 2013 by Week</td>
<td>88</td>
</tr>
<tr>
<td>A.0.7</td>
<td>Daily Weather Plots (Monday – Thursday)</td>
<td>89</td>
</tr>
<tr>
<td>A.0.8</td>
<td>Daily Weather Plots (Friday – Sunday)</td>
<td>90</td>
</tr>
<tr>
<td>A.0.9</td>
<td>Per-Medallion Trip Distribution</td>
<td>90</td>
</tr>
<tr>
<td>A.0.10</td>
<td>Cumulative Distribution Function of Taxicab Pick-ups (for All Pixels)</td>
<td>91</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Transportation at its best simply consists of where an individual is, where that individual wants to go, and when that individual wants to do so. Once this information is determined, sophisticated engineering systems handle everything in between to most efficiently satisfy that transportation demand.

A wide range of transportation modes have come and gone over past centuries, yet one in particular — aligned with the proliferation of infrastructure in the United States — will likely not disappear for an extended period of time. The automobile and its extensive network of roads, highways, bridges, and tunnels are here to stay. As the demand for personal rapid transit, as well as the level of congestion, continue to increase at an alarming rate [1], intelligent advancements in the ways we use the current transportation network are vital to the continued improvement of daily efficiency, national energy consumption, carbon emissions, quality of life, and overall safety.

It is important to note that throughout this thesis when talking about concepts related to “removing congestion” or reducing the number of vehicle miles driven, it is implied that the motivation for doing so is for all of the improvements listed above and more, not solely to ameliorate traffic.

1.1 Purpose

The goal of this thesis is to explore the feasibility of a ridesharing model within New York City, using the NYC TLC trip data as a proxy for all personal, vehicular transportation demand. In addition to initial investigations of the 2013 trip data, the research goes on to imagine the impact of different ridesharing implementations on future NYC transportation systems while also calling attention to the viability of autonomous taxi usage.
In this first chapter, this thesis provides a general introduction to the concepts of transportation demand modeling, ridesharing, and self-driving cars. The second chapter gives a more detailed background of the taxicab system that exists in New York City and of the data used in this research. Chapter 3 provides an understanding of current taxicab demand, trends, and other statistics. Next, Chapter 4 prepares for the subsequent chapters by explaining and visualizing the overall pixelization of New York City and its taxicab activity. Chapter 5 lays the ground work for all of the ridesharing research that was conducted, while Chapter 6 dives into more depth and explains the methodology behind, and results of, various ridesharing policies and simulations. Lastly, Chapter 7 looks to address limitations that were encountered throughout this research and provides possible next steps.

1.2 Transportation Demand Modeling within ORFE

For many years, transportation demand modeling within the Operations Research and Financial Engineering department at Princeton University has been conducted and facilitated by Professor Alain Kornhauser. Talal Mufti, Jingkang Gao, Alexander Hill Wyrough, and many members of past ORF 467 classes have contributed to Professor Kornhauser’s effort to create a comprehensive transportation demand model for the United States. Again, the completion of such a model would result in the possibility for a myriad of next-generation transportation system analyses — most pressing of which includes the potential for ridesharing amongst strangers and an effective autonomous taxi (aTaxi) network.

In the most recent significant piece of work, Hill Wyrough produced a daily transportation demand model for the entire United States. The model drew upon data from the 2010 U.S. Census, the 2010 American Community Survey, and other sources to simulate 308,745,538 individuals, their specific personal attributes, and the 1,009,322,835 automotive trips taken throughout the country on an average work day [2]. Wyrough developed a model that created synthetic trips based on probability distributions and data collected on aggregate bases — a model that is not accurate at the person-by-person level, but is accurate overall and on average. Such a complete model is the first of its kind, and must be created synthetically as it is unrealistic to assume data of such detail can feasibly be collected in its entirety.

The data set created by Wyrough is a first run, and multiple iterations through his data-creation process will inevitably result in the elimination of bold assumptions and potential mis-steps. However, to effectively argue that more advanced transportation systems can be realized on the existing web of roads, one must provide multiple pieces of compelling proof – particularly proof grounded in
This thesis focuses on serving as the next piece of an academic puzzle that ideally will provide convincing evidence that the current transportation system is in need of improvement and that the most practical enhancements involve the implementation of a thorough ridesharing system. The best way to create a compelling argument is to use real, historical data. Out of necessity, research in past theses have been conducted by creating synthetic data modeled after census and workplace data \cite{2} \cite{3} \cite{4}. In order to take a step closer to reality, this thesis considers actual data from a hyper-congested subset of the country.

In response to a Freedom of Information Law (FOIL) request, the New York City Taxicab and Limousine Commission recently released a comprehensive data set of every trip logged by every TLC medallion taxicab during the 2013 calendar year — the first significant data release of its kind by the NYC TLC. After removing discrepancies in the data set, it consists of 168,779,842 records, which belong to 42,821 unique drivers and 13,741 unique taxicabs. For the purposes of this thesis, the individual driver and taxicab identifications were not critical; the most valuable pieces of data consisted of the exact date and time of the trips, the exact geographic locations of pickups and drop-offs, the number of passengers, and the vehicle miles of each trip. The sheer detail of this real-life NYC TLC data set allows it to serve as a quality test case for analyzing the effect of implementing a ridesharing system and observing the potential for a future aTaxi network.

It is unquestionable that the New York City taxicab network is unique when compared to other taxicab systems throughout the nation. No city is as dense as NYC, and no city has as thorough of a street-hailing capability for the vehicles \cite{5}. Nevertheless, determining potential ridesharing systems to implement in New York City furthers progress and results in interesting insights being drawn about NYC transportation and about necessary characteristics of such a next-generation system for it to be effective.

1.3 Motivation

In a time where there are more than 1.2 billion vehicles on the road throughout the world and 4.09 million miles of road in the United States, it is unlikely that any new transportation system, no matter how revolutionary, will quickly or completely replace the expansive automobile system currently in place \cite{6}; unless this new transportation system repurposes or more intelligently uses the existing infrastructure. So to further progress the nation’s existing transportation system, one must examine its current state and consider how effective various changes could be. By considering
the 173 Million TLC trips taken during the 2013 calendar year, this thesis simplifies the overall transportation ecosystem and uses real, as versus synthetic, data with the assumption that identified potential improvements to the NYC TLC system can be extrapolated, in part or in whole, to broader transportation systems.

Dynamic ridesharing is the concept of arranging one-time shared rides between strangers, with very short notice, where the driver is a commercial operator rather than another passenger. This type of ridesharing is not a novel concept; rather it is merely an extension of the traditional style of carpooling that has become more realistic and popular due to the availability of new technologies. Throughout this thesis, the term “correlated ridesharing” will refer to any carpooling that involves individuals who know each other personally and have planned to rideshare ahead of time — for example, two siblings leaving an apartment together and heading to a restaurant. On the other hand, “casual ridesharing” will refer to this newer-age concept of dynamic ridesharing — an individual hailing a cab to go to a restaurant and then pairing up with a stranger who happens to be going to the same place, or the immediate surrounding area.

Internet enabled technologies — mobile GPS units, real-time public transportation information, live traffic reports on almost all public roads, mobile payment systems, and more — are allowing for a more pervasive and interconnected network that gives individuals the power to most effectively plan their mobility [7]. Yet, in addition to the literal benefits they provide, the existence of these technologies has shifted the expectations of the masses. As individuals in the modern world are more accustomed to instant and free gratification, these tendencies translate to the desire for immediate gratification in their transportation. More so than ever, individuals expect and demand to travel from point A to point B as quickly and as cheaply as possible.

Start-up technology companies such as Uber and Lyft have begun the process of commercializing these new-age internet enabled technologies in the public transportation sector. The companies capitalize on the GPS capability of smart phones in two major ways: first, registered drivers are provided a company iPhone that is programmed to solely function as their GPS unit; then, users locate, hail, and track drivers using the GPS function on their personal smartphones. Mobile payment systems are the other major feature of these services, requiring that users add a working payment card to their profile prior to hailing drivers so a seamless exit can be made as soon as the destination is reached [8]. New-age thinking from these types of companies has begun to push the boundaries of the existing transportation system.
1.3.1 Potential as Ridesharing Proof of Concept

To effectively demonstrate the potential for ridesharing, certain challenges — including sufficient population density, driver motivation, and passenger motivation — must be overcome. Motivation results when perceived benefit exceeds perceived cost. Although a driver’s cost-benefit conclusion is driven predominantly by revenue, a passenger’s conclusion concerns inconvenience, inefficiency, and safety in addition to cost. The earlier-described, FOIL-requested NYC TLC 2013 data is ideally suited to provide a ridesharing proof of concept that addresses these challenges a ridesharing implementation could face.

With New York City being one of the most densely populated urban areas in the United States [5], it is reasonable to assume that the city is in need of next-generation transportation improvements. Being such an established city, NYC would struggle to undergo a complete renovation of its existing infrastructure, so new systems will have to fit within the confines of its current transportation network. The effective implementation of dynamic, or casual, ridesharing is an example of a capable, next-generation concept. While this efficient method of transportation would ideally occur throughout the United States, a realistic analysis of its implementation in a compact and highly publicized area like New York City would begin to demonstrate its usefulness to individuals around the nation. The existing NYC TLC serves transportation demand within the five boroughs of New York City, but as Manhattan is such a dense urban area, one could expect that a concept of dynamic ridesharing would be most effective in the heart of the city. The five boroughs are depicted in Figure 1.3.1(a).

As a preview of the level of overlap between passenger trips — and therefore the potential for ridesharing — consider a 0.1-by-0.1 mile square centered outside of New York Penn Station’s 8th Avenue entrance. Over the course of 2013, this small region was home to 1,155,367 unique taxicab pickups. A three dimensional heat map of taxicab pickups throughout the year is depicted in Figure 1.3.1(b) as well as later in the report; Penn Station’s high taxicab demand can be seen poking out of the skyscraper-bars in Midtown Manhattan. More statistics about this location, and others, will be provided throughout the remainder of this thesis in support of ridesharing as a real and viable transportation system innovation.

This thesis looks at the technical feasibility and effect of ridesharing within the NYC TLC network, however it is important to note that this feasibility is theoretical. Once an ideal transportation system concept is devised, to practically implement it in the real world one must consider how individuals would adopt and react to it.

As previously mentioned, there has been a rapid development of internet enabled technologies
relevant to the transportation sector. With handheld devices able to access the internet, pinpoint locations on a map, find fastest routes from point A to point B, and more, the physical implementation of a convenient and efficient dynamic ridesharing system from a driver-side perspective is more than feasible. But, there needs to be a convincing adoption of such a system by all parties involved to realize ridesharing as integral to future transportation systems. “The question is not [the] technology,” the co-president of the Belgian Transport Union mentions, “the discussion is [whether] you want to have an organized taxi sector” [9].

The creation of an on-the-fly transportation management system would rely on the use of a mobile application. This application would have two major subsets of users: drivers and passengers. From the drivers’ perspective, all that would truly be needed to entice them to actively engage in the system would simply be some form of financial incentive to prefer carpooling customers to a single customer. One way, for example, would be to require that ridesharers pay a fraction of their original fare, but in such a way that the sum of all of the ridesharers proportions is greater than 100%. This way a driver would make more money through ridesharing than single rider pickups, while the time needed to complete the rideshare would not significantly differ by nature of the fact that an implemented ridesharing system would be effective in matching passengers and their destinations.

The adoption by users, from a passenger perspective, is more of a challenge. First, as with
drivers, a financial incentive must exist for many passengers. Though these individuals may not be willing to share a ride at the same cost as a solo trip, paying 60% of an original fare, for example, could likely be motivating enough for a passenger to share a ride. Even then, there is no point in implementing a new idea unless a base of users exists that is able and willing to adopt it. This is more pronounced when the idea revolves around sharing with others and is subject to the network effect — it requires a large pool of customers to be effective. Younger audiences are more prone to embrace new and disruptive technologies in their daily lives; they also share a mentality that caters toward sharing due to their use of major social networking platforms. Both of these characteristics mitigate concerns over convenience, inefficiency as well as safety. Younger individuals are also more likely to be hailing a cab in New York: more than two-thirds of NYC TLC passengers are 35 or under, with 35% of passengers reporting being younger than 21 and 35% reporting between the ages 21 and 35 [10].

Again, for such an on-the-fly transportation management system to work, it would need to rely on mobile technology such as a smart phone. An in-taxi survey conducted in 2012 and 2013 by the NYC TLC indicated that two-thirds of passengers use a smart phone. Furthermore, 55% of passengers indicated that they “would like the option of using their phone to locate taxicabs, and 54% say they would pay for their rides with their phone” if both of these options were a possibility [10]. As the popularity of Uber, and other mobile-technology-reliant ridesharing companies, has surged since 2012, it can be safely assumed that the actual proportion of taxi-riding individuals using and willing to incorporate mobile technology into normal taxicab usage has experienced a similar escalation. So if the NYC TLC were to develop an application for a ridesharing service, the market for it certainly exists; however the TLC would still need to determine how to market the application in order to spur a significant volume of downloads and frequent users.

In fact, the Taxi & Limousine Commission has already begun to look into the possibility of Taxi-Hail smartphone applications [11], the possibilities of which recently made headlines. On January 16, 2015, the Taxi Workers Alliance held a panel of union leaders to discuss major improvements to traditional taxi networks. Bhairavi Desai, the leader of the union’s New York chapter, and her counterparts are in the process of cultivating widespread adoption by creating “their own app, backed by taxi unions, that would work only with licensed drivers and wouldn’t [send] a chunk of their profits to investors” [9]. According to a member of the American Federation of Labor and Congress of Industrial Organizations, who has been working closely with taxi drivers in Maryland, the “convenience of [an] app, the ability to go from point A to point B on your cell phone” is really all that is limiting traditional taxicab services from competing at a significantly higher level [9].
New York City taxicabs are not the first to show signs of progress towards releasing a smartphone application that harnesses new technologies to enable the concept of e-hailing — the event of hailing a cab electronically. Cab companies in Brussels already have applications to seamlessly hail cabs and pay for rides; Washington D.C. will soon require cabs to use an industry-managed program; and other cities are currently in the process of designing their own applications [9]. It is clear that the technological capabilities required for enabling the necessary routing, matching, and networking of ridesharing already exist.

1.3.2 UberPool

In the second half of 2014, roughly five years after Uber’s inception and its accumulation of a user base, the ridesharing company introduced a next-generation twist on its original product — UberPool. This new carpooling option allowed passengers to mark their preference for shared Uber rides through the existing mobile application. Although the service inherently depends on the trip demand of others in the nearby area and thus does not guarantee a match, UberPool promises a significant decrease in fare price if one is found. In December 2014, the service was officially released for public use in New York City [12].

With UberPool, the Company is branching out to reap the benefits of the untapped casual-ridesharing market, especially in dense urban areas like New York City. A service similar to the omniscient ridesharing model simulated in this thesis for the NYC TLC service, UberPool requires users to enter their pickup and drop-off location. This means the Company has a real-time collection of all of the trips en-route as well as the current trip demand. The ability to apply Dijkstra’s shortest path algorithm to the current subset of trips, with limitations on matching so as to not dramatically reduce convenience to existing passengers, is all that is really needed to create an effective online ridesharing system.

However unlike the model in this thesis, Uber is willing to alter an individual’s route, even mid-route, in order to find a carpooling match. Figure 1.3.2 hints at how their casual ridesharing concept works. This thesis, on the other hand, determines a finalized route from origin to destination (or a set of destinations) before departure. Among other differences, this allows for a more predictable travel time of all passengers involved. Figures 6.2.1 and 6.3.1, later in this report, show similar diagrams of how the casual ridesharing, as proposed by this thesis, works.

The fact that Uber’s service generated a significant amount of interest, despite being live for a short amount of time, is another point of proof that the public possesses a tangible demand for next
generation transportation options. With promises of a “win-win-win scenario [where] passengers can save up to 40% on their fares, drivers make 20% more for the same trip, and the ridesharing model takes excess cars off the streets”, Uber’s carpooling service has been catching the attention of the public — of individuals who remain reliant on the old-fashioned technique of standing in the street to hail a cab, of individuals who want a cheaper ride, and of individuals who are concerned about decongesting American streets [13].

UberPool is also drawing attention to the gross inefficiencies in today’s standard taxicab service. By sensibly combining passengers trips into a rideshare, the service is not only increasing the amount of revenue for a driver and decreasing the cost to the rider, it is also dramatically reducing the amount of unpaid travel time and vehicle miles that taxicab drivers must otherwise experience. In the example of Figure 1.3.2, in order to serve the same customers with no ridesharing, the driver would have to first drop off passenger 1, back track to pick up passenger 2, then turn around again to head toward the destination of passenger 2, then back track one more time to pick up the third passenger. If new passenger demand exists that is both spatially and temporally fitting, it makes sense to meet it while still serving the original demand.

The continued growth of UberPool’s user base serves as a testament to the potential for casual ridesharing concepts amongst the NYC TLC. However unlike the young start-up company, the NYC TLC possesses possibly the largest and most established taxicab user base in the country. Furthermore, yellow taxicabs are an iconic aspect of one of the most well-known cities in the world. An effective ridesharing addition to such a notable service would help permanently revolutionize the concept of efficient transportation. Until such a system upgrade takes place, UberPool can serve as a very credible real-world case study of a similar, NYC TLC-backed service. In particular, viewing how customers adopt the ridesharing aspect of Uber’s taxi service will give insights into the willingness, or even eagerness, of the public to share rides.
1.3.3 Self-Driving Cars

The first automobile was created in the 1700s, the first Ford Model-T was released in 1908, and since at least the 1930s, self-driving cars have been conceptualized [14][15]. In fact in the 1950’s, long before the development of crucial technological components necessary for autonomous driving, advertisements were already portraying a promising future where passengers could relax and focus their attention on other matters (Figure A.0.1 on page 84 in Appendix) [16]. In addition to the luxury of allowing motorists to divert their attention from driving and from mind-numbing traffic, self-driving cars promise an assortment of benefits, both directly and indirectly.

Improvements on a traditional car’s capabilities, have historically helped save passengers from fatal crashes; but these have recently plateaued. Now, in order to significantly decrease the 42,000 deaths and 2.7 million injuries each year in the United States, “vehicles must avoid crashes rather than attempt to survive them” [17]. Self-driving cars would do just that, as the National Highway Traffic Safety Administration recently found that 93% of deaths are attributed to human error [18]. Furthermore, as more self-driving cars appear on the road, they can begin to communicate with each other and more effectively drive in an interconnected network of vehicles. Such a network would result in less stop-and-go congestion and fewer accidents. An autonomous car would also be able to tune its acceleration, deceleration, and cruising speeds in order to reduce the amount of wasted fuel and pollution that comes as a result of erratic and emotional driving.

Further improvements resulting from autonomy technology are iterative. As with any innovation, the capabilities and features only improve as the number of iterations and real-life examples increase. Over time, as the technology becomes ubiquitous, the physical car structure can be reduced in mass as “much of that mass is devoted to protecting occupants” [17]. Some benefits associated with exchanging vehicle mass for more vehicle intelligence include increased mileage range, decreased fuel consumption, lower construction costs, and more.

As with any major transformation of a currently-accepted system, there is a bigger hurdle than the actual technological developments and implementation surrounding self-driving cars. It is both emotionally and legally challenging for a car manufacturer to ask that individuals relinquish all control of their vehicle and allow a combination of sensors and algorithms to transport them at high speeds. “But by building [self-driving car features] step by step, you build people’s confidence” in autonomous vehicles [19].

Handing over complete control to a computer is certainly a frightening feeling for most, especially when the technology retains a lingering impression of being underdeveloped. Only 11 years ago, the
most technologically capable self-driving vehicles attempted to complete the 2004 DARPA Grand Challenge: a 150-mile closed course. “The best any of them could do was 7.32 miles—and that vehicle got stuck and caught fire” [20]. Such a substantial showcase of ineptitude sticks in peoples’ minds longer than the actual inability exists. Just a year after the 2004 drama, five vehicles completed DARPA’s 132-mile course [20]. Since then, the work behind autonomous automobile transportation—an idea so tantalizingly conveyed in those ambitious advertisements of the mid-1990s—has erupted with progress alongside a digital revolution taking place across industries.

Today’s autonomous vehicle environment has progressed to a much higher tier than that of earlier days. To list a few accomplishments: Google’s self-driving cars have logged more than 500,000 miles of autonomous driving [16]; Audi’s self-piloting A7 drove 500 miles from Palo Alto to Las Vegas for the 2015 Consumer Electronics Show [19]; and just earlier this April, an autonomous car drove from San Francisco to New York City by driving 3,400 miles in nine days [20]. As a professor at USC who is involved with autonomous driving points out, what is fascinating about this current autonomous vehicle environment is not that such feats are possible, but rather that they are “so easy” [20].

It would be unwise to ignore the direction that the automotive industry is headed. The Mercedes Benz S-class on the market, for example, “can already automatically accelerate, brake, and stay in the same lane under certain circumstances” [21]. Most other major car manufacturers—Audi, Nissan, Volvo, Ford—have begun to work on autonomous technology [20]. Even more manufacturers already implement active-assist features in units currently being sold. Present-day features available on the market today include: lane-departure warning systems, collision-warning systems, self parking, blind spot detection, cruise control, auto-climate control, driver recognition, and more [22].

This all being said, there are bridges that have yet to be crossed as self-driving cars continue to progress, especially in non-technical aspects. This thesis calls attention to two in particular. First, there is the fundamental truth that not all individuals in the nation desire to passively relinquish the act of driving for some other purpose—it would not be unreasonable to say that some find the process of driving to be therapeutic. Such a personal feeling need not quell progress; instead, it could steer development in a manner such that drivers are able to indicate when they are in the mood to drive versus be driven. Second, the legal aspects of self-driving cars have yet to be carefully presented. Individual state governments have yet to settle on what all an autonomous vehicle is and is not allowed to do on their roads, or on exactly how passive a passenger can be while a vehicle is in self-driving mode. Potentially the biggest cause of complications comes with questions such as, “who will be at fault in the likely event that autonomous cars are imperfect and get into accidents?” or “who would be held responsible if a robot car indeed ran a red light?” [23]. Though this may be
an uneasy topic for the time being, no true innovations in the world were created with legal terms already in place. As California Governor Jerry Brown responded after signing a bill that paved the way for self-driving cars within the West Coast state, “we will work that out’ [...] ‘that will be the easiest thing to work out’” [23]. Most often, technological breakthroughs come first, with legal boundaries being laid down shortly after.

Each innovation brings us closer to a society that will be able to seamlessly assimilate self-driving cars amongst human traffic. Google, an expert company at the frontier of autonomous driving technology, expects to be capable of placing fully autonomous vehicles on the roads by 2020. More traditional automotive manufacturers — focused on producing a mass-market self-driving car at affordable prices — expect to have units available by 2025 [21].

1.3.4 Replacement of NYC TLC Fleet with Autonomous Taxis

Taxi Driver Shift Dilemma

Upon the official public release of self-driving cars, one might anticipate that the next course of action would be to exploit their technology beyond personal use and create a taxi system capable of autonomously picking up and dropping off customers. If having a car without a driver is a possibility, utilizing these self-driving cars for 24/7 taxi service makes full use of their benefit, rather than subjecting them to spend most of their lifetime parked in a garage.

Even an “around the clock” service like the New York City TLC is not a genuinely consistent provider of empty taxicabs for passengers. Taxicab availability fluctuates day-by-day and hour-by-hour over the course of each day. Currently, in order to maximize the amount of action that a medallion-taxi can receive on a given day, owners commonly assign two drivers to each vehicle throughout the day. These drivers split shifts — an AM shift and PM shift — that each typically last about 9.5 hours. Traditionally, the AM shift begins around 6:30 in the morning and the switch to the PM shift occurs between 4 PM and 5 PM each day. As seen in Figure A.0.2 on page 84, each day sees two maximums and minimums in the number of taxis on the road looking to serve customers. Later plots, such as Figure 3.1.1, show how this exchange of drivers results in a sharp decline in the number of taxicab pickups; it is presumed that this drop is due to the lack of available taxicabs during the shift change rather than a drop in taxicab demand, resulting in unsatisfied customers.

If the NYC TLC fleet of taxicabs were to be replaced by self-driving cars, the autonomous taxi fleet’s availability would not fluctuate to the same extent that a fleet of non-autonomous cars does. Taxis would not require daily shift changes, personal breaks for the driver, and more. In a longer-
term perspective, the use of autonomous vehicles as taxicabs would result in individual vehicles being able to cover more demand on an average day, and the number of necessary vehicles on the already-congested NYC roads could decrease without impacting the convenience of passengers.

**Current aTaxi Fleet Progress**

In fact, even with self-driving cars still in research and development stages, this concept of converting driverless vehicles into autonomous taxis is beginning to take shape as two of the biggest players in the autonomous-ridesharing market have stepped in this direction. In February 2015, Uber announced a new partnership with Carnegie Mellon University to “develop its own autonomous vehicle technology” and mapping system [24]. This action, along with a large capital investment in autonomous vehicle technology [25], will place the company in a unique position in the near future, as Uber has collected substantial amounts of data that uncovers ridesharing and transportation demand from its existing business.

In a mirrored fashion, the world’s leading company in autonomous driving technology (Google) has shown signs of entering the ridesharing market, which is telling of its intentions, considering its driverless car technology is ahead of other car-manufacturers and researchers — “two to five years from being ready for widespread use” [24]. Google has begun to take steps in a direction that would result in their creation of a taxi-service application, with the vision that the vehicles involved would be self-driving cars. With major companies already making moves towards the autonomous taxis, and especially given that Google’s third-generation GX3200 vehicles have already been licensed for commercial use in the state of New York [25], it is not infeasible to expect that such a system could be set in place in NYC.

If such a system involving a network of aTaxis were to be implemented, a significant amount of work would also need to go into the repositioning of taxis while they are empty and waiting to be hailed. With the taxicab analysis later in this thesis providing a full picture of the minute-by-minute demand for taxis over the course of a year, a repositioning algorithm for such an aTaxi network would be laborious, but not infeasible to create.

**1.4 Re-Focusing**

The nation is in need of smarter transportation; to achieve national and societal goals we must reduce the total amount of vehicle miles traveled each day. Again, these goals involve reducing the quantity of unnecessarily-wasted fuel, diminishing the nation’s dependence on foreign oil, decreasing
the size of the nation’s carbon footprint, as well as improving the quality of life and safety of those using our transportation networks. Once a reduction in personal vehicle miles traveled is achieved, congestion, fuel consumption, pollution, and other transportation-related problems will diminish. This thesis believes that given the permanence of America’s road-based transportation network, the solution lies in using our existing network more intelligently. Casual ridesharing is the beginning of this solution.

By using the New York City TLC’s real taxicab trip data from 2013, this thesis suggests ridesharing improvements to the city’s current taxicab system with the goals that improvements be taken into consideration by the city and that the learned knowledge can be helpful on a broader, more general scale. An implementation of an effective ridesharing system in New York City would serve as a uniquely exceptional first step toward building familiarity with ridesharing as a commonly accepted, used, and expected method of transportation.

In today’s world, success for such an implementation involves the use of mobile applications and real-time matching of trips for identifying rideshares. Such an application used by the NYC TLC will need to be deliberately designed and introduced so as to foster driver- and passenger-adoption. Existing services, like UberPool, help prove the concept’s popularity while a TLC and union-backed application is yet to be developed.

In tomorrow’s world, self-driving autonomous taxis combined with an effective ridesharing system appear to be the solution. Despite lingering hesitations surrounding the true function of self-driving cars, they have continued to make considerable progress in recent years. So much so in fact that most major automotive manufacturers, as well as Google and Uber, have pushed towards the goal of autonomous taxis fleets.
2013 NYC TLC Data

This chapter presents a more involved description of the structure of the New York City Taxi & Limousine Commission and of the data that was used in this thesis research.

2.1 Background on 2013 NYC TLC Data

Under a Freedom of Information Law request, the New York City Taxi & Limousine Commission released all of the TLC trip records for the 2013 calendar year. Shortly after being released, the data was hosted on the internet and made available to be downloaded by any individual with adequate bandwidth and storage space [26].

Releasing over 170 million NYC fare records, complete with timestamps, GPS coordinates, passenger information, and more was an unprecedented action by the NYC TLC — one that can only result in innovation. Professional, academic, and independent workers can now harness the detail and breath of the data to produce valuable analyses of one of the most iconic transportation systems in the world. Utilizing this publicly available and real data, as opposed to synthetic data, for the basis of 100% of the analysis in this thesis allowed the research to be as close to real-life conditions as realistically possible.

For the purposes of this thesis, the decision was made to ignore outlier trip records that existed far from the location of most taxicab trips — many errant trips, for example, either originated or terminated in a body of water. As these data points were more likely to represent a collection error than an actual trip, the region for analysis was restricted to the five boroughs of New York City. It is believed that making such a constraint did not negatively impact the research, as NYC TLC vehicles can only technically serve demand within the five boroughs; but doing so was necessary as
the 2013 NYC TLC trip data contained clear outliers to this geographical region. An output of the five boroughs has already been visualized in Figure 1.3.1, but this region will become evident in later sections of this thesis. For further reference, the exact boundaries used to filter all trips from the original data set are further discussed in Section 2.4 and can be visualized on page 85 of the Appendix in Figure A.0.3.

## 2.2 New York City TLC Service

The New York City Taxi and Limousine Commission can be broken down into three different types of services: yellow taxis; boro taxis; and other for hire vehicles (FHVs) [10].

![Types of TLC Services](image)

The iconic yellow taxicab service consists of 13,437 unique medallions, each of which give a vehicle the right to operate as a yellow cab. The number of these yellow cab medallions available is limited, causing them to be in high demand. This explains why the vast majority of medallions are double-shifted, meaning two drivers split a medallion on a given day as described in Section 1.3.4, which maximizes the daily road-operation duration of a vehicle. Yellow cabs have the right to accept street hails or e-hails from anywhere in New York City [10].

Boro taxicabs are a recent addition to the TLC service. These wasabi-green vehicles were introduced on August 8, 2013 to “provide legal, yellow-caliber taxi service to the boroughs, since 94% of yellow taxi pickups occur either in Manhattan or at one of the airports” [10]. Boro cabs have the ability to serve as both a street-hail pickup service and a pre-arranged for-hire service. However to reduce congestion and cannibalism, as well as to encourage a more widespread transportation service, the TLC does not permit the Boro cabs to conduct street-hail style pickups of customers in Manhattan below E 96th St or W 110th St and at the NYC airports. While street-hailing in this region and using airport taxi stands are prohibited for Boro cabs, they are permitted to pre-arrange pickups in any location. Figure 2.2.2 shows the result of such clear distinction in geographical precedence between the two street-hail-capable services.

A slow adoption of Boro taxicabs followed their August introduction. Slow enough, in fact, that only 6% of the vehicle owners felt the need to be double-shifting. Yet throughout the remainder
of 2013, this number continued to increase as the street-hail demand outside of Manhattan increased. As plenty of residents of New York City do not own personal vehicles, the Boro taxis were found to serve a transportation demand that was not convenient through public transit and had yet to be tapped by the taxicab network [27]. Anticipating continued growth of the Boro taxi service as it becomes better known, it is reasonable to assume that the number of taxi pickups outside of Manhattan will increase as the service begins to find better sources of existing demand and generates new demand.

The final piece of the NYC TLC services is the collection of for-hire vehicles. These black cars and liveries do not rely on hailing. Instead, each vehicle is affiliated with a base located somewhere within New York City. Transportation demand served by these vehicles must go through their respective base, which sets the fare of the service. Except for use of black cars with corporate clients, this form of prearranged and contracted public transportation is fading from the New York City scene [10].

2.3 Source Data Description

The data set used by this thesis originally contained 173,179,759 individual records of taxi trips. Given that the pickup times alone spanned over 84% of all possible minutes of 2013, it is safe to
say that at almost any point in the year, at least one NYC inhabitant was riding in a taxicab. This immense data set contains a vast amount of information about an even more immense transportation service, and it does so on a trip-by-trip basis, with fields such as: medallion number, hack license number, pickup date and time, drop-off date and time, passenger count, trip duration (in seconds), trip distance, pickup latitude and longitude, and drop-off latitude and longitude. A sample of the original trip data is included in Figure A.0.4 on page 86. Recall that a medallion is a permit that is attached to a vehicle, allowing it to operate legally as a TLC taxicab. The hack licenses on the other hand are assigned to individual drivers, meaning that a medallion can have multiple hack licenses associated with it [28]. For the purpose of this thesis, the words medallion and taxicab will be used interchangeably; similarly the terms hack license and driver will be equivalent. For example, when describing the split shift occurrences in Section 1.3.4, another description of it would be: most NYC medallions use two different hack licenses on a given day to cover as much demand as possible.

2.4 Source Data Alterations

Any “big data” data set inevitably contains erroneous entries; the original 2013 NYC TLC data set is no exception. This section describes some of the data clean-up conducted to remove identified errors that ultimately could have impacted the analyses and conclusions of this thesis.

One error that came disguised, due to the anonymization of the driver information, was that some trips lacked identification numbers in the medallion and hack license fields. Upon initial analysis, a particular driver and a particular medallion appeared to be doing an incredible amount of business in 2013, more so than any of the others. However after further study, the reason for the superb performance was discovered. The NYC TLC used an MD5 hash to anonymize the 2013 taxi data before releasing it; and by nature of the MD5 hash function, the hash of the character “0” is “CFCD208495D565EF66E7DFF9F98764DA”. This meant that all of the trips containing “CFCD208495D565EF66E7DFF9F98764DA” as either the medallion ID or driver ID were not actually assigned to a physical car or driver in the TLC data set. All of these incomplete data entries were connected to the same particular MD5 hash, explaining why the medallion- and driver-specific analyses for this ID were greater than those for any actual legitimate medallion and driver. Though it is likely these entries represented trips that actually occurred, they were removed before the analysis of this data set was conducted because they were incomplete and there was no way to ex post facto populate them. Within the data set, there were a total of 14,305 violations in the hack license field and 962 violations in the medallion field. Due to the removal of trips involved in this anonymization
confusion, the total number of unique medallions was reduced from 13,742 to 13,741 and the total number of unique drivers was reduced from 42,822 to 42,821.

Another necessary alteration had to do with trip information that was not relevant to the analysis at hand. Trips recorded as traveling zero miles were detrimental to the analysis of determining how ridesharing could exist, as in reality no such trip would be demanded or taken. And if an actual trip did occur during that record’s timeframe, then the entry involved either a driver-error or a temporary malfunction in the taximeter itself. Thus as before, the manual removal of such incomplete trips was conducted to additionally remove trip data that was illogical or mis-represented. In order to execute such a removal, trip distances were inspected by a boolean operator to determine if the value was zero or non-zero. Within the entire data set, 1,129,639 trips violated this zero-mile rule and were removed.

Next, trips within the NYC TLC 2013 data set were analyzed to observe the trip records geographically. As previously mentioned, this thesis is concerned only with trips contained within the geographical bounds of the five boroughs of New York City. Using the road network of the five boroughs as a guide, a rectangular boundary was constructed so that it fit just around the limits of this network. Figure A.0.3 on page 85 depicts this boundary — the latitude limits of which consisted of everything within the coordinate set [40.496532, 40.915098] and the longitude limits of which consisted of the coordinate set [−74.255300, −73.699693].

Using mapping tools and the latitude-longitude combinations (or lat-longs) of pickup and drop-off points, trips that did not fall within this logical location were removed from the data set. Trips removed from the data set included those containing zero values in some or all of the lat-long fields, as well as values that fell outside of the geographic area consisting of the five boroughs (shown in Figure A.0.3 on page 85). Note that both the pickup and drop-off lat-longs were considered when checking whether a trip could pass this geographical test. Some of these lat-long inconsistencies could have stemmed from mechanical failures within the GPS component of the taximeter; others could have resulted from data processing errors by the NYC TLC. Ultimately, it was important to remove all trips violating the specified guidelines of this research. The total counts of trips that had points violating the aforementioned rectangular boundary were: 3,321,720 for pickup locations and 3,497,285 for drop-off locations.

It must be noted that sum of all of the above violations exceed the number of trips actually excluded from the total data set, as not each inconsistency represented a unique trip. Due to the entire data clean-up process, the number of trip records decreased by 4,399,917 trips, or 2.5% (from 173,179,759 to a total of 168,779,842 trips).
Lastly, additional fields were appended to the original data set to allow for the analyses conducted in subsequent chapters of this thesis. Specifically, nine additional pieces of information were added for each row of data: trip_ID, medallion_ID, hack_ID, pickup_pixel_ID, pickup_Y_pixel, pickup_X_pixel, drop-off_pixel_ID, drop-off_Y_pixel, and drop-off_X_pixel. These first three fields, containing “ID” were unique to specific values in the data set and allowed for faster data manipulation. The pickup and drop-off pixel IDs were generated as a result of splitting data into individual pixels, explained later in Section 4.1, and their X- and Y-coordinates were based on those assignments. Additionally, two of the original entries — date-time strings for pickup and drop-offs — were each split in half into the respective entries of date and time. A sample of the finalized, modified trip data is shown in Figure A.0.5 on page 87 of the Appendix and the Python code attached in Section B.3 shows how these modifications technically occurred.
Chapter 3

Assorted Analyses of the Base-Level Transportation System

In order to determine ways in which the New York City Taxi and Limousine Commission service can be improved, it is necessary to understand current demand, trends, and other statistics related to taxicabs. All of the analyses in this chapter used the 2013 trip data released by the NYC TLC as a basis for the demand of taxicab trips within the city. To begin designing well-informed ridesharing policies, the trip data was contextualized and “base level” insights of transportation within New York City were developed.

3.1 Taxicab Pickup Activity, by Week

The first insight into the NYC taxi demand was the examination of an average 24-hour day of NYC Taxi trips — shrinking an entire year into an average week helped present characteristics of demand that otherwise would be indiscernible at a larger scale. Of course no two days actually were identical, but by visualizing 2013 taxicab trip data over an average week, this thesis could gain a healthy understanding of what transportation demand looked like on a given day of the week. Figure 3.1.1 displays the demand of all trips during 2013 of a single week on a minute-by-minute basis, from 12:00 AM on Monday to 11:59 PM on Sunday. Note that this figure is also broken down by the geographic location of the trip pickup — a bucket for each of the five boroughs — and that the legend colors in this figure align with the boroughs visualized in Figure 1.3.1 on page 6. It is clear that the minimum demand for taxicabs occurred between 3 AM and 4 AM each day, whereas
the maximum demand hovered around 7 PM.

Manhattan contained the overwhelming majority of taxicab pickups. More specifically, 90.3% of taxicab pickup activity over the course of 2013 originated in Manhattan. From Figure 3.1.1, it appears as though Queens had an almost-constant amount of pickup demand from about 6 AM through midnight. It must be acknowledged, though, that Queens is home to both John F. Kennedy (JFK) and LaGuardia (LGA) airports, which made up 3.5% of all NYC TLC 2013 taxicab originations. As will be discussed later in this chapter, the constant stream of taxicab demand at these airports was a big reason for the shape of this curve in the graph. Flights usually arrive uniformly throughout the day during the timeframe of 6 AM through midnight, meaning that individuals looking to leave the airports were the cause for such constant levels of activity in Figure 3.1.1.

The remaining 6.2% of trips originated in the other four boroughs: Brooklyn with 3.1%, the rest of Queens with 1.5%, the Bronx with 0.9%, and Staten Island with just 0.8% [10]. In Figure 3.1.1, it can be seen that the Brooklyn trip frequency had a local peak around 6 AM, and a daily global maximum late at night. The Bronx and Staten Island curves may appear to be steady at the 0 marker, but this is merely due to the result of their low-magnitude trip counts and the large y-axis scale. Figure A.0.6 on page 88 of the Appendix has the same plot magnified along the y-axis to pick up the deviations in activity for the Bronx and Staten Island. There appears to be no clear pattern or trend in the data, however this could likely be due to a lack of sufficient historical trip data for a trend to be discernible in the plot. Such a lack of data was the byproduct of cabs not spending as
much time in these areas, not as a result of a poor quality data set.

The aforementioned daily drop in taxicab usage — from 3-4 AM visible in each of the curves — can be attributed to both the lack of demand by passengers and the lack of available drivers circling through the city. In the future, if an autonomous taxi fleet were to replace the current taxicab network of drivers, the 3-4 AM time period would be optimal for vehicles to undergo maintenance, check-ups, inspections, refueling, and more. For Manhattan, there was also a noticeable drop in taxicab pickups around 4 PM each day. As mentioned in Section 1.3.4, a taxicab vehicle on a given day usually is split between two drivers and the change of driving-shifts tends to occur around 4 PM. Therefore it can be concluded that the local minimum was more likely due to a shift change in drivers rather than some sort of natural drop in taxicab demand.

Peak demand on weekdays for taxicabs in Manhattan hovered around 7 PM, a time where many individuals leave work, head to dinner and drinks, run errands, etc. As the week transitioned from work-days to the weekend, a new peak appeared closer to midnight, plausibly as a result of the increase in number of people participating in nightlife-related activities.

Taxicab demand across a whole week fluctuated as well, though this oscillation may not be discernible in the graph. Over the course of a year, Mondays saw the fewest amount of trips at a total of 21.9 million, Tuesdays saw 24.1M, Wednesdays 24.3M, Thursdays 24.9M, Fridays the most at 25.7M, Saturdays 25.6M, and Sundays dropped to 22.3M. Saturday and Sunday did not have as high of a demand at their peak time, instead their demand was spread out more widely across the entire length of the day — due to both individuals staying up past midnight the night before and fewer individuals on a strict work schedule. The overall weekly trend in Figure 3.1.1 signifies that as a week gets later, schedules become less planned, and there are generally more activities to participate in outside of a work-routine, the demand for taxicabs grows.

Ridesharing programs are more effective when demand for transportation is at its peak. Although creating a ridesharing program that is only “in effect” between certain hours of the day might not be ideal for reducing congestion, pollution, and more, it could be a cost-effective way to introduce the general public to a new concept of transportation. Gradually a full-service ridesharing program could be introduced to a public prepared to accept it; and when self-driving cars begin to enter the equation, the NYC taxicab network would be primed for adoption of an autonomous taxi technology.
3.2 Taxicab Trip Distance

Given the above general understanding of when and where in New York City individuals enter taxicabs, it is beneficial to consider whether certain aspects of a pickup relate to certain types of trips. For this section, the thesis considers the length of a trip. Such an analysis allows for extra knowledge about what the characteristics of a taxicab trip could be before it occurs, based solely on the information regarding where and when an individual enters the vehicle. Knowing that a specific type of trip is likely to be a longer distance could result in a ridesharing service being able to alter the parameters of a rideshare in order to capitalize. This type of predictive analysis could prove useful in determining ideal policy parameters as well as in creating a future application of a taxicab repositioning and allocation system. This section analyzes the frequency of taxi trips with certain trip lengths, segmented by two different characteristics — pickup borough and pickup daypart — through the use of empirical cumulative distribution function plots.

3.2.1 By Borough

A cumulative distribution function (CDF) plot visualizes the “probability”, or proportion of times, that a given real-valued random variable ($X$) will be observed to have a value less than or equal to a certain number ($x$). Since such plots are theoretical, and we do not pretend to know (or assign) the actual distributions of such random variables as they relate to this thesis, the plots in this section and in later sections display the empirical distribution functions — a similar curve that represents the theoretical CDF for an empirical sample of points.

The way to best understand these plots is to follow a specific curve as the value in the x-axis increases. For a given value along the x-axis, $x$, the y-value of the curve represents the proportion of occurrences in an overall sample containing a value less than or equal to $x$. To give a preliminary example interpretation, consider the asparagus-colored line in Figure 3.2.1 representing trips originating out of Queens. At a trip length of about 10 miles, the curve crosses the 50% mark — in other words, only 50% of trips originating in Queens had a trip length of 10 miles or less.

Note that these types of plots do not specify the relative quantity of trips, as each curve is a proportion ranging from 0 to 1. Therefore, while viewing Figures 3.2.1 and 3.2.2, it is important to keep in mind what was previously presented (Figure 3.1.1) about the relative proportions of taxi trip origination.

By comparing the different slopes and levels of curvature in Figure 3.2.1, certain trip characteristics about the NYC TLC data were learned. The borough-by-borough analysis that follows pulls
The Bronx curve in the figure above shows how trips originating from this borough were either short (50% of trips from the Bronx were under 2.5 miles) or relatively-long (the 90% mark of all trips from the Bronx was only passed at a trip length of 9 miles). This makes sense: passengers in the Bronx could have been traveling directly to Manhattan or to somewhere else in New York City and avoiding the Manhattan congestion entirely (which could involve a more round-about trip). There are also plenty of individuals who would have taken a taxi to the airport from a location in the Bronx.

The Queens curve is dramatically different from the other boroughs. Queens is the largest borough in geographical terms, which would result in a less steep curve due to a higher frequency of longer trips; but this alone does not explain the curve seen in 3.2.1. Again, Queens is home to both JFK and LGA airports, which independently were responsible for about half the total taxicab traffic of the four, non-Manhattan boroughs. In fact, the number of trips originating from the airports was more than double that of the count originating from the rest of Queens, meaning the trip characteristics of these locations heavily affect the Queens curve. The reason behind the slowly-ascending Queens curve comes from the fact that many of these trips originated at an airport and headed toward a destination, most likely a home. The cumulative distribution curve of a uniformly
distributed random variable is a steady diagonal line that reaches from 0 to 1; although there is a slight deviation in the Queens curve, it resembles such a curve due to the fairly uniform geographic spread of homes of individuals who picked up cabs at an airport.

Brooklyn had fewer very short trips and very long trips; almost all trips fell into more of a mid-range distance. This can be confirmed visually through the steep ascension after the 1-mile trip-length mark. 50% of Brooklyn’s taxicab trips were under 3 miles in length, while 90% of them fell under the 7-mile mark. Such a characteristic of Brooklyn trips makes sense, as many taxicab trips involve either driving between places in Brooklyn, a borough which is slightly more spread-out when compared to Manhattan, or driving to Manhattan, which is commonly a mile or more away.

Given that the majority of taxicab trips originating in Manhattan stayed within the geographically small and population dense borough, Manhattan trips were more frequently shorter in distance than trips originating in any other borough — 50% of trips were under 2 miles in length, and 90% were under 5 miles.

Although Staten Island had dramatically fewer trips than Manhattan, the borough was able to boast a trip-distance frequency pattern similar to Manhattan for the short trips — 50% of trips were under 2 miles — but it took until a trip length of 8 miles to reach the 90% benchmark.

### 3.2.2 By Daypart

When considering how a more efficient taxi system could be implemented in New York City, it is also useful to understand the types of trips that are in demand at different points throughout the day. This thesis has already inspected the frequency at which taxis are hailed over the course of both a day and a week. To do further temporal analysis, this subsection divides the 2013 trips into groups constructed by time of day, or daypart, and considers their distribution of trip lengths.

Consider the cumulative distributions of trip length for each individual daypart in Figure 3.2.2. For this analysis, a day was split into five commonly used dayparts: Overnight (12:00AM - 6:00AM); Morning (6:00AM - 10:00AM); Daytime (10:00AM - 3:00PM); Afternoon (3:00PM - 7:00PM); and Nighttime (7:00PM - 12:00AM).

Unlike the clear variations shown in the taxicab trip-lengths split by pickup location, the same data when divided by daypart suggests that the trip-length of New York City taxi demand does not differ as dramatically over the course of the day. It can be seen in Figure 3.2.2 that overnight, morning, and daytime trips all shared the same distribution of trip-length demand, meaning that if a cab were to be hailed during one of these three dayparts, a ridesharing system would not be able
to gain an additional perspective on what type of trip would be requested. For these three dayparts, 50% of trips fell at about 1.5 miles or less and 90% of all trips were 6 miles or shorter in length.

The last two dayparts told a different story. Afternoon taxicab trips consisted of slightly fewer short trips, as the 50% mark for this daypart fell on the 2-mile marker but the 90% cutoff was no different than that of the previous dayparts. The nighttime daypart was the only curve that truly deviated from the others. With 50% of trips under 2.5 miles and 90% of trips under 7.5 miles, the nighttime daypart curve implied that the nature of trips taken between 7 PM and midnight were inherently different than the rest of the times during the day. Most likely, this difference was because these trips involved a lot of individuals who just wanted to head directly home from wherever they were geographically.

### 3.3 Taxicab Usage in 2013 and Impact of Precipitation

New York City’s location in the Northeastern portion of the United States means that its inhabitants are frequently subject to inclement weather. Although not a major undertaking of the analyses in this thesis, a helpful perspective when understanding NYC’s transportation demand — and all of its patterns and responses to external events — is to view how inclement weather related
to the TLC’s 2013 taxicab data. As more than 90% of the taxicab activity originated within the
borough of Manhattan, daily weather data over the entirety of 2013 was retrieved from a station
taking measurements in the middle of Manhattan and was used as a proxy for all of New York
City [29]. An initial day-by-day visualization of the number of taxicab pickups and the amount of
rainfall in New York City over 2013 is shown in Figure 3.3.1.

First by looking only at the taxicab pickups curve, it can be seen that over the year taxicab
pickups experienced two major effects. Taxicab usage was affected by seasonality: it experienced
an overall decrease over the warmer, summer months and over the winter holidays. And perhaps
more notably in the curve, taxicab usage fluctuated day-by-day. This was displayed earlier in
Figure 3.1.1, but is even more apparent in Figure 3.3.1. To give a relative sense of scale, the time
span of a Figure 3.1.1 is condensed into seven points in this year-long plot; meaning that each month
should contain about four of the weeks displayed in 3.1.1.

When reading Figure 3.3.1, the overall trend of pickups experiences some irregularities, so it is
important to clarify. The reason for most of the unusual drops in taxicab pickup demand over the span of 2013 was due to the lack of passenger demand during major holidays, while one succession of drops resulted primarily from a lack of drivers. The first major drop, which occurred toward the end of May, was on Memorial Day. The next major drop, at the beginning of July, was Independence Day weekend, as July 4th fell on a Thursday. The severe drops in early August were the only ones not related to a major holiday. After historical research, the dates of these drops were found to align with out-of-the-ordinary traffic events, where the city warned of heavy congestion due to numerous bridge and street closures [30]; as confirmed in a later section, such a major notice resulted in a decrease in overall taxicab workforce. Toward the end of 2013, the drop at the beginning of September was on Labor Day, while the drops at the end of November fell around the Thanksgiving holiday; the major spike towards the end of December coincided with the Christmas holiday.

When attempting to align any sense of correlation between precipitation and changes in taxicab demand, no immediate relationship could be found from Figure 3.3.1. The significant trends in the daily pickup demand appeared to be correlated with the specific day of the week and the seasonality over the entire year. In order to account for the impact of these trends, and to better view a possible effect of precipitation on taxicab pickup demand, each day of the week was separated into its own data set. Then, the specific days were ranked and plotted in descending order of total taxicab pickup quantities. If precipitation were to have had some sort of correlation with taxicab pickups, a trend would have emerged in the precipitation-half of each plot. As can be seen from the plots in Figures A.0.7 and A.0.8 of the Appendix, it was evident there was no correlation between the two metrics as the plot of precipitation can not rationally be explained by a trend.

Such a finding, or lack thereof, suggests that even though strong inclement weather may be unpleasant to commute in, on the whole it does not affect each individual’s demand for taxicabs enough to make an impact on the NYC TLC system.

### 3.4 Airport Activity

New York City is not the sole city with taxicab usage well-connected to airports. Taxicabs are useful due to the nature and length of trips being taken — multi-day or -week round trips, and even one-way trips, are common and may not merit the parking of a personal vehicle in an airport lot. Although a long taxi ride might be expensive, it can often be a better option than paying a fee to keep a car at the airport.

Airport taxicab usage is especially popular in New York City, where many individuals do not own
(or do not prefer to use) personal vehicles. Investigating the 2013 data set, it was found that taxicab trips involving an airport pickup or drop-off represent 5% of all taxi trips. Looking specifically at individual airports: yellow taxis alone were responsible for 26% of all passengers arriving at, or departing from, LGA; this metric dropped to roughly 10% for JFK, as the AirTrain service shuttled another 10% of passengers [10].

For airport-specific locations, daily peaks in taxicab demand occurred twice — once at 5:30AM and again at 4:30PM. When these spikes in usage occurred at the airports, the total number of pickups were high enough to represent between 4-7% of all taxicab pickups occurring in NYC. More impressively, over the span of an average week, the relative peak in taxicab demand occurred on Sunday nights, making up roughly 8% of all NYC TLC pickups [10]. This peak speaks to the relatively constant demand for taxis at airport locations, to the return of travelers in time for the work week, and to the comparatively lower demand elsewhere in the system on Sunday evenings.

3.5 Taxicab Fleet Analysis

As described in Chapter 1, the emergence of self-driving cars is near — close enough that it is prudent to keep their likely future implementation in mind. This thesis gleans trends of the taxicab fleet through analysis of the 2013 data set in order to enable initial exploration of the concept of replacing current vehicles with self-driving cars by introducing a next generation, ridesharing-capable taxicab system to New York City. Understanding how the existing taxicab fleet conducts business and serves inhabitants of the city’s five boroughs allows for the discovery of inefficiencies.

3.5.1 Fleet Usage

On an average day of 2013, the number of taxicabs that recorded at least one trip was 12,480. Given the total number of medallions recorded in this data set, this means that 90.8% of the total fleet was active on an average day. This average value was indeed high, but obviously not close to the maximum value, indicating that on any given day, there was some cause for taxicabs not being able to serve any customers — likely due to vehicles being serviced or repaired.

Figure 3.5.1 shows the oscillation of such a “percent of total fleet that is active” metric over the span of the entire year. A reason for explaining the ritualistic ebb and flow of active taxicabs over the course of the year is the nature of taxi demand over a given week, which hit a minimum on Mondays and a maximum on Thursdays, as made clear earlier by Figure 3.1.1. As introduced in Section 3.3, the beginning of August was host to three days of major drops in taxicab fleet percentage, also
made apparent in Figure 3.5.1; these occurred in sync with the previously mentioned major roadwork and road-closure announcements made by the City of New York [30]. While these road work events aligned with a significantly low taxicab availability, the remainder of the unusual drops in the activity analysis curve were due to holiday-related dates. By comparing the dips in medallion-activity in Figure 3.5.1 and the pickup-activity (black curve) in Figure 3.3.1, one can determine whether a shortage in taxicab activity correlated with a drop in pickups (in which case demand may not have been satisfied), or whether an external event caused a decrease in demand.

Figure 3.5.1: Per-Day View: Proportion of Fleet that is Active

![Fleet Activity Analysis for 2013](image)

Yet perhaps a more insightful approach to understanding what exactly was occurring with taxicab activity on a standard day is not to view overall fleet activity — measuring whether a cab recorded a trip each day — but rather to analyze *exactly* how many taxicabs generated revenue at a given moment in a day. Figure 3.5.2 provides this analysis. For such a study, a standard week with no extreme events was necessary. From the patterns and deviations in Figures 3.3.1 and 3.5.1, it was concluded that the month of March appeared stable both in terms of the number of taxicab pickups and the number of active taxicabs, so the first full week of March was selected for further analysis — March 4th through March 10th.

As should be clear from Figure 3.5.2, the curve traces through a week from midnight before Monday, March 4th to midnight at the end of Sunday, March 10th. For each minute in the week, all of the taxicabs in the process of transporting a passenger from pickup to drop-off point were tallied. By dividing each per-minute total value by the number of medallions in the TLC fleet (13,741),
this research determined a metric describing the proportion of the fleet that was *actively* generating revenue at any given moment.

Comparing Figures 3.5.1 and 3.5.2 is very telling of the inefficiency of the current taxicab system. Taking this standard week in March to be representative of normal taxicab demand in New York City, it can be concluded from the plot that very rarely was even half of the taxicab fleet generating revenue at a given minute during the week, and on average only 32% of taxicabs were transporting a customer. Within the week, there were clear peaks in the occupancy rate of taxicabs. High occupancy periods for the TLC vehicles predictably aligned with peak travel times: weekday morning rush hours (8 AM to 9 AM); weekday evening rush hours (6 PM to 7 PM); and weekend nightlife activity (late dinner-time through 4 AM, when NYC bars close). Although this thesis does not specifically address manners in which more taxicabs can be consistently utilized for generating revenue, a figure such as 3.5.1 signals just how inefficient the current NYC TLC system is. Human drivers, even professionals, need to take breaks throughout their shifts. It is feasible to expect that most drivers are not able to repeatedly complete back-to-back fares without taking a personal break. These independently-thinking drivers also do not necessarily know the optimal repositioning location upon the termination of a taxicab fare to most-quickly find the next customer. An aTaxi network of vehicles, on the other hand, could tell a dramatically different story.
3.5.2 Per-Medallion Information

Lastly, it is informative to consider NYC TLC information on a per-medallion basis. All of the metrics in this sub-section are relative to the 13,741 distinct medallions in this data set; they should help fill out an understanding as to what daily activity looks like for a single medallion-taxicab.

Figure A.0.9 on page 90 shows the distribution of trip totals for each taxicab. The density plot on the left of the figure indicates that the number of trips per taxicab is left skewed, meaning more cabs were less efficient than the norm and few cabs outperformed the rest of the fleet. Indicated on the density plot by dotted and textured lines, the annual average number of trips per taxicab was 12,247 and corresponding median statistic was 13,381 trips.

The scatterplot on the right of the figure gives this same distribution of trips per taxicab over the entire year, but also plots it as a function of the number of registered drivers associated with each medallion. Medallions with a larger number of associated drivers had a smaller variance in the number of trips recorded over the entire year. This is logical, as multiple drivers are capable of keeping a taxicab running on the streets for longer periods of time.

Further calculations on the data revealed that each taxicab conducted an average of 34 fares each day, with a median of 36 fares. On average, this taxicab work was conducted in a matter of 7.6 hours, leaving almost two thirds of the day where the medallion was effectively not a taxicab — this statistic, in a way, aligns with the aforementioned fact that only 32% of the taxicab fleet was transporting customers at a given moment. A lot of taxicab time is spent not shuttling customers around New York City.

When a registered medallion is not operating as a taxicab, and is not driving along the streets of New York, there is no issue. However, non-operative taxicabs driving along the streets serving no passengers are problematic; they cause the transportation system to be inefficient due to there being additional vehicles on the road that are serving zero purpose to society.

Ideally, any automobile traveling has a purpose most easily defined as serving the transportation needs of the individuals inside the vehicle. In a place as dense and as perpetually busy as New York City, it is imperative that each vehicle contributing to the city’s congestion be absolutely necessary. An empty taxicab adds to the congestion of New York without satisfying the transportation needs of even one individual, since the driver is not using the vehicle to complete a personal trip from point A to point B. The elimination, or minimization, of purposeless vehicles in NYC would help contribute to the decongestion of the road network.
3.6 Concluding Remarks

The set of varying analyses in this chapter provided a series of valuable insights into the occurrences and characteristics of the New York City TLC taxicab activity during 2013.

Over the course of a week, taxicab demand was found to vary both by day of the week and throughout each day. This demand continued to fluctuate seasonally, responding to the ebb and flow of the schedules of NYC inhabitants and of the four seasons.

In terms of the sheer magnitude of taxicab demand, Manhattan dominated the amount of pickups throughout 2013. And relative to their geographic size, JFK and LGA airports also were home to a large number of taxicab pickups. Such high activity leads one to believe that these areas would be likely candidates for the implementation of a ridesharing system, if one were not to be opened up to the entire city.

Variations existed with regard to the distribution of trip lengths between the four boroughs. Longer trips are significantly more likely to be associated with trips originating in Queens — specifically the airports — as well as late at night on an average day. Such knowledge of trip length distributions can begin to help with the implementation of a smart ridesharing system and the creation of taxicab repositioning systems. Trip lengths become an even more important piece of the transportation puzzle when one recognizes that longer trips are better candidates for ridesharing opportunities: there is more likelihood that another passenger will have a trip that meets the ridesharing requirements; there is likely more willingness of the passenger to share a ride, as the fare would be much higher; and passengers would feel less affected by ridesharing if the addition would only be an incremental addition to their originally intended, long route.

No correlation — at least no significant relationship — existed between the precipitation on a given day and the amount of taxicabs that were demanded. Of course, such an analysis is only preliminary. It is possible that a relationship would exist if a taxi system existed that could satisfy a higher spike in demand at a given moment — a benefit of ridesharing, as the number of drivers needed is not strictly proportional to the number of passengers requesting a ride.

Lastly, taxicab fleet usage is far from optimal. The proportion of taxicabs actively generating revenue at a given moment was noticeably low, even though overall fleet usage was fairly high. This is a sign that an aTaxi fleet adoption would result in more efficient use and would have the capability of either: reducing congestion by requiring fewer vehicles to satisfy a similar demand; or satisfying an expanded user-base while maintaining the same number of vehicles in the fleet. A lot of a taxicab’s time is spent not shuttling passengers around New York City, which raises the issue
of cabs merely adding to the congestion of the city when they are not transporting customers and serving no purpose.
Chapter 4

Pixelization of NYC Trips

The precisely detailed 2013 NYC TLC data proved to be too computationally intensive to conduct a ridesharing analysis using the specific lat-long coordinates of each trip’s pickup and drop-off location. This was due to the fact that the entire region of analysis (specified in Section 2.4) covered roughly 850 square miles of land, containing essentially an infinite number of lat-long possibilities due to the continuous nature of the coordinate system. In order to reduce the complexity of a ridesharing analysis, the geographic area of the five boroughs of Manhattan was pixelated. As the term implies for any digital image, this pixelization of trip data took a less detailed look at the 2013 trip demand, by turning the total area into a collection of 84,970 adjacent pixels. This discrete — yet still detailed — set of pixels allowed both for a clearer visualization of transportation demand and for more realistic computations.

4.1 Overall Grid Structure

In order to not sacrifice the high level of detail in the original trip data, the dimensions of the created pixels were set to be just 0.1-by-0.1 miles (or 528-by-528 feet). A visualization of the pixelization set up is shown in Figure A.0.3 on page 85 of the Appendix.

As the grid of 84,970 pixels was made ignoring the context of the geography assigned to each part, many of the pixels were void of trips. In fact only 36,164 of the pixels contained at least one trip pickup. Each pixel created contained a unique ID, as well as an X- and Y-coordinate to keep track of their relative position to each other. To help understand this new coordinate system, consider Figure A.0.3 on page 85. The bottom left pixel has a pixel ID of 0 and XY-coordinate of (0, 0) and there were a total of 293 pixels in a given row of the grid and 290 pixels in a given column.
The pixel IDs incremented first horizontally and then vertically as one traversed initially from left to right and then from bottom to top throughout the grid. This means that a pixel ID of 292 had the coordinate location of (292, 0) while 293 represented the pixel located at (0, 1) in the overall grid.

Several attempts to rotate the network by 28.9 degrees were made in order to align the major Manhattan avenues with the grid system for ease of use. These attempts were stymied primarily because application of a rotation matrix to the trip data severely complicated it and diminished its value for future ridesharing analysis using the physical roads. Additionally, rotated pixels would only exactly align with some of the intersections in Manhattan, given that its road network is not a perfectly symmetrical grid, and would not aid in the analysis of the rest of the NYC roads, not aligned in a grid structure. After much deliberation it was determined not to proceed with a rotation of the grid.

4.2 Yearly Pickup and Drop-off Trip Demand Distributions

Viewing the origins, destinations, and frequency of NYC taxi trips allows for an understanding of general trip patterns of the city’s inhabitants. Figure 4.2.1 graphically depicts this distribution using a gradient heat map to represent pixels containing a varying number of trip origins or destinations.

On the left of Figure 4.2.1, we see the trip distribution of all NYC TLC pickups contained in the geographical boundary of the five boroughs of New York City. As mentioned earlier, the borough with the greatest number of pickups was Manhattan. Furthermore, the most concentrated region for taxicab hailing within Manhattan was Midtown, a major and dense business region teeming with activity and full of transit hubs.

It is worth noting the relative scarcity of pickups throughout the other four boroughs, as compared to drop-offs. On the right side of Figure 4.2.1, it can be seen that the drop-off locales for taxicab riders are much more spread out in the four other boroughs. This realization would imply that empty taxicabs searching for riders are most likely going to be heading towards, and roaming the crowded streets of, Manhattan. Outside of Manhattan, there is one other type of region that is highly active for taxicab pickups and drop-offs — airports. As found in the previous chapter, both LaGuardia Airport and John F. Kennedy International Airport possess high amounts of taxicab traffic, signifying that there is significant potential for ridesharing at these locations. The figures within this chapter (4.2.1 – 4.2.3) help to visualize such demand for taxis.

In fact, some airport and train stations in the nation already employ ridesharing systems. Though
these are predominantly manual, where a steward manages a line of waiting taxicabs and attempts to group individuals with similar destinations together, they emphasize the fact that the ridesharing concept is realistic. Such a system exists on 7th Avenue outside of New York Penn Station in Manhattan, but there is not yet an official ridesharing manager for LGA and JFK airports. 3rd-party applications are just now beginning to come to market. Bandwagon, for example, is an iPhone application that allows users to identify potential ridesharers by either searching for individuals traveling to a similar destination, or by ordering a taxi and waiting to see if another individual joins. Bandwagon has already begun to apply its dynamic taxi service to the clogged taxi lines at transportation hubs such as LGA and JFK [31]. Such a service is another example of how the NYC TLC could begin to implement ridesharing policies described later in this thesis.

Using the same scale for the heat map coloring as in Figure 4.2.1, 3D visualizations have been produced, where the extrusion length of each pixel represents the number of trips falling within it. Figure 4.2.2 displays different points of view for taxicab pickup demand in 2013, followed by Figure 4.2.3, which displays taxicab drop-off points using the same points of view.
Figure 4.2.2: 3D Visualization of 2013 NYC TLC Pickup Demand

(a) View 1
(b) View 2
(c) View 3
(d) View 4
Figure 4.2.3: 3D Visualization of 2013 NYC TLC Drop-off Points
Chapter 5

Ridesharing Introduction

An effective implementation of ridesharing within today’s current transportation system is a solution that progresses society and mitigates the negative effects currently experienced due to heavy congestion, significant pollution, and more. The most complete, single metric to consider for representing all of these byproducts of transportation is vehicle miles. With the total number of vehicle miles driven by an individual representing their complete daily life transportation needs, the goal of a ridesharing system is simply to reduce those total number of vehicle miles driven in a region. Recent research amongst 85 different urban cities indicates that “traffic congestion will continue to worsen as the number of vehicle miles traveled continues to grow” [1]. This trend must be reversed. An improvement to a transportation system that limits or constricts the lives of individuals is unrealistic and likely would prove ineffective. A replacement system must be able to substantially reduce vehicle miles, while allowing individuals to accomplish their daily routines — moving from point A, to B, to C, etc.

The ridesharing analysis aspect of this thesis, explored over the next two chapters, looks to determine the feasibility of just such a system based on NYC TLC trip demand data. Many individuals living in New York City substitute taxi usage for their personal means of transportation, therefore this analysis assumes that the city-wide data of taxicab demand is a reasonable estimation of the vehicle miles required by NYC individuals to live their lives. A substantial reduction in taxicab vehicle miles within this analysis could be extrapolated to have a similar effect on vehicle miles in other densely populated cities across the nation, where the same, or a similar, ridesharing system could be implemented.

Though touched on earlier, it merits highlighting prior to diving into two chapters on ridesharing
that a massive value of implementing a ridesharing system comes from the fact that it not only works within the confines of current infrastructure, but also it is flexible and capable of handling significant influxes of demand. Contrasted with common traffic solutions, such as the physical widening of existing roads and highways, ridesharing systems also address matters more important than just congestion by actually reducing the number of vehicle miles driven.

Within this chapter, the concept and calculations behind ridesharing are explained, followed by an overall inspection of the 2013 taxicab data. Additionally, the chapter considers a brief analysis of the top-ranked pixels for taxicab pickups.

5.1 Analysis Clarification and Terminology

While it is helpful to understand how the taxicab pickups are distributed throughout New York City as a whole, we must approach a ridesharing analysis pixel-by-pixel. This is the case in order to ensure that no passenger walks more than 0.1 mile if he or she were to share a ride with an existing taxicab. For simplification purposes, this thesis assumes that each departure from a pixel occurs at its centroid.

As should become evident in the following sections, this thesis refers to a “policy” as the overarching, theoretical algorithm on how a rideshare match can be found. A “framework”, on the other hand, refers to the different geographical-boundary and time-window combinations of a given policy. For this research, each policy consisted of eight different frameworks, but in reality there are an infinite amount of possibilities.

Specific metrics used to describe the effectiveness of ridesharing implementations included average departure occupancy (ADO), percent reduction in taxicab usage (%TaxiRed), and percent vehicle miles saved (%vMiles). Average departure occupancy is a number representing the amount of passengers involved in an average departure under a given ridesharing framework. As can likely be concluded, this ADO metric is computed by calculating the number of passengers of every existing departure under a given framework simulation, and then taking the average. Percent reduction in taxicab usage is a metric that refers to fewer taxicabs summoned for trips over the course of the year; it is calculated by looking at the percent reduction in vehicles between the original trips and the new set of rideshare departures, assuming each departure requires one vehicle. These metrics will be seen frequently in the following sections and tables.

Most important, however, is the %vMiles metric. This thesis determined the value, or effectiveness, of ridesharing policies and frameworks by examining the percent of vehicle miles that were
reduced through the use of ridesharing. %vMiles gives a solid perspective as to how effective the
system would be at reducing repetitive taxicab usage. Of course, a reduction in redundant taxicab
usage translates to a reduction in overall city congestion (total fuel consumption, pollution, etc.).
While this metric was applied to the NYC taxicabs in this thesis, it could equally pertain to personal
vehicles when considering a ridesharing application on a larger scale, or in a different location.

It should be noted that the “vehicle miles saved” metric was computed in this research using
two different methods, depending on which ridesharing policy was used — \((CD = 1)\) or \((CD = x)\).
These methods are described in more detail later in this chapter.

5.2 Methodology of Rideshare Departure Creation

As previously mentioned, the entire New York City region was first broken down into tens of
thousands of 0.1-by-0.1 mile square pixels. All trips originating in each pixel over the course of 2013
were sorted chronologically. After this trip file preparation, the simulation began. The logic behind
how casual ridesharing calculations were computed is as follows.

An unobserved individual (person-A) enters a taxicab and a theoretical timer starts, which lasts
for the duration of a pre-determined time window that ranges from 30 seconds to 5 minutes. Any
other rider who arrives at the same pixel within the time window of person-A — \(\text{and}\) whose trip
satisfies the particular ridesharing policy’s requirements for what can constitute a match — is then
recorded as joining person-A’s departure. Let an individual who meets these requirements be labeled
person-X. Once the time window has passed, person-A and all of the person-Xs are logged as being
in a single rideshare departure and they theoretically set off towards their destinations. What was
counted as “satisfying a policy’s requirements” depends on various aspects of a trip and a departure,
and will be explained in greater depth later.

To sum up, for a given rideshare framework (a specific combination of a time window and geo-
graphic boundary) within a given policy, the trips were iterated through and potential matches were
bucketed together into individual “departures”. These geographic boundaries will be explained next,
but an example of notation for a rideshare framework is MP\(_{300}\), meaning a rideshare with a 300-
second time window and a macroPixel geographical boundary. The algorithm’s logic explained above
is visually depicted in Figure 5.2.1(a) and the resulting departure file is presented in Figure 5.2.1(b).

Again, the time window refers to the maximum amount of time an individual entering a taxicab
would wait for other individuals with a common destination to show up. The degree to which
strangers are capable of having a common destination relates to the geographic boundary.
Figure 5.2.1: MP_300 Rideshare Framework Sample Execution

(a) Searching Original Trip Files

(b) Finalized Departure List
5.2.1 Geographic Boundaries

A geographic boundary named “superPixel” refers to a 9-by-9 pixel grid, while a “macroPixel” refers to a 25-by-25 pixel grid. The relative sizes of these geographic boundaries are shown in Figure 5.2.2. Given the size of a single pixel, this translates to a superPixel having the size of 0.3-by-0.3 miles and a macroPixel having the size of 0.5-by-0.5 miles. Given that the average pedestrian walking speed is 1.25 m/s [32], this translates to a maximum traversal time of 1.5, 4.5, and 7.5 minutes for a pixel, superPixel, and macroPixel respectively as the drop-off is assumed to be at the center of each boundary. Although these times are upper limits, it is safe to say that New Yorkers in particular walk faster than the average pedestrian.

![Figure 5.2.2: Visualization of Pixel Sizes](image)

As should now be clear, when determining opportunities for ridesharing this research considered whether the destinations of potential rideshare matches fell within a geographic boundary of an existing destination. This thesis will occasionally refer to such a boundary as a “geo-fence”

5.2.2 Common Destination Terminology

For the research conducted in this thesis, a common destination (or CD) equal to 1 signified that all trips within a rideshare departure were going to the same destination (determined by the geographic boundary of a superPixel or macroPixel).

Evolved from this, a ridesharing analysis consisting of CD = x (where x is greater than 1) referred to all of the trips in a rideshare departure that went to one of x unique destinations. In order to ensure that the unique destinations did not add too much inconvenience to all the members in the overall rideshare, a destination was only added to the existing departure if it added no more than a specific percentage of distance to the existing total trip distance. This thesis refers to such a metric as “circuity”. Circuity can be thought of as the maximum inconvenience allowed by the system when forming rideshare matches. Any trip that fit within the time window, as well as the geographic boundary, of one of the existing destinations of a given departure was automatically
5.3 Overall NYC Taxicab Inspection

An effective ridesharing program would likely only exist at locations where there is a sufficient volume of demand to result in successful casual rideshare matches between individuals. Therefore, it is useful to have a general understanding of how trip demand in New York City is broken down by pixel. This can be gained from Figure 5.3.1, which is to be read similarly to the plots in Chapter 3. As we can see in the figure, just 294 pixels out of 36,164 active pixels accounted for 50 percent of the taxicab pickups in New York City; and 2,249 pixels accounted for 99 percent of all taxicab pickups. In the remainder of this thesis, when referring to “all pixels” or “all of New York City taxi demand”, this selection of 2,249 pixels is the intended subject as opposed to the 36,164 pixels that contain at least one trip. Note that Figure 5.3.1 is zoomed in along the x-axis; to provide a complete sense of just how skewed the pickup count distribution amongst all of the pixels actually is, refer to Figure A.0.10 on page 91 of the Appendix.

Figure 5.3.1: Cumulative Distribution Function of Taxicab Pickups
5.4 Top-Five Pixel Investigation

The concept of ridesharing depends on the network effect — the value of such a system increases as the number of users increases. Similarly, the potential for a ridesharing system increases with the number of possible users. As this thesis cannot implement a real-life ridesharing system in New York City, it is instead concerned with the latter perspective of how to value a ridesharing system. The first ridesharing analysis of this thesis involved inspecting the pixels within NYC that contained the heaviest amount of passenger traffic.

Prior to such an analysis, it could be hypothesized that the most likely locations for a high demand of taxicabs would be public transportation points, due to their high flux of individuals, almost all of whom have no connection to any kind of personal vehicle at the moment in which they exist at the station. The results in Tables 5.1 and 5.2 made this conjecture evident, suggesting that an incremental implementation of such a ridesharing system amongst the NYC taxicabs could first begin at major public transportation locations.

### Table 5.1: Pixels with the Most Pickups

<table>
<thead>
<tr>
<th>Pixel ID</th>
<th>Pickup Count</th>
<th>Latitude</th>
<th>Longitude</th>
<th>Contextual Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>51412</td>
<td>1,155,367</td>
<td>40.7509003</td>
<td>-73.9942017</td>
<td>Penn Station (8th Ave entrance)</td>
</tr>
<tr>
<td>51120</td>
<td>1,133,281</td>
<td>40.7494011</td>
<td>-73.9923019</td>
<td>Penn Station (7th Ave entrance)</td>
</tr>
<tr>
<td>52586</td>
<td>997,013</td>
<td>40.7566986</td>
<td>-73.9904022</td>
<td>NY Port Authority Bus Terminal</td>
</tr>
<tr>
<td>49355</td>
<td>661,744</td>
<td>40.7406998</td>
<td>-74.0056992</td>
<td>14th St &amp; 9th Ave</td>
</tr>
<tr>
<td>51714</td>
<td>641,810</td>
<td>40.7523003</td>
<td>-73.9770966</td>
<td>Grand Central Station</td>
</tr>
</tbody>
</table>

### Table 5.2: Pixels with the Most Drop-offs

<table>
<thead>
<tr>
<th>Pixel ID</th>
<th>Drop-off Count</th>
<th>Latitude</th>
<th>Longitude</th>
<th>Contextual Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>51414</td>
<td>1,123,473</td>
<td>40.7509003</td>
<td>-73.9904022</td>
<td>Penn Station (7th Ave &amp; 34th St)</td>
</tr>
<tr>
<td>51120</td>
<td>980,991</td>
<td>40.7494011</td>
<td>-73.9923019</td>
<td>Penn Station (7th Ave entrance)</td>
</tr>
<tr>
<td>51713</td>
<td>684,258</td>
<td>40.7523003</td>
<td>-73.9789963</td>
<td>Grand Central Station</td>
</tr>
<tr>
<td>49355</td>
<td>543,071</td>
<td>40.7406998</td>
<td>-74.0056992</td>
<td>Grand Central Station</td>
</tr>
<tr>
<td>52586</td>
<td>536,007</td>
<td>40.7566986</td>
<td>-73.9904022</td>
<td>NY Port Authority Bus Terminal</td>
</tr>
</tbody>
</table>

By looking at Figure 5.4.1(b), 5.4.2(b), and 5.4.5(b), we can see that the pixels belonging to most major public transit stations in Manhattan had an almost constant stream of taxi demand throughout all hours in the day, with the overnight hours of 12 AM - 5 AM associated with slightly less demand.

Pixel #52586, centered around the NY Port Authority Bus Terminal, is likely to show a different demand pattern. In fact, figure 5.4.3(b) had low demand in the same overnight-hour range, but very quickly picked up from 5 AM through the afternoon. This makes sense, as individuals in this area...
are likely to be traveling during standard business hours, when public buses normally run.

On the other hand, Figure 5.4.4(b) shows how pixel #49355 had a much higher taxi demand late at night. This different activity pattern makes sense as this pixel is centered around 14th St and 9th Ave in the Meatpacking District of Manhattan. This neighborhood has been ranked by a Zagat Survey as the “hottest nightlife neighborhood” in New York City [33], which would explain such a skew towards late-night activities.

These brief snapshots into the behavior of popular taxicab-pickup locations in New York City give a sense for some of the types of areas that could be the best to focus on while considering a ridesharing system implementation. It appears as though focusing on public transit terminals first could be an efficient starting point.

Figure 5.4.1: Pixel #51412 Pickup Analysis

![Cumulative Distribution of Trip Length (Pixel #51412 Origin)](image1)

(a) Trip Lengths CDF by Daypart

![Cumulative Distribution of Pickups (Pixel #51412 Origin)](image2)

(b) Pickups Over Average 24-hour Day
Figure 5.4.2: Pixel #51120 Pickup Analysis

(a) Trip Lengths CDF by Daypart

(b) Pickups Over Average 24-hour Day

Figure 5.4.3: Pixel #52586 Pickup Analysis

(a) Trip Lengths CDF by Daypart

(b) Pickups Over Average 24-hour Day
Figure 5.4.4: Pixel #49355 Pickup Analysis

(a) Trip Lengths CDF by Daypart

(b) Pickups Over Average 24-hour Day

Figure 5.4.5: Pixel #51714 Pickup Analysis

(a) Trip Lengths CDF by Daypart

(b) Pickups Over Average 24-hour Day
Chapter 6

Ridesharing Policies

Now that there has been a thorough introduction to the concept of ridesharing and initial research into its potential existence within New York City, the thesis dives deeper. This next chapter is focused on providing results of thorough simulations of various ridesharing policies implemented at different service levels. A service level refers to the geographical breadth in which a ridesharing service happens to be implemented. For example, a service level of one pixel would mean only one 0.1-by-0.1 mile region in all of NYC is included in the ridesharing system; a service level of 50, on the other hand, would represent a ridesharing service where the taxicab pickups of 50 pixels are individually included and considered.

First, this chapter introduces a specific, more qualitative, selection of pixels throughout New York City that was considered for ridesharing. Subsequent sections then calculate more in-depth figures under various ridesharing policies and frameworks, providing plots that contain a range of service levels and tables that display detailed information for two specific service levels.

As the chapter progresses, the characteristics of the policies behind each ridesharing implementation evolve for multiple reasons. First, the progression in the research was an attempt to build ridesharing policies based on increasingly realistic factors. Second, with each analysis differing slightly from previous analyses, this chapter provides a high number of comparison points for readers to determine which ridesharing policy might make the most sense. Of course, the variety of analyses conducted in the research for this thesis only touches on the possible combinations and alterations of parameters a policy might include.
6.1 Selection of Major Transportation Hubs

Any major form of change in either consumer habits or transportation system operations will not occur overnight, but instead through an initial implementation followed by a continued expansion. Rather than expect the TLC to immediately shift from a zero-ridesharing to a full-ridesharing operation, it makes sense, from a cost perspective as well as a marketing perspective, to begin the ridesharing program in areas within New York City that have a taxicab demand most suited for ridesharing.

As implied from the top-five pixel analysis in the previous chapter, the locations of the most popular and consistent demand for taxicabs in 2013 were centered around major public transportation hubs. Qualitatively these locations make sense for housing a new ridesharing system, as they are packed with commuters who need to go from A to B. Also, these commuters are likely to not have a personal vehicle on hand, since being at a hub means they would have been using other forms of transportation.

The choice was made to simulate the possibility for ridesharing amongst taxicab riders from five specified locations in the city. These locations and pixel selections are depicted in Figure 6.1.1 and include: (A) New York Penn Station; (B) New York Port Authority Bus Terminal; (C) Grand Central Station; (D) LaGuardia Airport; and (E) John F. Kennedy International Airport. Although the analyses in the remainder of this chapter will be calling attention to results of various ridesharing policies being applied over the entirety of NYC, note that every result table (such as Table 6.1 on page 56) includes a section presenting how the results of the same analyses apply to the service level consisting of the five major transportation hubs; this table layout will be repeated throughout the entire chapter.

In order to determine the pixels that would represent these five major transportation hubs for use in later analyses, all of the pixels intersecting the transportation hub were selected. Then, they were filtered to only consist of pixels in the top-1000 rank of pixel pickups. As a further sanity check, the geographic location of the physical structures (or physical terminals in the case of the airports) were double checked with mapping software to ensure that all major entrance- and exit-points of the structures were included in the selected group of pixels. Ultimately 50 pixels were identified to represent these five transportation hubs. These pixels, highlighted in Figure 6.1.1, account for 16 million taxicab pickups over the course of 2013. Though accounting for less than 10% of the total amount of the system’s demand for taxicabs, these 50 pixels are tied to locations that simply make sense when considering the implementation of a ridesharing system due to their high and consistent...
demand. When reading the tables in future sections of this chapter, it is important to keep in mind the difference in magnitude between the two service levels, since two of the metrics (%vMiles and %TaxiRed) are given as a percent of total for the respective service levels.

The identification of the importance of major transportation hubs in the previous section has led to the selection of 50 pixels to represent such areas in this section. It would be logistically much simpler to implement a dedicated ridesharing system at a handful of these locations rather than to do so everywhere in New York City at once. Throughout the remainder of this chapter, ridesharing simulations at this major transportation hub service level will be compared to the service level consisting of the entirety of New York City.

Figure 6.1.1: Major Transportation Hubs Visualization

(a) Penn Station, NY Port Authority Bus Terminal, Grand Central Station
(b) LaGuardia Airport (LGA)
(c) John F. Kennedy Int’l Airport (JFK)
6.2 Analysis I: Major Metrics For Hubs vs. Entire NYC

(CD = 1)

Now that we have identified the specific pixels representing five major transportation hubs within NYC, it is instructive to look analytically at the results of ridesharing policy implemented at these hubs and over the entire city, as it will provide insight into how effective a given framework would be for a range of service levels. This section is focused on a (CD = 1) ridesharing policy, and considers the benefits of implementing it over various service levels for eight frameworks — a superPixel or macroPixel geographic boundary combined with a 30, 60, 120, or 300 second time window. This ridesharing simulation was conducted through the use of the Python code in Section B.5.1 and the subsequent analysis consisting of the code in Section B.5.2, starting on pages 108 and 120 respectively in the Appendix.

6.2.1 Methodology

As summarized in Section 5.2.2, a (CD = 1) ridesharing policy only is concerned about a single common destination between passengers. Following the process described in Section 5.2, this type of analysis matched all trips that fit in a given time window and specified geographic boundary. This would be considered a very primitive ridesharing system, as it did not concern itself with the journey of a taxicab trip. Rather, it only depended on the origin, destination, and temporal information of each trip.

Assuming that trips B and C fit within the time window of trip A, 6.2.1 gives a visual of a basic (CD = 1) analysis. A departure consists of trips that all fit within the specified geographic boundary of each other.

Figure 6.2.1: Logic of (CD = 1) Ridesharing Policy
6.2.2 Vehicle Miles Saved Computation

With the \((CD = 1)\) policy, the number of vehicle miles saved can be determined due to the nature of the trips included in a rideshare. The departure’s total length can computed as the length of the longest trip in the departure list, as this is the farthest distance a shared vehicle will be required to travel. This means that the total vehicle miles saved can be computed by subtracting the distance of the rideshare from the sum of the individual, original trips.

For example, if trips A, B, and C in Figure 6.2.1 were all recorded to be in a single departure under a given ridesharing policy and framework, the taxicab would drive along towards its destination and drop off passengers in a logical and timely manner — closest goes first, farthest goes last. Given the simple spatial nature of a \((CD = 1)\) ridesharing policy, as exemplified by Figure 6.2.1, we can conclude the driver would drop off the passengers from trip A, then the passengers from trip C, and finally the passengers from trip B. This translates to the taxicab of the rideshare departure effectively traveling the distance of trip B. Therefore, the vehicle miles we have “saved” is the sum of the trip A and trip C distances.

6.2.3 Results

When simulating a policy, the Python code (examples attached in Sections B.5.1 and B.6.1) iterated through all of the data for New York City, pixel-by-pixel. For each pixel, it considers the entire year’s worth of data originating from that location and assigns all the trips to departures in a process similar to what was described in Section 5.2. The algorithm does so for each framework and it saves the specific departure pairings. Then, a second algorithm computes the vehicle miles saved as a result of each simulated ridesharing implementation.

To understand the effectiveness of each ridesharing framework as a function of the value it generated on the original NYC TLC trip data, Table 6.1 lists average-valued metrics for all eight ridesharing frameworks. The left half of the table gives the values for a service level consisting of the 50 major transportation hub pixels in Section 6.1, whereas the right half of the table gives the same metrics for the service level of all 2,249 pixels making up 99% of NYC TLC trips. As can be seen from the table by comparing the %vMiles and %TaxiRed metrics, implementing ridesharing with a common destination of 1 at the five major transportation hubs of NYC, as opposed to all of the city, will be roughly two times as effective at both reducing the percent of vehicle miles driven and the percent of vehicles needed to serve the relative taxicab demand.

Though mentioned earlier, the information in this paragraph must be brought to attention again.
Table 6.1: \((CD = 1)\) Ridesharing Policy at Two Service Levels — Transit Hubs vs. All of NYC

<table>
<thead>
<tr>
<th>Rideshare Framework</th>
<th>Major Transportation Hubs</th>
<th>All NYC Pixels</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ADO</td>
<td>% vMiles</td>
</tr>
<tr>
<td>SP,30</td>
<td>1.77</td>
<td>0.62</td>
</tr>
<tr>
<td>SP,60</td>
<td>1.80</td>
<td>1.25</td>
</tr>
<tr>
<td>SP,120</td>
<td>1.85</td>
<td>2.09</td>
</tr>
<tr>
<td>SP,300</td>
<td>1.95</td>
<td>3.95</td>
</tr>
<tr>
<td>MP,30</td>
<td>1.81</td>
<td>1.41</td>
</tr>
<tr>
<td>MP,60</td>
<td>1.88</td>
<td>2.68</td>
</tr>
<tr>
<td>MP,120</td>
<td>1.98</td>
<td>4.25</td>
</tr>
<tr>
<td>MP,300</td>
<td>2.22</td>
<td>7.34</td>
</tr>
</tbody>
</table>

It is crucial to acknowledge that the comparison of the metrics on the left versus right side of the table is comparing how effective different ridesharing frameworks are relative to the service level (or amount of trips) they are applied to. If one were to desire an approximate “absolute” figure for the miles saved by comparing the results in this table, it must be kept in mind that the major transportation hub trips account for roughly 10% of all of NYC’s trips.

As previously described, it may not be the case that a ridesharing system be implemented in all of the geographic regions in New York City at once. However, it makes sense to gauge how a particular form of ridesharing performs at certain service levels — or sub-selections of pixels — not constrained only to the context of major transportation hubs, because a ridesharing system that would ultimately be implemented in real life would ideally be applicable to more than just one type of location.

In order to do so, it is helpful to view the effectiveness of a ridesharing policy as a function of the value it generates for each framework, shown in Figures 6.2.2 and 6.2.3. The cumulative percentage reduction plots in 6.2.2 show the reduction in vehicle miles as a percentage of the total original vehicle miles given an increasing number of pixels added to the overall ridesharing system. The x-axis of both plots indicates the total number of pixels included in the system, so traversing it from left to right increases the service level of the ridesharing simulation one pixel at a time. Furthermore, the x-axis incorporates values in descending order of individual-pixel vehicle mile reduction, meaning that the first pixel along the x-axis is the most productive at reducing departure vehicle miles relative to its own original trip miles. As seen in both plots of Figure 6.2.2, the percent of vehicle miles saved plummeted quickly as the number of pixels involved in the ridesharing framework increased. And, no matter the ridesharing framework, once all of the pixels covering 99% of NYC TLC trips were included, the overall performance of each framework dropped below a 5% reduction in vehicle miles saved.

Figure 6.2.3 takes a different cut at the same ridesharing policy by plotting effectiveness curves
Figure 6.2.2: Taxicab Vehicle Mile Reduction

(a) SuperPixel Boundary

(b) MacroPixel Boundary

Figure 6.2.3: Effectiveness of \((CD = 1)\) Policy Under Different Frameworks
of different-sized service levels. An effectiveness curve considers how well each individual rideshare framework did at reducing total vehicle miles. The x-axis on this effectiveness curve is a discrete ordering of rideshare frameworks by increasing inconvenience to the passenger, where it is less convenient to wait longer before departing and it is less convenient to be included in a destination consisting of a larger geographic area. Recall that a “service level” was defined earlier as a sub-selection of pixels. The different service levels refer to the overall size of the ridesharing service by considering the total number of pixels included: a ridesharing system that serves only 5 pixels worth of geographic area in New York City is significantly smaller than one that services 100; and so on.

To be clear as to how the cumulative and effectiveness plots relate, each of the four service levels — allPixels, top5, top100, top1000 — represent a place along the x-axis of the plots in Figure 6.2.2. In the cumulative plot: one compares the frameworks vertically, by observing each curve’s value at a given service level point along the x-axis; and the service levels horizontally, by tracing along a specific curve. In an effectiveness plot: one compares the different frameworks for a given service level by tracing along the same curve; to compare the performance of different service levels, one moves vertically between the curves at the same point on the x-axis.

As evident in the effectiveness curves of Figure 6.2.3, there was a sharp decrease in the percent of vehicle miles saved when the service level of a ridesharing system was broadened past the ‘top5’ pixels. This observation relates to the sharp, immediate decline of the curves in Figure 6.2.2. Though utilizing a ridesharing system comprising of only five pixels may not be practical, viewing its performance can help show just how effective a policy is — or what its upper performance bound is — across different frameworks. In this case, Figure 6.2.3 highlights how the \((CD = 1)\) ridesharing policy placed more value on the geographic size of the destination than on the time window an initial passenger was required to wait in a taxicab for potential ridesharers. This conclusion can be drawn by tracing along one of the curves from SP\(_{30}\) to MP\(_{300}\) and noticing how the system became more effective with a macroPixel framework, and dropped in effectiveness when switching to a superPixel framework with the next-largest time window. Although this effect is slight at the ‘allPixels’ service level, this plot still makes such a suggestion.

The cause for such a conclusion could be that when considering a policy with a common destination equal to one, there were not enough individuals making such a similarly overlapping trip — in the sense that they would fall within the same superPixel — in a given time window. And so when grouping passengers together, it was more valuable to consider neighboring trips rather than simply waiting a bit longer.
6.3 Analysis II: Evolved Ridesharing Policy

\((CD = 3, \text{CIR} = 0.2)\)

While the previous section conducted an analysis of a ridesharing policy on the five major transportation hubs, the entirety of New York City, and various service levels in between, the policy itself was rudimentary. The policy in this section adds layers of complexity while maintaining an analysis on similar service levels. This analysis considers a ridesharing policy where each departure had a maximum of three common destinations, and where each additional destination did not add more than 20% circuity to the overall distance of the departure — both concepts described earlier in Section 5.2.2.

6.3.1 Methodology

By allowing for additional destinations in a departure, the process of finding rideshare matches increases in complexity. To consider all possible rideshare candidates for each departure, this \((CD = 3, \text{CIR} = 0.2)\) policy followed the same iterative searching algorithm described in Section 5.2, but the process for determining which passengers can be assigned to the same rideshare departure depended on much more information.

An example of such a scenario is depicted in Figure 6.3.1 to go along with the bulleted explanation below. One has to keep in mind, when considering such a diagram for a variable number of destinations, that once the maximum number of destinations is reached, no more destinations can be created. Although this section involves a policy where \((CD = 3)\), the explanation of the \((CD = x)\) ridesharing methodology in Figure 6.3.1 uses an example with \(x > 3\); so let us claim that for the example diagram in Figure 6.3.1 and its related steps below, \(x = 5\).

For the purposes of the diagram logic in Figure 6.3.1, assume that the trips within the “Departure List” all fit within the time window of the first trip, they are all unassigned, and they represent all of the rideshare possibilities meeting the defined policy requirements for a departure. Of course in the practical application of this policy’s algorithm, any number of trip assignments can occur and it is likely that some of the trips in the sorted list be skipped as they are already in an existing departure.

- **Step 1.** A trip (trip A) that does not fit into any existing departures is assigned to a new departure. The pixel it belongs to is marked as the center of the pre-specified geographic boundary, creating a geo-fence around the first destination.
Figure 6.3.1: Scenario of Common Destination and Circuity Logic
• **Step 2.** The next trip (trip B) is identified as a possibility for rideshare. It does not fit within the geo-fence of the first destination, and so it is considered a potential rideshare only if the circuity of such an addition is less than or equal to the required circuity ratio and if the policy allows for another destination in the departure. In this case, the new total departure distance is less than or equal to the current total departure distance and \( x = 5 \) so the destination and trip are added.

• **Step 3.** A third trip (trip C) is considered. It does not fall within the geo-fence of either destination in the departure, but since the common destination limit \( (x = 5) \) is greater than the current destination count of two, trip C is considered as its own destination similar to how trip B was considered. As we can see in the figure, the circuity ratio of the new total trip distance compared to the old is not violated. Thus, trip C is added to the existing departure.

• **Step 4.** A fourth trip (trip D) is considered. As previously mentioned, all trips are unassigned and meet the ridesharing framework’s time window constraint. Since trip D falls within the geo-fence of one of the existing destinations, it is added to the existing departure immediately and circuity does not need to be checked. Next, a fifth trip (trip E) is identified for consideration; the same logic ensues and it is added.

• **Step 5.** The last remaining trip (trip F) of all of the possibilities is considered. It does not fall within the geo-fences of the existing destinations, so it is considered as its own destination since the current destination count is less than five. Keep in mind that, if the common destination limit had been \( x = 3 \), then the ridesharing algorithm would no longer consider new destinations and would rather look to see if any additional trips fit within the geo-fences of the three existing destinations. In this case trip F is not added to the departure as its addition to the existing departure does not satisfy the circuity constraint.

• **Final Step.** The path and departure list in this step make up the departure found for trip A under the given ridesharing policy and ridesharing framework. Note that in this particular case, trip F is still not assigned to a departure. If this were the actual last un-matched trip in an entire trip file (like in this example, where only six trips exist), the departure would only consist of one trip; if it were not the last trip, the logic would repeat itself with trip F acting as the first trip in another departure.

As was hopefully clear when reading through the steps above and following along with Figure 6.3.1, the geographic boundary of a framework does not affect the destination assignments
within a departure as it relates to circuity. The boundaries simply allow for more trips to join in a rideshare without adding a new destination if they are similar to a trip already existing in the rideshare.

6.3.2 Vehicle Miles Saved Computation

The manner in which the reduction in vehicle miles was computed in this type of policy differs from that of the \((CD = 1)\) policy. This policy assumed that, for each destination in a departure, the associated “jiggle” within its geo-fence is negligible compared to the total departure distance. Therefore the calculation was as follows.

Once a complete departure and departure trip list was created, as shown by the Final Step in Figure 6.3.1, an original trip distance metric was computed, which was the sum of all of the individual trips that were included in the departure. Rather than use the odometer mileage like in the previous policy, each distance was estimated by the Manhattan distance between a trip’s origin and destination. The Manhattan distance function is commonly used and accounts for the grid nature of city blocks by multiplying the cartesian distance between two points by a factor of 1.2. The total original trip distance metric represents the total vehicle miles that were actually driven in 2013 with no ridesharing in effect.

Next, the rideshare departure total distance was found by computing the shortest path between the centroids of each destination within a departure. The only requirement for such a shortest-path computation was that the path had to start at the departure origin. Although a departure keeps track of the order in which destinations were added to a rideshare departure, this computation sacrificed such an order if it meant finding a different permutation of departures with a shorter total path distance. This type of compromise is something that a taxicab driver would make if presented with a list of destinations to reach in as short a time as possible.

Finally to calculate the \(\%vMile\) metric, the difference between the first and second totals were divided by the first total. Although this metric is a percentage relative to the size of the service level, for the specified major transportation hub and “all pixels” service levels, the original vehicle miles of the trips are the same across all of the analyses (because the pixel selections are the same). This means that this thesis is able to compare the \(\%vMile\) metrics as if they were an actual physical quantity of vehicle miles saved.
6.3.3 Results

As the tables and plots in subsequent sections of this chapter are similarly structured to the previous analysis, only notable differences will be brought to light.

The shift from a \((CD = 1)\) policy to a \((CD = 3, CIR = 0.2)\) policy resulted in a major improvement in ridesharing effectiveness, particularly with the shorter time windows. This improvement can be seen by comparing Tables 6.1 and 6.2, where the \%vMiles increased by an average factor of 13 for the combination of both superPixel and macroPixel in the 30-second time window. Meanwhile, the ridesharing effectiveness of the geographic boundaries in the 300-second time window increased on average by a factor of 8.

Table 6.2: \((CD = 3, CIR = 0.2)\) Ridesharing Policy at Two Service Levels

<table>
<thead>
<tr>
<th>Rideshare Framework</th>
<th>Major Transportation Hubs</th>
<th>All NYC Pixels</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ADO</td>
<td>% vMiles</td>
</tr>
<tr>
<td>SP_30</td>
<td>1.98</td>
<td>12.74</td>
</tr>
<tr>
<td>SP_60</td>
<td>2.23</td>
<td>21.11</td>
</tr>
<tr>
<td>SP_120</td>
<td>2.53</td>
<td>28.71</td>
</tr>
<tr>
<td>SP_300</td>
<td>3.16</td>
<td>38.78</td>
</tr>
<tr>
<td>MP_30</td>
<td>1.99</td>
<td>12.92</td>
</tr>
<tr>
<td>MP_60</td>
<td>2.24</td>
<td>21.44</td>
</tr>
<tr>
<td>MP_120</td>
<td>2.56</td>
<td>29.33</td>
</tr>
<tr>
<td>MP_300</td>
<td>3.24</td>
<td>40.12</td>
</tr>
</tbody>
</table>

It makes sense that as the policy shifted toward one with multiple destinations per departure, the frameworks within smaller time windows experienced a more significant increase. A single common destination resulted in the ridesharing system considering a very small fraction of trips for potential matches due to its exclusive geographic nature; add onto this a very short time window and the number of rideshare opportunities became scarce. But a policy with a larger number of common destinations allowed a given rideshare departure to have more wiggle-room in terms of determining different combinations of the trip possibilities to be included.

The cumulative percentage reduction plots in Figure 6.3.2 and the ridesharing effectiveness curves in Figure 6.3.3 further emphasize this dramatic improvement from the previous policy.

Under a given time-window, a ridesharing framework with a macroPixel geographic boundary must always outperform one with a superPixel boundary by nature of the fact that a macroPixel always contains a superPixel. As was expected, for each time window the macroPixel framework outperformed the superPixel framework. However, the effectiveness curves of Figure 6.3.3 indicate that there was a much less significant difference between the two geographic boundaries as compared to the \((CD = 1)\) policy. Again, this can be explained through the nature of the ridesharing policy. Once the new policy allowed for a significantly higher number of ridesharing possibilities, the
importance of the geographic boundary diminishes and that of the length of time window increased.

Figure 6.3.2: Taxicab Vehicle Mile Reduction

(a) SuperPixel Boundary
(b) MacroPixel Boundary

Figure 6.3.3: Effectiveness of \((CD = 3, CIR = 0.2)\) Policy Under Different Frameworks
6.4  Analysis III: Directional Considerations

\((CD = 3, \text{dCIR} = 0.2)\)

This next analysis represents the last major wrinkle in the methodology behind a given ridesharing policy — *directional* circuity. While the policy in the previous section took into account relative direction through the use of its circuity metric — eliminating the addition of any destination that would be too inconvenient — it did not account for the initial direction of the first destination added to a departure. The manner in which the concept of directional circuity works is explained in the following sub-section.

The Python code for simulating this \((CD = 3, \text{dCIR} = 0.2)\) policy is attached in the Appendix starting on page 124. Similar to the previously attached code, this includes both the scripts for simulating the rideshare (B.6.1) and for conducting the departure analysis (B.6.2). In order to limit the number of pages, only the code for \((CD = 1)\) and \((CD = 3, \text{dCIR} = 0.2)\) are attached to this thesis. However please note that the code for the previous section’s \((CD = 3, CIR = 0.2)\) policy is similar to that in Section B.6. The only major difference involves the boolean at line 902 of the code in Section B.6.1; in the `common_destination_x_ridesharing()` function, ridesharing using non-directional circuity logic replaces the `directionalCircuitySatisfied()` function with `circuitySatisfied()`. Moving forward, the most significant differences in code for the subsequent analyses involve changing the common destination and circuity parameter values in the code.

6.4.1  Methodology

As established in the previous section, a policy that allows for more than one common destination opens itself up to the possibility for a much more diverse set of passenger combinations when creating a rideshare departure, and therefore a greater reduction of taxicab vehicle miles. Although a rideshare policy should allow for slight deviations in an originally intended route in order to include additional passengers, current passengers would become irritated, if not irate, when their departure heads in the direction opposite from their intended destination. Upgrading a policy from circuity \((CIR)\) to directional circuity \((\text{dCIR})\) ensures this does not occur.

For directional circuity, the ridesharing system follows the process visualized in Figure 6.3.1 and described earlier in Section 6.3.1. However the system pays attention to one more detail.

When the first trip in a departure is recorded and represents the departure’s first destination, its location relative to the originating pixel is considered. If the destination is further from the origin
pixel in the North-South orientation than it is in the East-West orientation, the policy eliminates all trips on the opposite hemisphere created by a horizontal division. In Figure 6.4.1(a), due to $\Delta y$ being larger than $\Delta x$ and the destination being located north of the origin, all the pixels below the origin are removed from consideration for that specific departure. We can easily imagine the opposite event in which all of the pixels above the origin are removed if the destination is located south of the origin.

Figure 6.4.1: Directional Circuity Logic

Likewise, if the destination’s centroid results in a $\Delta x$ that is larger than $\Delta y$, a vertical division is created and the opposite hemisphere of pixels is removed from consideration for that departure. The vertical removal of the west hemisphere is shown graphically in Figure 6.4.1(b).

The inclusion of directional circuity to the logic of ridesharing policies helps take into account the convenience and emotional capacity of a passenger who has agreed to rideshare. Directional circuity recognizes that passengers would not be pleased to witness the vehicle head in the direction opposite of their intended destination if they feel in a rush.
6.4.2 Results

With the exception of that explained in Section 6.4.1, there are no fundamental differences between the logic of the policies that adjust for directional circuity and those that do not — \((CD = 3, CIR = 0.2)\) vs. \((CD = 3, dCIR = 0.2)\) — but there is a fundamental difference between the logic of policies when incorporating additional common destinations — \((CD = 1)\) vs. \((CD = 3, CIR = 0.2)\). The comparative results of \((CD = 3, CIR = 0.2)\) versus \((CD = 3, dCIR = 0.2)\) reflect such a relative lack of fundamental difference in logic as opposed to what was witnessed in the previous section’s comparison.

By comparing the cumulative percent reduction curves and the effectiveness curves of this policy (Figures 6.4.2 and 6.4.3) and of that in the previous section (Figures 6.3.2 and 6.3.3), the policy performances look almost identical to the human eye.

Figure 6.4.2: Taxicab Vehicle Mile Reduction

Analyzing the differences in the raw values of the different metrics in Tables 6.2 and 6.3 helps give a better insight. Specifically, it can be seen that a policy with the exact same CD and CIR inputs appear to have been more effective when it included directional circuity. Under the directional circuity policy, the \%vMiles metric managed to increase in every framework despite being slight — an average of only 0.4\% across all frameworks. This result is promising, as the application of directional circuity not only is appreciated by passengers, but also enhances the overall effectiveness of a ridesharing system by reducing the number of vehicle miles driven by taxicabs.

A reason for such a slight increase in \%vMiles could be due to the manner in which the system’s algorithm selects rideshare pairings. Under the older, non-directional, policy, as long as a new
destination did not increase the circuity by too much, it was added to an existing departure even if there may have been a better departure occurring at roughly the same time. Whereas this newer policy uses the directional circuity logic to better verify that such an addition of a trip is indeed the “best” assignment.

Table 6.3: \((CD = 3, dCIR = 0.2)\) Ridesharing Policy at Two Service Levels

<table>
<thead>
<tr>
<th>Rideshare Framework</th>
<th>Major Transportation Hubs</th>
<th>All NYC Pixels</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ADO</td>
<td>% vMiles</td>
</tr>
<tr>
<td>SP_30</td>
<td>1.97</td>
<td>12.77</td>
</tr>
<tr>
<td>SP_60</td>
<td>2.21</td>
<td>21.18</td>
</tr>
<tr>
<td>SP_120</td>
<td>2.51</td>
<td>28.86</td>
</tr>
<tr>
<td>SP_300</td>
<td>3.11</td>
<td>40.95</td>
</tr>
<tr>
<td>MP_30</td>
<td>1.98</td>
<td>12.95</td>
</tr>
<tr>
<td>MP_60</td>
<td>2.23</td>
<td>21.52</td>
</tr>
<tr>
<td>MP_120</td>
<td>2.54</td>
<td>29.47</td>
</tr>
<tr>
<td>MP_300</td>
<td>3.20</td>
<td>40.38</td>
</tr>
</tbody>
</table>
6.5 Analysis IV: Reducing Inconvenience

\((CD = 3, dCIR = 0.1)\)

Although there is no existing ridesharing system to give us a sense of what is an “appropriate” upper-limit of circuity for individuals to feel the desire to partake in such a service, it will likely be low. Therefore it is informative to study the effect of ridesharing system policies with various upper-limits of inconvenience to the passengers. Other than a change of the directional circuity metric to 0.1, the ridesharing policy analyzed in this section is identical to the policy from the previous section.

6.5.1 Results

A reduction of the directional circuity of this analysis, from 0.2 to 0.1, logically limited the number of viable rideshares and thus one would expect an overall decrease in rideshare effectiveness.

By comparing Tables 6.3 and 6.4, it can be calculated that the percent reduction in policy effectiveness ranged from 11\% for the MP_300 framework to 24\% for the SP_30 framework. It makes sense that the more narrow frameworks experienced a greater magnitude of underperformance under less favorable parameters. The corresponding drop in performance is noted by the overall decrease of the curves in Figures 6.5.1 and 6.5.2 relative to the corresponding curves in the previous section.

<table>
<thead>
<tr>
<th>Rideshare Framework</th>
<th>Major Transportation Hubs</th>
<th>All NYC Pixels</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ADO</td>
<td>% vMiles</td>
</tr>
<tr>
<td>SP_30</td>
<td>1.90</td>
<td>9.12</td>
</tr>
<tr>
<td>SP_60</td>
<td>2.07</td>
<td>16.30</td>
</tr>
<tr>
<td>SP_120</td>
<td>2.29</td>
<td>23.57</td>
</tr>
<tr>
<td>SP_300</td>
<td>2.78</td>
<td>34.61</td>
</tr>
<tr>
<td>MP_30</td>
<td>1.91</td>
<td>9.33</td>
</tr>
<tr>
<td>MP_60</td>
<td>2.09</td>
<td>16.67</td>
</tr>
<tr>
<td>MP_120</td>
<td>2.32</td>
<td>24.19</td>
</tr>
<tr>
<td>MP_300</td>
<td>2.86</td>
<td>35.83</td>
</tr>
</tbody>
</table>

A reasonable interpretation of the decrease in rideshare effectiveness of the \((CD = 3, dCIR = 0.1)\) policy is that the entire system was negatively affected when its constraints were pulled very tight. Although more research must be conducted to determine which types of circuity ratio limits correspond to which feelings, or levels, of inconvenience, a directional circuity limit of 0.1 is likely about as stringent as a ridesharing system would want to get. Otherwise, there may be no significant benefit in even implementing such a system.
Figure 6.5.1: Taxicab Vehicle Mile Reduction

(a) SuperPixel Boundary
(b) MacroPixel Boundary

Figure 6.5.2: Effectiveness of \((CD = 3, dCIR = 0.1)\) Policy Under Different Frameworks

Performance of Rideshare Frameworks
\((CD = 3, dCIR = 0.1)\) Policy
6.6 Analyses V and VI: More Destinations

\((CD = 5, dCIR = 0.1)\) and \((CD = 5, dCIR = 0.2)\)

This last section presents the final two ridesharing policies. \((CD = 5, dCIR = 0.1)\) and \((CD = 5, dCIR = 0.2)\) have logic similar to the previous two sections’ policies, except that they explore the possibility of having the potential for up to five common destinations within one rideshare departure. Of course, as many as five common destinations are not required; destinations are added to an existing departure, up to a maximum of five, if they do not result in a percent increase in trip distance greater than the circuity limit.

6.6.1 Results

Further increasing the number of potential common destinations provided some interesting results. For the \((CD = 5, dCIR = 0.1)\) policy, the frameworks with shorter time-windows experienced a slight increase in \%vMiles compared to \((CD = 3, dCIR = 0.1)\). Meaning that if a ridesharing system is only waiting for a small amount of time, having the possibility to add more destinations to a departure is not significant. This is likely because, on average, there simply are not enough passengers to make a dramatic difference in filling all of the departure requirements and be going to a location that is a sensible additional-destination for a departure. This holds true for both the superPixel and macroPixel frameworks, verifying an earlier observation that policies with larger common destination limits are less affected by differences in geographic boundary size.

As can be seen from comparing Table 6.5 to the previous Table 6.4, when a framework consisted of a larger time window, the addition of two more possible common destinations increased the additional ridesharing policy efficiency to about 5\% larger than that of the policy with three common destinations.

<table>
<thead>
<tr>
<th>Rideshare Framework</th>
<th>Major Transportation Hubs</th>
<th>All NYC Pixels</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ADO</td>
<td>% vMiles</td>
</tr>
<tr>
<td>SP_30</td>
<td>1.90</td>
<td>9.30</td>
</tr>
<tr>
<td>SP_60</td>
<td>2.08</td>
<td>16.83</td>
</tr>
<tr>
<td>SP_120</td>
<td>2.53</td>
<td>24.84</td>
</tr>
<tr>
<td>SP_300</td>
<td>2.91</td>
<td>37.65</td>
</tr>
<tr>
<td>MP_30</td>
<td>1.91</td>
<td>9.47</td>
</tr>
<tr>
<td>MP_60</td>
<td>2.09</td>
<td>17.09</td>
</tr>
<tr>
<td>MP_120</td>
<td>2.35</td>
<td>25.21</td>
</tr>
<tr>
<td>MP_300</td>
<td>2.96</td>
<td>38.28</td>
</tr>
</tbody>
</table>

Next, a similar comparison between \((CD = 5, dCIR = 0.2)\) and \((CD = 3, dCIR = 0.2)\) in
Tables 6.6 and 6.3 show almost the same effect, but with a greater magnitude. This makes sense, as the larger common destination bound of predictably allows for many more ridesharing opportunities. It is the larger directional circuity limit — double in size, in fact — that pushes the percent difference between the effectiveness of these two policies to be roughly double that of the change between the \((CD = 3, dCIR = 0.1)\) and \((CD = 5, dCIR = 0.1)\) policies; this performance result holds true for each of the eight frameworks. Comparing the figures in this section with those from the last two sections visually confirms these conclusions.

The difference between the effectiveness of \((CD = 5, dCIR = 0.2)\) and \((CD = 5, dCIR = 0.1)\) was much larger than the previous two comparisons, implying that if only one parameter were to change, the circuity of a ridesharing policy would be most impactful to the system’s performance. Actual percent changes can be calculated by comparing the Tables 6.5 and 6.6, but they range from a 15\% increase in effectiveness (for MP_300) to 33\% (for SP_30). Both these quantities and the percent differences are larger than the similarly computed quantities when comparing the policies with fewer common destinations — \((CD = 3, dCIR = 0.2)\) and \((CD = 3, dCIR = 0.1)\).

Table 6.6: \((CD = 5, dCIR = 0.2)\) Ridesharing Policy at Two Service Levels

<table>
<thead>
<tr>
<th>Rideshare Framework</th>
<th>Major Transportation Hubs</th>
<th>All NYC Pixels</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ADO</td>
<td>% vMiles</td>
</tr>
<tr>
<td>SP_30</td>
<td>1.98</td>
<td>13.24</td>
</tr>
<tr>
<td>SP_60</td>
<td>2.24</td>
<td>22.35</td>
</tr>
<tr>
<td>SP_120</td>
<td>2.59</td>
<td>31.29</td>
</tr>
<tr>
<td>SP_200</td>
<td>3.40</td>
<td>43.86</td>
</tr>
<tr>
<td>MP_30</td>
<td>1.99</td>
<td>13.36</td>
</tr>
<tr>
<td>MP_60</td>
<td>2.25</td>
<td>22.54</td>
</tr>
<tr>
<td>MP_120</td>
<td>2.61</td>
<td>31.59</td>
</tr>
<tr>
<td>MP_300</td>
<td>3.45</td>
<td>44.51</td>
</tr>
</tbody>
</table>

The relative difference in performance due to the circuity of a policy — \((CD = 5, dCIR = 0.2)\) versus \((CD = 5, dCIR = 0.1)\) — can be viewed through observing Figures 6.6.1 – 6.6.2. All of these curves have a noticeable difference in effectiveness when comparing ridesharing at a smaller service level. Yet even when comparing at an “allPixel” service level, there is still an observable difference in performance.

As a ridesharing system for New York City, it would make sense to have a high upper-bound for the number of common destinations allowed in a rideshare departure, regardless of the circuity parameter. If such a trip exists that could tack onto an existing departure, the system would allow it to do so.

The circuity, or directional circuity, parameter is a different matter. The circuity parameter of a ridesharing policy is critical to the success of ridesharing system because people do not prefer to
Figure 6.6.1: Taxicab Vehicle Mile Reduction

(a) SuperPixel Boundary (dCIR = 0.1)

(b) MacroPixel Boundary (dCIR = 0.1)

(c) SuperPixel Boundary (dCIR = 0.2)

(d) MacroPixel Boundary (dCIR = 0.2)
wait. Individuals only have so much willingness to be inconvenienced by tacking on more pickups, and thus more time, to their taxi ride. As it relates almost directly to the amount of inconvenience a passenger can feel while taking the trip, it cannot be raised too high or passengers will not feel motivated to rideshare.

Therefore from the results gleaned through these analyses, this thesis would recommend a ridesharing system in New York City with a high number of common destinations, and a relatively low directional circuity bound.

6.7 Concluding Remarks

The analyses within this chapter helped bring to light different viewpoints about how various ridesharing systems could affect the New York City TLC service. When considering a realistic initial application of a ridesharing system, major transportation hubs are likely the best location to start in order to acquaint the city’s inhabitants with the new concept. Then, as it begins to grow in popularity, the system could be expanded to a wider level of service throughout the city.

Across all policies, as would be expected, adding larger time windows to frameworks resulted in an increase in the effectiveness of ridesharing. From interpreting the results, ridesharing policies with a common destination of 1 tended to place more value on the size of a destination’s geographic boundary than on the length of time window. As ridesharing evolved to include more destinations,
the effectiveness of such policies escalated; this makes sense as a larger number of destinations results in a system that is more flexible to mold to the existing trip demand and squeeze out every last rideshare. As opposed to \( CD = 1 \), rideshare policies with more common destinations placed less value on the specified geographic boundary surrounding each destination.

When accounting for the inconvenience of passengers, ridesharing policies with a lower circuity metric logically were less effective. However, the inclusion of directional circuity into a policy — which is impactful in terms of its emotional effect on passengers — increased performance.

Lastly, as mentioned in the previous section, a likely “best practice” ridesharing policy for New York City taxicabs would consist of one with a high number of common destinations and a relatively low circuity ratio. Although a more stringent directional circuity value is less effective, it is more realistic because it results in less inconvenience for the passengers. Further (and ideally, real-world) testing would be necessary to have a better understanding of the appropriate circuity levels for a ridesharing implementation.
Chapter 7

Limitations, Next Steps, and Conclusion

7.1 Limitations

7.1.1 Data Concerns

As with any massive data set collected from real life experience, it is bound to contain anomalies or inconsistencies that would not exist with synthetically-generated data. Section 2.4 on page 18 lays out a series of actions that were taken to reduce data concerns prior to conducting research and analysis for this thesis. Although these actions removed a plethora of technical errors and eliminated over 4 million entries, or 2.4% of the 170 million in the original 2013 NYC TLC data set, the reduction is unlikely to greatly affect the outcome of any analysis. Of course it remains possible that additional errors exist within the data set, that mistakes in the manner in which the system collected data are still present, or that the process by which each individual driver records the data is imperfect. Nevertheless, it is reassuring to consider that the sheer size of this data set reduces the effect of any biased data.

The pixelization process outlined in Chapter 4 was unquestionably the key to unlocking the massive amount of data for analysis. As mentioned in Section 5.2, the methodology involved constructing a grid system over the New York City area and assigning each trip origination and destination to the specific pixel they fell within. While the pixelization of the 2013 NYC TLC data allowed for the analyses conducted in this research, it did add a layer of ambiguity to the results. This thesis, in
conjunction with past transportation research conducted by the ORFE department, assumed that all trips assigned to a pixel depart (or arrive) at the pixel’s centroid. This assumption removed a slight amount of accuracy, as it was certainly possible that not all trip-points within a pixel averaged out to the center. However for this particular research, as each pixel was only 0.1-by-0.1 miles, the assumption becomes less relevant as it is feasible to presume individuals are willing to walk a small distance if they see a waiting taxicab.

7.1.2 Evolving Transportation Demand

Transportation demand is constantly changing. As humanity, technology, and even the dynamics of New York City evolve, it is unrealistic to assume that any two periods of time will be exactly the same. The research in this thesis was only able to consider NYC TLC trip data from 2013, while the conclusions drawn from such data were assumed to be true moving forward. This research also assumed that the taxicab fleet characteristics of 2013 have not significantly changed up to today. Technically it is reckless to extrapolate results and characteristics to a time period outside of the research sample’s time frame — current and future taxicab demand as it relates to this thesis — however such an act is unavoidable in this type of academic research as real-time data is not feasible.

Similar to the problem of assuming 2013 TLC demand can be extrapolated to future time periods, it is unrealistic to assume that there are no changes in the competitive landscape for taxicabs within New York City. Over the past few years, the number of private companies providing taxicab and ridesharing services similar to those of Uber and Lyft have multiplied. Concurrent with such an expansion, the number of individuals who use and are familiar with their services has skyrocketed. 2013 was a year in which these types of services existed in New York City, but were not nearly as prevalent as today. It must be admitted that their increase in popularity has altered, and continue to alter, the landscape of NYC TLC demand and the role that taxicabs play in the lives of the city’s inhabitants. Nevertheless, the value that real-life data brings — as opposed to synthesized data — is significant enough to live with the imperfect assumptions that the NYC TLC historical data set is representative of the current demand and environment.

Finally, this thesis uses NYC TLC trips as a proxy for all personal, vehicular transportation demand within New York City. The city, especially Manhattan, has enormous subway, rail, and bus systems all operating simultaneously to serve inhabitants. Plus, there are a substantial number of individuals who walk and bike. The use of the words “personal” and “vehicular”, however, make the proxy used in this thesis a reasonable one for NYC.
7.1.3 Departure Passenger Count

Another assumption made by this thesis was that the size of a rideshare was not limited, as long as the trips fit the policy’s specifications. Although this did not happen frequently, such an assumption resulted in some trips containing a much larger count of passengers than could be satisfied in real life — most taxicabs can serve no more than 6 passengers at one time. An actual implementation of a ridesharing service would require that such a consideration be taken; but as this research focused on the potential for ridesharing, it was preferable to leave such a restriction out in order to view the true performance of each policy.

7.1.4 Run-Time and Memory Limitations

As could be expected, the sheer size of the rideshare simulations proved to be a major challenge and a bottleneck for the progress of this research. Significant effort was expended in addressing computational issues while attempting to find the most practical method for running analyses on this data.

Initially, a PostgreSQL database was set up on an external server through the ORFE department, but connection to the server and the size of data that would be generated was too large for effective use. After other failed attempts, the ultimate solution consisted of using external hard drives on multiple desktop computers, running Python scripts all on the same major CSV file containing the trip data.

To give a sense as to the run-time and related limitations, a single rideshare simulation took roughly 12 hours to run when split over three, simultaneously-running desktop computers. Each simulation generated roughly 300 GB of data. This data needed to be saved on external hard drives for multiple reasons: to not crash the computer (or computers) being used; to prohibit the need to re-run the simulation’s 36 hours of computation if needed later; and most importantly, to be able to run the ridesharing analysis scripts on the simulation outputs. The storage capacity created a separate issue for the research in this thesis and limited the number of analyses to be run.

The data manipulation and pixelization Python scripts (included in the Appendix) were not as computationally intensive, but were run on a local MacBook Pro and took longer due to the lack of a parallel computing process during their execution.
7.2 Next Steps

This section elaborates on topics that could help shed more light on the revolutionary technological concept of ridesharing.

7.2.1 Focus on Vehicle Mile Reduction

For any of the ridesharing policies assigned to New York City, there was a clear disparity of vehicle mile reduction percentages across the 2,249 pixels. As repeatedly emphasized in this thesis, such a reduction is the crux of implementing a new transportation system. As found in this research, specific areas possessed a much greater capability for effective ridesharing than others; these regions have the potential to be a stepping stone towards a full, city-wide adoption of a program. Continued research should be devoted towards determining the sets of characteristics — both in terms of the geographic locations and the ridesharing policy parameters — that lead to a high percent reduction of vehicle miles, so that an actual ridesharing system could be tuned to maximize efficiency at every point.

7.2.2 More Intelligent Departure Grouping

The methodology of rideshare departure creation, described in Section 5.2, grouped existing passengers into rideshares if they fit the correct specifications and had yet to be included in an existing rideshare. In other words, there was no distinct algorithm that determined whether a particular trip should join departure A, departure B, etc.

Future research that not only considers the policy parameters described earlier when creating departures, but also analyzes corresponding trip characteristics such as length of trip would be valuable. For example, it could be informative to simulate a ridesharing policy that groups short trips with short trips, and long trips with long trips, assuming that they still satisfy the existing ridesharing requirements.

Another consideration would be to define the concept of directional circuity in a more exclusive sense. Rather than limit the other possible rideshare matches to a specific hemisphere of the pixels, they could be limited to a specific quadrant or an even smaller filament of pixels extending some distance on either side of a line drawn between the origin and the first destination.

These next steps would bring benefits in that they could likely make ridesharing more amenable to passengers, more impactful, and more operationally cost effective.
7.2.3 Ridesharing Policy Variation

The simulations conducted in this thesis’s research remained constant throughout the entire New York City region and the entire year of 2013. However a major benefit resulting from running such a service over a mobile application is that a policy does not need to be static.

In fact, enabling ridesharing policies to vary according to demand would almost certainly be better. For example, it could prove instructive to consider a simulation consisting of different ridesharing policy structures for the various neighborhoods of the five boroughs. Areas with higher demand could benefit from more closely matched assignments and taxicab pickups in low demand regions could do the opposite in order to extend ridesharing potential while not inconveniencing the passengers in high-demand areas. Looking at a set of different policies applied uniformly across all pixels in this research, different characteristics of New York City areas were brought to light. But further work can shed more light on which locations are best-suited for a specific types of ridesharing policies.

Another application for such further study would be to consider the effect of observing a longer time window during off-peak hours. Because it would not be during rush hour, the chance that an individual would be willing to wait longer increases. Additionally because it is not the peak time for taxicab demand, this longer time window would result in a higher potential for reducing vehicle miles, thus contributing toward making the entire system more effective. Countless variations and skews could be made on the more traditional, uniform application of the policies laid out in this research.

More detailed and intentionally-varying policies could be developed that adjust parameters over the course of a day, throughout the entire year, depending on the relative geography of certain pixels, etc. Once again, Uber serves as a case-study as they initiate price surges for fares in particular locations and during specific time frames, depending on the current demand for taxicabs. Observing passenger response to such non-uniform services could help shape planning for future, more intelligent rideshare systems with variable policies.

7.2.4 Additional Weather Analyses

Outside of a pure transportation need — getting from point A to point B — understanding what else motivates individuals to take taxicabs can lead to a more applicable ridesharing system. An analysis conducted in Chapter 3 looked only at one dimension of how weather events could be correlated with taxicab demand and usage. Further and more in depth analyses of precipitation,
temperature, humidity, etc. could prove useful.

7.2.5 Unrestricted Ridesharing Policy

Similar to how viewing the effectiveness of a ridesharing simulation with an extremely low service levels provides perspective as to a “best case scenario”, it would prove useful to conduct an ultimate rideshare simulation to understand the extent to which vehicle miles are reduced.

Such an analysis would involve placing individuals in a ridesharing departure from a specific pixel as long as they arrive within the appropriate time window, irrespective of where they are going. Effectively, this would be a ridesharing policy that would have no limit on common destinations and would not consider any form of circuity. Once all of the individuals are piled into one departure, the system would have to determine the optimal sequence of destinations to drop passengers off in order to reduce the total departure distance (and therefore minimize the overall travel time).

Given the Python code from the previous analyses, creating this analysis would not prove too difficult. All it would require to simulate this analysis is to: first, re-arrange and briefly edit of the main simulation logic in the common_destination_ridesharing() function of the Section B.6.1 code; and second, update the logic of computing vehicle miles saved in the pullVehicleMilesSaved() function of the Section B.6.2 code.

This type of a simulation is not a policy that would be feasible to implement due to the inconvenience it would cause passengers, the capacity of taxicabs, and more. But again, it would be informative to understand the degree to which an “unrestricted” ridesharing service can reduce vehicle miles.

7.2.6 aTaxi Fleet Analyses

In many sections throughout this thesis, the benefits of and opportunities for a fleet of self-driving cars were explained. However, no thorough analysis was conducted on key attributes that would impact an autonomous taxi network.

One piece of work, mentioned several time throughout the thesis, would be to create a thorough taxicab repositioning and allocation system based on the 2013 NYC TLC data. This would help self-driving cars determine the optimal direction to start heading in whenever empty in order to most quickly find a new passenger. Such a system could be based on more than simply historical taxicab demand, as these self-driving cars could communicate with each other to know which areas of the city are too concentrated with taxicabs and which areas are in need of supply. Ideally, this kind
of a repositioning system would minimize the amount of taxicabs acting as “purposeless vehicles” aimlessly driving throughout the city.

Another direction of research, focused on the potential of self-driving cars, would be to determine an adequate Taxi fleet size. Given the historical demand over 2013 and assuming that all taxicabs in New York City were to be autonomous taxis, one could find what the minimum number of taxicabs would be to serve all of the demand under different ridesharing policies. This kind of an analysis provides a quantitative understanding as to just how much congestion can be reduced through the effective implementation of ridesharing and self-driving cars.

7.3 Conclusion

The United States is in desperate need of a reduction in personal vehicle miles. Given the current, thoroughly developed status of the networked transportation infrastructure throughout the country, smarter transportation is a strong alternative that progresses the current system. Effective and ubiquitous ridesharing is the key to evolving a smarter transportation system on the existing road network. Such an evolution will help reduce wasted fuel and pollution levels, shrink dependence on foreign oil, and decrease congestion levels all while increasing overall transportation safety and the availability of lower-cost transportation options.

Unlike many technologies that require future development, a widespread ridesharing system has the potential to be installed immediately and to be working quickly. A significant benefit is the manner by which ridesharing handles spikes in demand — both expected and unexpected. As taxicabs are able to handle multiple destinations in a single departure, the number of additional taxicabs required to satisfy an increase in demand is not directly proportional. If more riders were to switch from using personal vehicles to ridesharing taxicabs, more passenger matches would be found and it would be possible that the number of total vehicles on the road (personal automobiles and taxicabs) would decrease as the passenger to rideshare departure ratio is greater than one to one. Further in the future as self-driving cars gain the trust of society and become a reality on the streets of the nation, the subsequent implementation of autonomous taxis would again increase the efficiency and practicality of a ridesharing system.

Developing an effective ridesharing analysis across the United States, which effectively argues that more advanced transportation systems can be realized on the existing web of roads, is a major undertaking for which this thesis provides a piece of compelling proof grounded in the real world. Specifically, this thesis focused on data from a subset transportation system of a subset region of
America consisting of 170 million trips. By applying ridesharing concepts to this 2013 New York City Taxi & Limousine Commission data set, this thesis pointed out flaws in the current system and identified characteristics of a ridesharing service that would significantly reduce the amount of vehicle miles driven by the city’s taxicabs. As evident from the previous chapters, numerous and varied future analyses will add to the knowledge gained from this thesis for a complete recommendation when building an effective ridesharing system.

The knowledge gained from the thesis helps serve as another piece to the puzzle of academic research. Continued and more developed work would be valuable to allow for an official recommendation of such ridesharing concepts to the NYC TLC as the organization plans for how to combat competitors such as Uber. Additionally, an actual adoption and implementation of such a recommendation by the TLC would result in an indescribable amount of press and exposure to the world; the success of which would serve as a major stepping-stone for ridesharing to be adopted in other regions. The research in this thesis gleans initial ridesharing trends and performances that are sufficient proof that ridesharing has significant potential in New York City and by extrapolation on a broader, even national, scale.
Appendix A

Additional Figures

Figure A.0.1: Self-Driving Car Advertisement from 20th Century

Figure A.0.2: Average Number of Taxis Driving on the Road by Time of Day [10]
Figure A.0.3: Pixelization of Five Boroughs of Manhattan

(a) Entire Geographic Boundary

(b) Example of Downtown Manhattan

(c) Example of Midtown Manhattan
<table>
<thead>
<tr>
<th>Date</th>
<th>Data Source</th>
<th>Data Type</th>
<th>Data Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01/01/2016</td>
<td>NYC TLC</td>
<td>Trip Data</td>
<td>Modified 2013 data</td>
</tr>
</tbody>
</table>

**Figure A.0.5:** Sampling of Modified 2013 NYC TLC Trip Data.

**Table:**

<table>
<thead>
<tr>
<th>Date</th>
<th>Data Source</th>
<th>Data Type</th>
<th>Data Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01/01/2016</td>
<td>NYC TLC</td>
<td>Trip Data</td>
<td>Modified 2013 data</td>
</tr>
</tbody>
</table>

**Figure A.0.5:** Sampling of Modified 2013 NYC TLC Trip Data.
Figure A.0.6: Per-Minute Trip Demand during 2013 by Week
(Zoomed-In to Capture the Bronx and Staten Island Curves of Figure 3.1.1)

Average Weekly Pickup Demand for all 2013 Trips
- Bronx
- Queens
- Brooklyn
- Manhattan
- Staten Island
Figure A.0.7: Daily Weather Plots (Monday – Thursday)

(a) Monday

(b) Tuesday

(c) Wednesday

(d) Thursday
Figure A.0.8: Daily Weather Plots (Friday – Sunday)

(a) Friday  

(b) Saturday  

(c) Sunday

Figure A.0.9: Per-Medallion Trip Distribution

(a) Density Plot  

(b) Scatter Plot
Figure A.0.10: Cumulative Distribution Function of Taxicab Pick-ups (for All Pixels)
Appendix B

Technical Documentation of Research

The following sections are by no means inclusive of all of the code written for this thesis. In fact, they only consist of a fraction of all of the finalized code used in this thesis, let alone the code that became outdated as the research evolved. Outside of Python, the main languages and programs used for different stages in the analysis consist of: R, ArcGIS ArcMap, PostgreSQL, Google BigQuery, ArcGIS ArcScene, and Microsoft Excel.

This chapter is split into different sections that include the major Python scripts used to run each section of work as the thesis developed. It is again imperative to note that the inclusion of such work does not make up the full amount of code required to create this thesis; therefore attempting to re-run, or continue, the analysis in this thesis is futile with these files alone. As such work would be encouraged, please contact me directly and a transfer of all of files can be arranged.

B.1 Creation of Unique IDs

In order to efficiently keep track of individual taxicabs and drivers belonging to each record of data, a unique identifier (belonging to a sequence of integers) was created and used to compress the size of the data while keeping the same amount of information.

The following code iterates through all the separate files at once and creates unique identifiers for both medallion and hack_license fields within the dataset. In order to condense the amount of attached code to this thesis, additional functions from similar scripts were saved to this single python
file. The executable script, however, was not updated. Meaning that one must make the necessary
changes to the executable script (replacing function names) to run the other pieces.

```python
# Helper Functions

def assignIDs(lst):
    ""
    # Assign medallion IDs based on unique medallions
    ""
    newlist = []
    i = 0
    for item in lst:
        i += 1
        newlist.append([str(i), item])
    return newlist

def writeListToCSV(filepath, list, filename, boolean, header):
    ""
    # Write unique medallion IDs to CSV
    ""
    os.chdir(filepath)
    list_out = open(filename, 'w')
    writer = csv.writer(list_out, lineterminator = '\n')
    if boolean:
        writer.writerow(header)
    writer.writerows(list)
    list_out.close()

# Functions

def unique_hackLIST(filepath, filenames):
    ""
    # Find all unique hack licenses in a list
    ""
    hack_list = []
    for file in filenames:
        # Monitor program while running
        i = 0
        print(f'Current filename: {file} (\n')
        f = open(file, 'r')
        for line in f:
            l = line.split(' ', 1)
            create a vector of each of the elements in row
            if l[1] in hack_list:
                pass
            else:
                hack_list.append(l[1])
        if i == 250000 or (i == 500000 or (i == 750000):
            return hack_list

def add_unique_hackLIST(hack_list, data_filepath, data_filenames):
    ""
    # Add more unique hack licenses
    ""
    os.chdir(data_filepath)
    for data_file in data_filenames:
        ""
        # Monitor program while running
        ""
        i = 0
        print(f'Current filename: {data_file} (\n')"
```python
data_file = open(data_file, "rU")
for line in data_file:
    i = i + 1
    line = line.split(\",") # NEED TO IGNORE THE \n !!!!!!
    if line[1] is "":
        pass
    elif line[1] in hack_list:
        pass
else:
    hack_list.append(line[1])
if (i == 25000) or (i == 50000) or (i == 75000):
    print(i)
data_file.close()
return hack_list

def add_unique_hack_id_fromfile(data_filepath, data_filenames, hack_id_filename, hack_id_path):
    """ Impoert an existing list from a csv file and add more unique hack_license from trip files
    :param data_filepath: path to get to folder on Hard Drive with more trip files to read
    :param data_filenames: vector of strings, each of which is a filename for a trip file
    :param hack_id_filename: name of the csv file with the existing list of hack_id / hack_license
    :param hack_id_path: path to get to location of existing list of hack_id / hack_license
    :return: a list of every unique hack_license
    """
    os.chdir(hack_id_path)
    hack_list = ""
    f = open(hack_id_filename, "rU")
    for line in f:
        line = line.split(\",") # CAN WE IGNORE \n HERE??
    hack_list.append(line[1])
    f.close()
    """ Read in multiple trip files and add to the list """
    os.chdir(data_filepath)
    for data_file in data_filenames:
        """ To monitor program while running """
        i = 0
        data_file = open(data_file, "rU")
        for line in data_file:
            i = i + 1
            line = line.split(\",") # NEED TO IGNORE \n !!!!!!
            if line[1] is "":
                pass
            elif line[1] in hack_list:
                pass
else:
                hack_list.append(line[1])
        if (i == 25000) or (i == 50000) or (i == 75000):
            print(i)
data_file.close()
return hack_list

def unique_medallion_LIST(filepath, filenames):
    """ Read multiple trip csv files and return all of the unique medallions in a list
    :param filepath: path to get to folder on Hard Drive
    :param filenames: vector of strings, each of which is a filename
    :param tryNumber: a string to be placed in front of the medallionID numbers
    :return: a list of every unique medallion
    """
    os.chdir(filepath)
    medallion_list = []
    for file in filenames:
        """ To monitor program while running """
        i = 0
        file = open(file, "rU")
        for line in file:
            i = i + 1
            line = line.split(\",") # CREATE A VECTOR OF EACH OF THE ELEMENTS IN ROW
            if line[0] is "": # CATCH THE BLANK SPACE
                pass
            elif line[0] in medallion_list:
                pass
else:
                medallion_list.append(line[0])
        if (i == 25000) or (i == 50000) or (i == 75000):
            print(i)
return medallion_list

def add_unique_medallion_LIST(medallion_list, data_filepath, data_filenames):
    """ Start with an existing python list and add more unique medallions from trip files
    :param medallion_list: existing list of unique medallions
    :param data_filepath: path to get to folder on Hard Drive
    :param data_filenames: vector of strings, each of which is a filename
    :param tryNumber: a string to be placed in front of the medallionID numbers
    :return: a list of every unique medallion
    """
    os.chdir(data_filepath)
    for data_file in data_filenames:
        """ To monitor program while running """
        if (i == 25000) or (i == 50000) or (i == 75000):
            print(i)
data_file.close()
return medallion_list
```
i = 0
print("Current filename: ", data_file)

# Import an existing list from a csv file and add more unique medallions from trip files
for line in data_f:
    i += 1
    l = line.split(",")
    if len(l) > 3:
        pass
    elif l[0] in medallion_list:
        pass
    else:
        medallion_list.append(l[0])
    if (i == 250000) or (i == 500000) or (i == 750000):
        print(i)
data_f.close()
return medallion_list

def add_unique_medallions_LIST_fromfile(data_filepath, data_filenames, medallion_id_filename, medallion_id_path):
    *** First read in the current medallion_id assignments ***
    os.chdir(medallion_id_path)
    medallion_list = []
    csvfile = open(medallion_id_filename, "rU")
    for line in csvfile:
        l = line.split("
            ")
    csvfile.close()
    *** Read in MULTIPLE trip files and add to the list ***
    os.chdir(data_filepath)
    for data_file in data_filenames:
        csvfile = open(data_file, "rU")
        data_f = csvfile
        for line in data_f:
            i += 1
            l = line.split("
            ")
            if len(l) > 3:
                pass
            elif l[0] in medallion_list:
                pass
            else:
                medallion_list.append(l[0])
            if (i == 250000) or (i == 500000) or (i == 750000):
                print(i)
data_f.close()
return medallion_list

M_PATH = "/Volumes/SwobodaThesis/tripData2013/"
PART_1 = ["trip_data_1"]
FILENAME_1 = ["trip_data_1/part_a.csv", "trip_data_1/part_b.csv", "trip_data_1/part_c.csv", "trip_data_1/part_d.csv", "trip_data_1/part_e.csv", "trip_data_1/part_f.csv", "trip_data_1/part_g.csv", "trip_data_1/part_h.csv", "trip_data_1/part_i.csv", "trip_data_1/part_j.csv", "trip_data_1/part_k.csv", "trip_data_1/part_l.csv", "trip_data_1/part_m.csv", "trip_data_1/part_n.csv"]
d1 = unique_medallions_LIST(M_PATH + PART_1, FILENAME_1)
id = assign_IDs(d1)
csv_filename_1 = "medallionIds_tripdata1.csv"
write_list_to_csv(M_PATH + PART_1, id, csv_filename_1, False, **)
part_1_time = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
pStart = time.clock()
pEnd = time.time()
write_list_to_csv(MPATH + PART2, ids, csv_filename2, False, **)  
partelapsedtime = time.clock()  
print('PART2 took + str(partelapsedtime) + " seconds to run")

PART2 = "trip_data3"  
FILENAMES3 = ["trip_data3_part_01.csv", "trip_data3_part_02.csv", "trip_data3_part_03.csv", "trip_data3_part_04.csv", "trip_data3_part_05.csv", "trip_data3_part_06.csv", "trip_data3_part_07.csv", "trip_data3_part_08.csv", "trip_data3_part_09.csv", "trip_data3_part_10.csv"]

d3 = add_unique_medallion_LIST(d2, MPATH + PART3, FILENAMES3)  
ids = assign_IDs(d3)  
csv_filename3 = "medallionIds_tripdata3.csv"  
write_list_to_csv(MPATH + PART3, ids, csv_filename3, False, **)  
partelapsedtime = time.clock()  
print('PART3 took + str(partelapsedtime) + " seconds to run")

PART2 = "trip_data4"  
FILENAMES4 = ["trip_data4_part_01.csv", "trip_data4_part_02.csv", "trip_data4_part_03.csv", "trip_data4_part_04.csv", "trip_data4_part_05.csv", "trip_data4_part_06.csv", "trip_data4_part_07.csv", "trip_data4_part_08.csv", "trip_data4_part_09.csv", "trip_data4_part_10.csv"]

d4 = add_unique_medallion_LIST(d3, MPATH + PART4, FILENAMES4)  
ids = assign_IDs(d4)  
csv_filename4 = "medallionIds_tripdata4.csv"  
write_list_to_csv(MPATH + PART4, ids, csv_filename4, False, **)  
partelapsedtime = time.clock()  
print('PART4 took + str(partelapsedtime) + " seconds to run")

PART2 = "trip_data5"  
FILENAMES5 = ["trip_data5_part_01.csv", "trip_data5_part_02.csv", "trip_data5_part_03.csv", "trip_data5_part_04.csv", "trip_data5_part_05.csv", "trip_data5_part_06.csv", "trip_data5_part_07.csv", "trip_data5_part_08.csv", "trip_data5_part_09.csv", "trip_data5_part_10.csv"]

d5 = add_unique_medallion_LIST(d4, MPATH + PART5, FILENAMES5)  
ids = assign_IDs(d5)  
csv_filename5 = "medallionIds_tripdata5.csv"  
write_list_to_csv(MPATH + PART5, ids, csv_filename5, False, **)  
partelapsedtime = time.clock()  
print('PART5 took + str(partelapsedtime) + " seconds to run")

PART2 = "trip_data6"  
FILENAMES6 = ["trip_data6_part_01.csv", "trip_data6_part_02.csv", "trip_data6_part_03.csv", "trip_data6_part_04.csv", "trip_data6_part_05.csv", "trip_data6_part_06.csv", "trip_data6_part_07.csv", "trip_data6_part_08.csv", "trip_data6_part_09.csv", "trip_data6_part_10.csv"]

d6 = add_unique_medallion_LIST(d5, MPATH + PART6, FILENAMES6)  
ids = assign_IDs(d6)  
csv_filename6 = "medallionIds_tripdata6.csv"  
write_list_to_csv(MPATH + PART6, ids, csv_filename6, False, **)  
partelapsedtime = time.clock()  
print('PART6 took + str(partelapsedtime) + " seconds to run")

PART2 = "trip_data7"  
FILENAMES7 = ["trip_data7_part_01.csv", "trip_data7_part_02.csv", "trip_data7_part_03.csv", "trip_data7_part_04.csv", "trip_data7_part_05.csv", "trip_data7_part_06.csv", "trip_data7_part_07.csv", "trip_data7_part_08.csv", "trip_data7_part_09.csv", "trip_data7_part_10.csv"]

d7 = add_unique_medallion_LIST(d6, MPATH + PART7, FILENAMES7)  
ids = assign_IDs(d7)  
csv_filename7 = "medallionIds_tripdata3.csv"  
write_list_to_csv(MPATH + PART7, ids, csv_filename7, False, **)  
partelapsedtime = time.clock()  
print('PART7 took + str(partelapsedtime) + " seconds to run")

PART2 = "trip_data8"  
FILENAMES8 = ["trip_data8_part_01.csv", "trip_data8_part_02.csv", "trip_data8_part_03.csv", "trip_data8_part_04.csv", "trip_data8_part_05.csv", "trip_data8_part_06.csv", "trip_data8_part_07.csv", "trip_data8_part_08.csv", "trip_data8_part_09.csv", "trip_data8_part_10.csv"]

d8 = add_unique_medallion_LIST(d7, MPATH + PART8, FILENAMES8)  
ids = assign_IDs(d8)  
csv_filename8 = "medallionIds_tripdata8.csv"
write_list_to_csv(MPATH + PART10, ids, csvfile_filename10, False, ")
part10_time = time. clock()
partelapsedtime = part10_time - part9_time
print("PART10 took " + str(partelapsedtime) + " seconds to run")

PART10 = "trip_data10"
FILINGAMES10 = ["trip_data10_part_a.csv", "trip_data10_part_b.csv", "trip_data10_part_c.csv", "trip_data10_part_d.csv", "trip_data10_part_e.csv", "trip_data10_part_f.csv", "trip_data10_part_g.csv", "trip_data10_part_h.csv", "trip_data10_part_i.csv", "trip_data10_part_j.csv", "trip_data10_part_k.csv", "trip_data10_part_l.csv", "trip_data10_part_m.csv", "trip_data10_part_n.csv"]
d10 = add_unique_medallion_LIST(d10, MPATH + PART10, FILINGAMES10)
csv_filename10 = "medallionIDs_tripdata10.csv"
write_list_to_csv(MPATH + PART10, ids, csvfile_filename10, False, ")
part10_time = time. clock()
partelapsedtime = part10_time - part9_time
print("PART10 took " + str(partelapsedtime) + " seconds to run")

PART11 = "trip_data11"
d11 = add_unique_medallion_LIST(d11, MPATH + PART11, FILINGAMES11)
csv_filename11 = "medallionIDs_tripdata11.csv"
write_list_to_csv(MPATH + PART11, ids, csvfile_filename11, False, ")
part11_time = time. clock()
partelapsedtime = part11_time - part10_time
print("PART11 took " + str(partelapsedtime) + " seconds to run")

PART12 = "trip_data12"
FILINGAMES12 = ["trip_data12_part_a.csv", "trip_data12_part_b.csv", "trip_data12_part_c.csv", "trip_data12_part_d.csv", "trip_data12_part_e.csv", "trip_data12_part_f.csv", "trip_data12_part_g.csv", "trip_data12_part_h.csv", "trip_data12_part_i.csv", "trip_data12_part_j.csv", "trip_data12_part_k.csv", "trip_data12_part_l.csv", "trip_data12_part_m.csv", "trip_data12_part_n.csv"]
d12 = add_unique_medallion_LIST(d12, MPATH + PART12, FILINGAMES12)
csv_filename12 = "medallionIDs_tripdata12.csv"
write_list_to_csv(MPATH + PART12, ids, csvfile_filename12, False, ")
part12_time = time. clock()
partelapsedtime = part12_time - part11_time
print("PART12 took " + str(partelapsedtime) + " seconds to run")

totalelapsedtime = time. clock()
print("This program took " + str(totalelapsedtime) + " seconds to run")

B.2 Data Manipulation and Analysis

This section contains Python code that adds the previously-computed unique IDs as new columns in the data, splits the date-time cell into a date cell and a time cell, flips order of the longitude-latitude
into lat-long, and does other tasks that are no longer immediately relevant to the analysis of this thesis. At the time that this code was run, the data had not been collected into one major comma-separated values (CSV) file, and thus the function filenamesLIST_given_month() was crucial to run a script on all the files at once.

```python
# Author: Swohda
import os
import csv
import time
import numpy as np
import trippy spatial

# HELPER FUNCTIONS


# filenamesLIST

def pathName_given_month(x):
    string = f"/Volumes/SwohdaThesis/tripData2013/" + f"trip_{x}".join(str(x))
    return string

FUNCTIONS

def filenamesLIST_given_month(x):
    FILENAMES_1 = ['trip_data_1_part_a.csv', 'trip_data_1_part_ab.csv', 'trip_data_1_part_ac.csv',
                    'trip_data_1_part_ad.csv', 'trip_data_1_part_ac.csv', 'trip_data_1_part_ag.csv',
                    'trip_data_1_part_ah.csv', 'trip_data_1_part_ai.csv', 'trip_data_1_part_m.csv',
                    'trip_data_1_part_am.csv', 'trip_data_1_part_an.csv', 'trip_data_1_part_ag.csv']

    FILENAMES_2 = ['trip_data_2_part_a.csv', 'trip_data_2_part_ab.csv', 'trip_data_2_part_ac.csv',
                    'trip_data_2_part_ad.csv', 'trip_data_2_part_ac.csv', 'trip_data_2_part_ag.csv',
                    'trip_data_2_part_ah.csv', 'trip_data_2_part_ai.csv', 'trip_data_2_part_m.csv',
                    'trip_data_2_part_am.csv', 'trip_data_2_part_an.csv', 'trip_data_2_part_ag.csv']

    FILENAMES_3 = ['trip_data_3_part_a.csv', 'trip_data_3_part_ab.csv', 'trip_data_3_part_ac.csv',
                    'trip_data_3_part_ad.csv', 'trip_data_3_part_ac.csv', 'trip_data_3_part_ag.csv',
                    'trip_data_3_part_ah.csv', 'trip_data_3_part_ai.csv', 'trip_data_3_part_m.csv',
                    'trip_data_3_part_am.csv', 'trip_data_3_part_an.csv', 'trip_data_3_part_ag.csv']

    FILENAMES_4 = ['trip_data_4_part_a.csv', 'trip_data_4_part_ab.csv', 'trip_data_4_part_ac.csv',
                    'trip_data_4_part_ad.csv', 'trip_data_4_part_ac.csv', 'trip_data_4_part_ag.csv',
                    'trip_data_4_part_ah.csv', 'trip_data_4_part_ai.csv', 'trip_data_4_part_m.csv',
                    'trip_data_4_part_am.csv', 'trip_data_4_part_an.csv', 'trip_data_4_part_ag.csv']

    FILENAMES_5 = ['trip_data_5_part_a.csv', 'trip_data_5_part_ab.csv', 'trip_data_5_part_ac.csv',
                    'trip_data_5_part_ad.csv', 'trip_data_5_part_ac.csv', 'trip_data_5_part_ag.csv',
                    'trip_data_5_part_ah.csv', 'trip_data_5_part_al.csv', 'trip_data_5_part_m.csv',
                    'trip_data_5_part_am.csv', 'trip_data_5_part_an.csv', 'trip_data_5_part_ag.csv']

    FILENAMES_6 = ['trip_data_6_part_a.csv', 'trip_data_6_part_ab.csv', 'trip_data_6_part_ac.csv',
                    'trip_data_6_part_ad.csv', 'trip_data_6_part_ac.csv', 'trip_data_6_part_ag.csv',
                    'trip_data_6_part_ah.csv', 'trip_data_6_part_al.csv', 'trip_data_6_part_m.csv',
                    'trip_data_6_part_am.csv', 'trip_data_6_part_an.csv', 'trip_data_6_part_ag.csv']

    FILENAMES_7 = ['trip_data_7_part_a.csv', 'trip_data_7_part_ab.csv', 'trip_data_7_part_ac.csv',
                    'trip_data_7_part_ad.csv', 'trip_data_7_part_ac.csv', 'trip_data_7_part_ag.csv',
                    'trip_data_7_part_ah.csv', 'trip_data_7_part_al.csv', 'trip_data_7_part_m.csv',
                    'trip_data_7_part_am.csv', 'trip_data_7_part_an.csv', 'trip_data_7_part_ag.csv']

    FILENAMES_8 = ['trip_data_8_part_a.csv', 'trip_data_8_part_ab.csv', 'trip_data_8_part_ac.csv',
                    'trip_data_8_part_ad.csv', 'trip_data_8_part_ac.csv', 'trip_data_8_part_ag.csv',
                    'trip_data_8_part_ah.csv', 'trip_data_8_part_al.csv', 'trip_data_8_part_m.csv',
                    'trip_data_8_part_am.csv', 'trip_data_8_part_an.csv', 'trip_data_8_part_ag.csv']

    FILENAMES_9 = ['trip_data_9_part_a.csv', 'trip_data_9_part_ab.csv', 'trip_data_9_part_ac.csv',
                    'trip_data_9_part_ad.csv', 'trip_data_9_part_ac.csv', 'trip_data_9_part_ag.csv',
                    'trip_data_9_part_ah.csv', 'trip_data_9_part_al.csv', 'trip_data_9_part_m.csv',
                    'trip_data_9_part_am.csv', 'trip_data_9_part_an.csv', 'trip_data_9_part_ag.csv']

    FILENAMES_10 = ['trip_data_10_part_a.csv', 'trip_data_10_part_ab.csv', 'trip_data_10_part_ac.csv',
                     'trip_data_10_part_ad.csv', 'trip_data_10_part_ac.csv', 'trip_data_10_part_ag.csv',
                     'trip_data_10_part_ah.csv', 'trip_data_10_part_al.csv', 'trip_data_10_part_m.csv',
                     'trip_data_10_part_am.csv', 'trip_data_10_part_an.csv', 'trip_data_10_part_ag.csv']

    FILENAMES_11 = ['trip_data_11_part_a.csv', 'trip_data_11_part_ab.csv', 'trip_data_11_part_ac.csv',
                     'trip_data_11_part_ad.csv', 'trip_data_11_part_ac.csv', 'trip_data_11_part_ag.csv',
                     'trip_data_11_part_ah.csv', 'trip_data_11_part_al.csv', 'trip_data_11_part_m.csv',
                     'trip_data_11_part_am.csv', 'trip_data_11_part_an.csv', 'trip_data_11_part_ag.csv']

    FILENAMES_12 = ['trip_data_12_part_a.csv', 'trip_data_12_part_ab.csv', 'trip_data_12_part_ac.csv',
                     'trip_data_12_part_ad.csv', 'trip_data_12_part_ac.csv', 'trip_data_12_part_ag.csv',
                     'trip_data_12_part_ah.csv', 'trip_data_12_part_al.csv', 'trip_data_12_part_m.csv',
                     'trip_data_12_part_am.csv', 'trip_data_12_part_an.csv', 'trip_data_12_part_ag.csv']

    if x == 1:
        return FILENAMES_1
    if x == 2:
        return FILENAMES_2
    if x == 3:
        return FILENAMES_3
    if x == 4:
        return FILENAMES_4
    if x == 5:
        return FILENAMES_5
    if x == 6:
        return FILENAMES_6
    if x == 8:
        return FILENAMES_7
```

98
if x == 9:
    return FILENAMES_9
if x == 10:
    return FILENAMES_10
if x == 11:
    return FILENAMES_11
if x == 12:
    return FILENAMES_12

# Deals with the issue that some cells may be empty
def convertToFloat(x):
    if (x is None or x is ''):
        return float(0)
    else:
        return float(x)

# Calculates the nearest distance using standard distance formula
def distance(lat1, long1, lat2, long2):
    x1 = (lat1 - lat2) ** 2
    x2 = (long1 - long2) ** 2
    return ((x1 + x2) ** 0.5)

# Creates a dictionary with a key (hack_license) and a value (hackID)
def create_hackID_dict(hackIDFILENAME, hackIDPATH):
    # Load the hackID assignments
    os.chdir(hackIDPATH)
    hackIDdict = open(hackIDFILENAME, "rU")
    # Create a dictionary to return
    hackIDdict = dict()
    for hack_line in hackIDdict:
        l = hack_line.split(' ')
        [l[1] = l[1].rstrip('\n')]
        hackIDdict[l[1]] = l[0]
    hackIDdict.close()
    return hackIDdict

# Creates a dictionary with a key (medallion) and a value (medallionID)
def create_medallionID_dict(medallionFILENAME, medallionIDPATH):
    # Load the hackID assignments
    os.chdir(medallionIDPATH)
    medallionIDs = open(medallionFILENAME, "rU")
    # Create a dictionary to return
    medallionIDs = dict()
    for medallion_line in medallionIDs:
        l = medallion_line.split(' ')
        [l[1] = l[1].rstrip('\n')]
        medallionIDs[l[1]] = l[0]
    medallionIDs.close()
    return medallionIDs

# Reads in the intersectionIDs and spits out a list of (lat,long) points
def convert_intersectionFile_to_pointlist(intersectionFILENAME, intersectionPATH):
    os.chdir(intersectionPATH)
    intersectFile = open(intersectionFILENAME, "rU")
    lat = []
    for intersect_line in intersectFile:
        l = intersect_line.split(' ')
        [l[1] = intersect.line.split(' ')]
        l[lat] = convertToFloat(l[1][1])
        l[long] = convertToFloat(l[1][2])
        list.append([l[lat], l[long]])
    list.append([l[lat], l[long]])
    intersectFile.close()
    return list

# KDTree search
def do_kdtree(mytree, point):
    dist, indexes = mytree.query(point)
    return dist, indexes

# General search function inputting (lat,long) of point and list of intersection (lat,long)'
# ( uses the KDTree)
def findNearest2DPoint(pointOfInterest, longOfInterest, mytree):
    # Given a point and find the index and distance away of the nearest point in the list
    # of all comparable points
    point = [pointOfInterest, longOfInterest]
    result, dist = mytree.query(point)
    if dist > 0:
        result, index = result + 1
        return result, index, result.dist

# FUNCTIONS

def masterfile_intersection(outputFilename, headerBoolean, outputPath, hackIDdict, medallionIDdict, FILENAME, PATH, KDtree, filestarttime):
    # Adds the unique hackID
    # Splits the date/time cells into date cells and time cells
    # Add the pickup intersection ID, (dist between actual pickup and assigned intersection)
    # Add the dropoff intersection ID, (dist between actual pickup and assigned intersection)
    # Finds the longitude, latitude order in the masterfile to latitude, longitude
    # return
    # File/function start time

99
154  print(" Global Start Time of File: " + str(filestartTime) + " seconds")
155 # Load the trip file
156 os.chdir(FILEPATH)
157 file = open(FILENAME, "r")
158
159 # Create the output file
160 os.chdir(outputPATH)
161 output = open(outputFilename, "w")
162 wr = csv.writer(output, lineterminator="\n")
163
164 # Write header if 'headerBoolean' is True
165 if headerBoolean:
166     wr.header = csv.writer(output, lineterminator="\n")
168     wr.writerow(header)
169
170 i = 0
171
172 # Iterate through each line in the file, change it, and then print it into the new file
173 for line in file:
174     i += 1
175     if ((i % 200000) == 0):
176         print(" Elapsed time since start of file: " + str((time.clock() - filestartTime) / 60) + " minutes")
177     print(" Have iterated through : " + str(i) + " lines")
178     line = line.split("")
179     if store_and_fwd_flag is empty, replace with N/A
180     if ("[1]" == ""):
181         "[1]" = "N/A"
182     # Create the new list to be outputted
183     newLine = []
184     "FIND THE BACKID"
185     # Create assignment backID (if we don’t find and assign a hackID, we’ll know because it’ll stay as -1)
186     assignment_backID = -1
187     "FIND THE MEDALLIONID"
188     assignment_medallionID = -1
189     "FIND THE DATE-TIME"
190     dt1 = [15, split("")]
191     dt2 = [16, split("")]
192     "FIND THE INTERSECTION IDs AND DISTANCES"
193     # Deal with the last cell in row problem (it has a "\n" attached to it)
194     "[13]" = "[13]".rstrip("\n")
195     # Create lat/long variables, and placeholder variables for closest-intersection area
196     pickup_lat = convertToFloat("[14]")
197     pickup_long = convertToFloat("[15]")
198     dropoff_lat = convertToFloat("[16]")
199     dropoff_long = convertToFloat("[17]")
200     "THE CLOSEST POINT, WITH ASSOCIATED DISTANCE AND INDEX"
201     pickup_intersectID, pickup_dist = findNearest2DPoint(pickup_lat, pickup_long, KD_tree)
202     dropoff_intersectID, dropoff_dist = findNearest2DPoint(dropoff_lat, dropoff_long, KD_tree)
203     newLine.append([0]) # add medallion
204     newLine.append([1]) # add hackID
205     newLine.append([2]) # add vendorID
206     newLine.append([3]) # add rateCode
207     newLine.append([4]) # add store_and_fwd_flag
208     newLine.append([5]) # add pickup_datetime
209     newLine.append([6]) # add pickup_locationID
210     newLine.append([7]) # add passenger_count
211     newLine.append([8]) # add trip_distance
212     newLine.append([9]) # add trip_time
213     newLine.append([10]) # add dist_between_pickup_and_dropoff
214     newLine.append([11]) # add medallionID
215     newLine.append([12]) # add dropoff_locationID
216     wr.writerow(newLine)
217     file. close()
218     output. close()
B.3 Data Dimension Reduction and Trip Indexing

The following code runs through a single CSV file that contains all 170MM taxicab trips. In a line-by-line process, the python script checks for and removes trips that violate rules described in Section 2.4 on page 18 and it assigns unique trip IDs to each trip. These unique identifiers are a best practice for any type of big data management, and they helped with further analyses in this thesis.

```python
medallionID_filepath = "Volumes/SwobodaThesis/tripData2013/producedFiles"
intersection_filepath = "Volumes/SwobodaThesis/tripData2013/producedFiles"

### Step 2
# Create dictionary by reading in the unique hackID file that was generated earlier
unique_hackID_dict = create_hackID_dict(hackID_filepath)

### Step 4
# Create list of all of the intersection points, where their index is ordered similar to their ID
intersection_point_list = convert_intersectionFile_to_point_list(intersection_filepath)

### Step 6
# Create the kd_tree of all intersection points
KD_tree = scipy.spatial.cKDTree(intersection_point_list)

### Step 7
# Run code for every folder worth of files, editing them and then outputting them to new folders
for i in int_list:
    print("Month "+str(i))
    out_PATH = "Volumes/SwobodaThesis/tripData2013/altered_05_tripdata" + str(i)
data_filename_LIST = filenamesLIST_given_month(i)
data_filePath = pathName_given_month(i)

    for data_filename in data_filename_LIST:
        startfile = time.clock()
        newfilename = "altered" + data_filename
        print("NOW BUILDING " + newfilename + ")")
        masterfile = alterFile(newfilename, False, out_PATH, unique_hackID_dict, unique_medallionID_dict, data_filename, data_filePath, KD_tree, time.clock())
        print("Time for " + newfilename + " = " + str((time.clock()-startfile)/60) + " minutes")
        print("\n")

# End the file timer
elapsedTime = time.clock() - start
print("Seconds to run program: " + str(elapsedTime))
print("Minutes to run program: " + str(elapsedTime / 60))
```

---

![Image](image_url)
def assignTripIds():
    """Searches through a csv file
    Removes vendor_id, store_end_flag, pickup_dist_bw_actualID, dropoff_dist_bw_actualID
    Makes sure that the trip falls within the geographical boundaries, if not, it does not include it
    Also makes sure that the trip hack_license is not "CFCD208495D65EF66EDFDFF98764DA"
    Aids tripIds to the trips that fit in the geographical boundaries
    barebonesTripInfo():
    Opens the tripID-containing tripdata files
    Extracts only the tripID and the lat-long of pickups and dropoffs (to reduce filesize for ArcMap)
    (this information can be joined back together with the tripID-containing tripdata files
    using python)
    """

    def assignTripIds(FILeNAME, FILEPATH, outputFILENAME, outputFILEPATH, headerBoolean, row_count_limit):
        """Note: If row_count_limit is negative, then the row_count_limit does not affect the search"
        # Find and Open File
        # ex. chdir(FILEPATH)
        file = open(FILENAME, "U")
        # Create the Output File
        # ex. chdir(outputFILEPATH)
        output = open(outputFILENAME, "a")
        wr = csv.writer(output, lineterminator="\n")
        # Write a header in the file
        if headerBoolean:
            wr.header = csv.writer(output, lineterminator="\n")
            wr.header.writerow(header)
        # "NOTE THAT I REMOVED THE FOLLOWING COLUMNS FROM DATA"
        # vendor_id col14
        # store_end_flag col16
        # pickup_datetime col17
        # dropoff_datetime col18
        # pickup_dist_bw_actualID col19
        # dropoff_dist_bw_actualID col21
        # "NOTE THAT I AM ALSO ADDING A trip_ID COLUMN TO THE DATA"
        # To track searched trips
        searchedTripsCount = 0
        # To track recorded trips
        recordedTripsCount = 0
        # To track count of cause for removed trips
        pickup_remove_count = 0
        dropoff_remove_count = 0
        bad_medallion_count = 0
        zero_mile_trip_count = 0
        # Iterate through each line in the file, then print it into the new file if it passes the tests
        for line in file:
            # Break out of this for loop if the row_count_limit is surpassed
            if row_count_limit > 0:
                if row_count_limit is negative, there is no limit
                if searchedTripsCount >= row_count_limit:
                    break
            # +1 to searched trip count
            searchedTripsCount += 1
            # Split the string into a list of strings, using comma delimiter
            l = line split("")
            # Find pickup/dropoff lat/longs
            p_lat = convertToFloat(l[18])
            p_long = convertToFloat(l[19])
            d_lat = convertToFloat(l[22])
            d_long = converterFloat(l[23])
        # Deal with last cell in each row having an "\n" after it
        d_long = convertToFloat(l[23])
        # Assign lat-long boundaries (from NYC Borough Shape File in ArcMap)
        lat_max = float(40.915098)
        lat_min = float(40.496552)
        long_max = float(-73.699693)
        long_min = float(-74.255386)
        # Create boolean checks
        pickup_violation = p_lat > lat_max or p_lat < lat_min or p_long > long_max or p_long < long_min
        dropoff_violation = d_lat > lat_max or d_lat < lat_min or d_long > long_max or d_long < long_min
        if pickup_violation:
            pickup_remove_count += 1
        if dropoff_violation:
            dropoff_remove_count += 1
# Medallion boolean
isBadMedallion = isBlacklistedMedallion(1[0])
if isBadMedallion:
    bad_medallion_count += 1

# Hack license boolean
isBadHack = isBlacklistedHackLicense(1[2])
if isBadHack:
    bad_hack_count += 1

# Trip Length boolean
tripDist = convertToFloat(1[15])
isNotZeroMileTrip = tripDist > 0
if not isNotZeroMileTrip:
    zero_mile_trip_count += 1

# If this trip passes the checks, add it and save it:
if isNotZeroMileTrip and not pickupViolation_bool and not dropoffViolation_bool and not isBadMedallion and not isBadHack:
    # Count number of recorded trips
    recordedTripsCount += 1
    if (recordedTripsCount % 100000 == 0):
        print("Recorded Trip Tally = " + str(recordedTripsCount))
    
    # Create the new list to be outputted
    new_ = []
    
    # Create new row
    tripID = str(recordedTripsCount)
    new_.append(tripID) # add trip_ID to the data
    new_.append(1[0])
    new_.append(1[2])
    new_.append(1[3])
    new_.append(1[4]) # removed vendor_id
    new_.append(1[5])
    new_.append(1[6]) # removed store_and_flag
    new_.append(1[7])
    new_.append(1[8])
    new_.append(1[9])
    new_.append(1[10])
    new_.append(1[12])
    new_.append(1[13])
    new_.append(1[14])
    new_.append(1[15])
    new_.append(1[16])
    new_.append(1[17]) # removed pickup_dist_btw_actualID
    new_.append(1[18]) # removed pickup_latitude
    new_.append(1[19]) # pickup_longitude
    new_.append(1[20])
    new_.append(1[21]) # removed dropoff_dist_btw_actualID
    new_.append(1[22]) # dropoff_latitude
    new_.append(1[23]) # dropoff_longitude
    
    # Save row
    wr.writerow(new_)

    # Schedule to log
    
    # Close the file
    file_ = open(FILENAME, "w")
    file_.write(outputFILENAME, "w")

    # Create the Output File
    output(FILENAME, "w")
    output = open(outputFILENAME, "w")
    wr = csv.writer(output, lineterminator="\n")
    
    # Write a header in the file
    header = ['tripID', 'pickup_latitude', 'pickup_longitude', 'dropoff_latitude', 'dropoff_longitude']
    header.writerow(header)
    
    # Counters
    row_count = 0
    
    # Split and save the subset of information
    for row in file_:
        row_count += 1
        r = row.split(" ")
        
        # Create the new list to be outputted
        new_ = []
        
        # Add to the new list, which comprises the row
        new_.append(r[0])
        new_.append(r[14])
        new_.append(r[15])
        new_.append(r[17])
B.4 Pixelization of All Trips

This next section contains a Python script that iterates through a CSV file containing all of the 2013 NYC TLC trip data. The final objective of this script is to assign pickup and drop-off pixel IDs to each trip, as well as assigning a unique trip ID and count the frequency of taxicab pickups and drop-offs at each pixel. There are three mid-sized functions and one major function in this script.

The three mid-sized functions are the following:

1. `createPixeltripCountDictionary()` is a function called to keep a live counter of the number of trip pickups and drop-offs as the CSV file is iterated through.
2. `createPixelInfoDictionary()` is a function that creates a dictionary containing additional information for each of the 84,970 pixels in the grid, including each pixel’s X- and Y-coordinates; and
3. `pixel_KDTree()` creates a 2-dimensional KD-tree of all of the lat-long centroids of each pixel.

The major function is `pixelize()`, which takes the following inputs: the trip data CSV file, the KD-tree, the Python dictionary data structure of pixel information, and the dictionaries to record counts of the number of pickups and drop-offs at pixel. This function iterates through the CSV file and for each trip, it finds the nearest pixel centroid for both the pickup and drop-off based on the lat-long information in the original data. Once it has assigned a pickup and drop-off pixel for each trip, it uses the dictionary to add additional information about the pixel (the X- and Y-coordinates),

```python
r[18] = r[18].strip() + "n"
new_row.append(r[18])
wr.writerow(new_row)
file.close()
output.close()
pixelize()()

executeCode()

### Step 1***

#### Create inputs for function

data_filepath = "Volumes/SwobodaThesis/tripData2013/altered_2_13_tripdata.csv"
data_filename = "altered_2_13_tripdata_ALL.csv"
output_filename = "all2013_tripdata_allFields_noReader.csv"

### Step 2***

#### Create trip ID file with all the column fields

assignTripIDs(data_filename, data_filepath, output_filename, False, -1)

### Step 3***

#### Strip all other information (when I was considering doing all Pixelization in ArcMAP)
data_filename2 = "Volumes/SwobodaThesis/tripData2013/altered_2_12_tripdata.csv"

#### Output file path

output_filepath = "Volumes/SwobodaThesis/tripData2013/altered_2_13_tripdata.csv"

#### Barreness Trip Info

barenessTripInfo(data_filename2, data_filepath2, output_filename2, output_filepath2, True)

#### End the file timer

elapsedTime = time.clock() - start
print("Seconds to run program: " + str(elapsedTime))
print("Minutes to run program: " + str(elapsedTime / 60))

#### Executable Code

```
```
and it adds a count to the record count dictionaries. Before the function moves to the next row of data (the next trip), it outputs this new information as a row in a new file, effectively re-building the data as it reads through the old data file. Ultimately, upon completion, the function also saves the record count dictionaries to the hard drive for later reference.

```python
import os
import csv
import time
import numpy as np
import scipy.spatial

# HELPER FUNCTIONS

def convertToFloat(x):
    return float(x)

def findNearest2DPoint(lat, long):
    mytree = scipy.spatial.KDTree(pointsList)
    result = mytree.query([lat, long])
    return result

# KDTree search

def do_kdtree(mytree, point):
    dist, indexes = mytree.query(point)
    return indexes

# General search function inputting (lat, long) of point and list of pixel centroid (lat, long)'

def findNearest2DPoint(lat, long):
    # Given a lat and long, find the index of the nearest point in the list
    # of all comparable points
    # Note: (uses the KDTree search function)
    point = [lat, long]
    result_index = do_kdtree(mytree, point)
    # Note that the index of the list starts at 0 and the pixel_ID list also starts at 0
    # so no need to add 1
    return result_index

# Function to create a dictionary containing all the extra pixel information

def createPixelInfoDictionary(pixelinfoFILENAME, pixelinfoFILEPATH):
    pixelFile = open(pixelFileFILENAME, 'rU')
    pixelInfoDict = dict()
    for line in pixelFile:
        l = line.split(',')
')
        pixelInfoDict[l[0]] = [l[3], l[4]]  # y, x
    return pixelInfoDict

# Function to create a dictionary that counts the number of pickups or dropoffs that occur
# in each pixel

def createPixelTripCountDictionary(pixelinfoFILENAME, pixelinfoFILEPATH):
    pixelFile = open(pixelFileFILENAME, 'rU')
    tripCountDict = dict()
    for line in pixelFile:
        l = line.split(',')
        tripCountDict[l[0]] = 0
    return tripCountDict

# Function to print a dictionary to a csv file

def saveDictionary(dict, outputFILENAME, outputFILEPATH, headerBoolean, type):
    output = open(outputFILENAME, 'a')
```

105
def pixelize(starttime, FILENAME, FILEPATH, outputFILENAME, outputFILEPATH, pixel_info_dict, pickup_tripCount_dict, dropoff_tripCount_dict):
    # Find and Open File
    os.chdir(FILEPATH)
    file = open(FILENAME, "rU")
    # Create the Output File
    output = open(outputFILENAME, "w")
    wr = csv.writer(output, lineterminator="\n")
    # Write a header in the file
    if headerBoolean:
        wr.header.writerow(header)
    # Counters
    row_counter = 0
    for line in file:
        # Output the program's progress
        row_counter += 1
        if (row_counter % 500000 == 0):
            print("Iterated through " + str(row_counter) + " rows")
        if (row_counter % 200000 == 0):
            elapsed = time.clock()
            diff = elapsed - starttime
            print("Program has run so far for: " + str(diff/60) + " minutes")
        # Split the string into a list of strings, using comma delimiter
        line = line.split(","
        # Create lat-long variables to put into kdTree search
        pickup_lat = convertToFloat(line[14])
        pickup_long = convertToFloat(line[15])
        dropoff_lat = convertToFloat(line[17])
        dropoff_long = convertToFloat(line[18])
        pickup_pixel = findNearest2DPoint(pickup_lat, pickup_long, kdTree)
        dropoff_pixel = findNearest2DPoint(dropoff_lat, dropoff_long, kdTree)
        # Pull related x and y pixel coords from dictionary
        pickup_pixelX = pixel_info_dict[str(pickup_pixel)]
        pickup_pixelY = pickup_pixel[0]
        dropoff_pixelX = dropoff_pixel[0]
        dropoff_pixelY = dropoff_pixel[1]
        # Add the count to the total for pickup and dropoff pixels
        i = pickup_tripCount_dict[str(pickup_pixelX)]
        i += 1
        pickup_tripCount_dict[str(pickup_pixelX)] = i
        j = dropoff_tripCount_dict[str(dropoff_pixelX)]
        j += 1
        dropoff_tripCount_dict[str(dropoff_pixelX)] = j
        # Create the new list to be outputted
        newline = []
        # Add information to the row
        **Dropped intersection ID information!!!**
new_.append([0])
new_.append([1])
new_.append([2])
new_.append([3])
new_.append([4])
new_.append([5])
new_.append([6])
new_.append([7])
new_.append([8])
new_.append([9])
new_.append([10])
new_.append([11])
new_.append([12])
new_.append([13])
new_.append([14])
new_.append([15])
new_.append([16])
new_.append([17])
new_.append([18])
new_.append([19])
new_.append([20])
new_.append([21])
new_.append([22])
new_.append([23])
new_.append([24])
new_.append([25])
new_.append([26])
new_.append([27])
new_.append([28])
new_.append([29])
new_.append([30])
new_.append([31])
new_.append([32])
new_.append([33])
new_.append([34])
new_.append([35])
new_.append([36])
new_.append([37])
new_.append([38])
new_.append([39])
new_.append([40])
new_.append([41])
new_.append([42])
new_.append([43])
new_.append([44])
new_.append([45])
new_.append([46])
new_.append([47])
new_.append([48])
new_.append([49])
new_.append([50])
new_.append([51])
new_.append([52])
new_.append([53])
new_.append([54])
new_.append([55])
new_.append([56])
new_.append([57])
new_.append([58])
new_.append([59])
new_.append([60])
new_.append([61])
new_.append([62])
new_.append([63])
new_.append([64])
new_.append([65])
new_.append([66])
new_.append([67])
new_.append([68])
new_.append([69])
new_.append([70])
new_.append([71])
new_.append([72])
new_.append([73])
new_.append([74])
new_.append([75])
new_.append([76])
new_.append([77])
new_.append([78])
new_.append([79])
new_.append([80])
new_.append([81])
new_.append([82])
new_.append([83])
new_.append([84])
new_.append([85])
new_.append([86])
new_.append([87])
new_.append([88])
new_.append([89])
new_.append([90])
new_.append([91])
new_.append([92])
new_.append([93])
new_.append([94])
new_.append([95])
new_.append([96])
new_.append([97])
new_.append([98])
new_.append([99])

end = time.clock()
print('Seconds to run program : ' + str(end - start))

filename = "\Volumes/SwohoThesis\tripData2013\altered_2-19_tripdata.csv"
output_filename = "\Volumes/SwohoThesis\tripData2013\altered_2-19_tripdataStats.csv"

# Export the dictionary to a csv file
dict_output_PATH = "\Volumes/SwohoThesis\tripData2013\altered_2-19_tripdataStats.csv"
dict_output_dictionary = dict(output_PATH = True, output_filename = output_filename)
dict_output_dictionary = saveDictionary(d, dict_output_dictionary, dict_output_PATH)

# End the file timer
print('Total time elapsed : ' + str(end - start))
B.5  \((CD = 1)\)

The first Python script attached to this section, B.5.1, handles two major tasks: the trip file preparation and the simulation of a ridesharing policy.

This trip file preparation part of the code was run only once, and not included in following simulation files as the results it produced were saved to an external hard drive. For this preparation portion, the following logic ensued. For every pixel in a specified list, the script opens up the entire trip data file and pulls all of the trips originating at the specific pixel. Once all of these pixel-specific trip files are loaded and saved, the Python script runs through each of them again, and sorts each file of trips in chronological order of pickup date and time; it then re-saves the file. These sorted trip files, specific to each pixel, are what were used for all of the different ridesharing policy simulations.

The simulation portion of this script first specifies the input and output filepaths and filenames. Then for each pixel, time window, and geographic boundary combination, the script computes the correlated ridesharing and the casual ridesharing analyses specific to the overarching policy — in this instance, \((CD = 1)\). For each of these combinations, the script saves the new departure file of trips and it writes a summary statistic line to an overall file containing aggregated information of all rideshares. Note that the \texttt{casualRidesharingADONew()} function included in this script is the main function for executing the \((CD = 1)\) logic and creating the correct departure files.

The second Python script in this section, B.5.2, represents the ridesharing analysis performed for \((CD = 1)\). It calculates the vehicle miles saved, and other statistics, in each departure combination of geography, time window, and pixel in NYC. It then re-saves the overall summary statistic file with this additional information appended to each row.

B.5.1  \((CD = 1)\) — Ridesharing Preparation and Simulation

```python
# Function to create a dictionary containing the extra pixel information
def createPixelInfoDictionary(ALL(pixelinfoFILENAME, pixelinfoFILEPATH):
    os.chdir(pixelinfoFILEPATH)
    file = open(pixelinfoFILENAME, "rU")
    # Columns in Pixel file: pixel_ID, pixel_lat, pixel_long, y_AJ_Pixel, x_AJ_Pixel
    pixelinfo_dict = {}
    for line in file:
        l = line.split("
"")
        pixelinfo_dict[int(l[0])] = [l[1], l[2], l[3], l[4]]  # lat, long, y, x
    file.close()
```
# Creates a dictionary of all of the dropoff Pixels in the trip data
d = {}

for key , value in tripFileDictionary.items():
    # There won't be a header in this tripFile dictionary
    if int(value[20]) not in d:
d[ int(value[20])] = pixelInfoDictionary[int(value[20])]

return d

# Checks if there are intersecting values between two lists
# Returns three things: boolean , a list comprising of all within intersection , the count of intersecting points
def intersectionAnalysis(list1 , list2):
    intersection_list = list(set(list1) & set(list2))
    intersection_boolean = False

    if intersection_list:
        intersection_boolean = True
        # if there's an intersection , change to true

    return (intersection_boolean , intersection_list , len(intersection_list))

# Pulls the pickup date from a row of data in a trip file
def extractPickupDate(line):
    line = line.split(',')

    if not line[6]:
        return "00:00:00-00-00"

    return line[6]

# Pulls the pickup time from a row of data in a trip file
def extractPickupTime(line):
    line = line.split(',')

    if not line[7]:
        return "00:00:00"

    return line[7]

# Creates a dictionary of all of the trip files
# key = tripID , value = row of data in list form
def create_tripFile_dict(FI L E N A M E, FILEPATH):
    os.chdir(FILEPATH)

    file = open(FI L E N A M E, "rU")
    d = {}

    header = []

    for line in file:
        line = line.split(',')

        if line[0] == "tripID":
            header.append(line[0])

        if line[0] == "tripID":

    file.close()

    return d , header

# Creates a list where each entry is a list that comprises of a trip
# the indices of this big list are in order the of time (the order of the file being read in)
def create_sortedTripFile_list(sorted_FILENAME, sorted_FILEPATH):
    os.chdir(sorted_FILEPATH)

    file = open(sorted_FILENAME, "rU")
    tripdata_list = []

    header = []

    for line in file:
        line = line.split(',')

        if line[0] == "tripID":
            header = line

        if line[0] == "tripID":
            printTime(startTime , endTime , partName) :
FUNCTIONS

# FUNCTIONS

def pull_pixelSpecificData(dataFILENAME, dataFILEPATH, outputFILENAME, outputFILEPATH, outputHeaderBoolean, pickup_pixel_id, dropoff_pixel_id):
    """Note: If pickup_pixel_id or dropoff_pixel_id is -1, then the function considers all possibilities"
    """Also Note: The data file shouldn’t have a header"

    functionStartTime = time.clock()
    # Find and Open File
    os.chdir(dataFILEPATH)
    file = open(dataFILENAME, "rt")
    # Create the Output File
    os.chdir(outputFILEPATH)
    output = open(outputFILENAME, "a")
    wr = csv.writer(output, lineterminator="\n")
    # Write a header in the file if specified
    if outputHeaderBoolean:
        wr.writerow(outputHeaderBoolean
        wr.writerow(header)
        # Counters
        tripSearchCount = 0
        tripMatchCount = 0

    if not outputHeaderBoolean:
        wr.writerow(outputHeaderBoolean
        wr.writerow(header)
        # Counters
        tripSearchCount = 0
        tripMatchCount = 0
    # If we care about only pickup id...
    if pickup_pixel_id > 0 and dropoff_pixel_id < 0:
        print("Searching for all trips with pixel pickup id = {} + str(pickup_pixel_id))
    # Search through all trips and save the ones that fit the correct specifications
    for line in file:
        tripSearchCount += 1
        # Print progress as it is running
        if tripSearchCount % 20000000 == 0:
            print("Currently at row {} + str(tripSearchCount))
            print("Number of matches = {} + str(tripMatchCount)))
    # Split the string into a list of strings, using comma delimiter
    l = line.split(
    l[22] = l[22].strip("\n")
# Pull trip pixel information
line[pixel_id] = int(line[15])

# Save to new file if the ids match
if line[pixel_id] == pickup_pixel_id:
    tripMatchCount += 1
    wr.writerow(1)

# If we only care about dropoff id...
if pickup_pixel_id < 0 and dropoff_pixel_id > 0:
    print("Searching for all trips with pixel dropoff id = \"+ str(dropoff_pixel_id) + ");

    # Search through all trips and save the ones that fit the correct specifications
    for line in file
        tripSearchCount += 1
        # Print progress as it is running
        if tripSearchCount % 20000000 == 0:
            print("...currently at row \" + str(tripSearchCount) + ");
            print("number of matches = \" + str(tripMatchCount) + ");

        # Split the string into a list of strings, using comma delimiter
        l = line.split(';')
        l[22] = l[22].rstrip('\"\n')

        # Pull trip pixel information
        line[dropoff_pixel_id] = int(l[15])

        # Save to new file if the ids match
        if line[dropoff_pixel_id] == dropoff_pixel_id:
            tripMatchCount += 1
            wr.writerow(1)

# If we care about both pickup id and dropoff id...
if pickup_pixel_id >= 0 and dropoff_pixel_id >= 0:
    print("Searching for all trips with pixel pickup id = \" + str(pickup_pixel_id) + ") and pixel dropoff id = \" + str(dropoff_pixel_id) + ");

    # Search through all trips and save the ones that fit the correct specifications
    for line in file
        tripSearchCount += 1
        # Print progress as it is running
        if tripSearchCount % 20000000 == 0:
            print("...currently at row \" + str(tripSearchCount) + ");
            print("number of matches = \" + str(tripMatchCount) + ");

        # Split the string into a list of strings, using comma delimiter
        l = line.split(';')
        l[22] = l[22].rstrip('\"\n')

        # Pull trip pixel information
        line[pickup_pixel_id] = int(l[15])
        line[dropoff_pixel_id] = int(l[20])

        # Save to new file if both of the set of ids are both matches
        if line[pickup_pixel_id] == pickup_pixel_id and line[dropoff_pixel_id] == dropoff_pixel_id:
            tripMatchCount += 1
            wr.writerow(1)

    file.close()
    output.close()
def superPixel(pixelID, pixelInfo_dictionary):
    # superPixel()
    # - takes in a pixel ID
    # - returns a list of all of the pixels that make up the "super pixel"
    # - a "super pixel" consists of the pixel itself, and the 8 pixels surrounding it
    # (1) (2) (3)
    #   (4) (5) (6)
    #     (7) (8)
    # - a "super pixel" is 0.3x0.3 mi in dimension
    superPixel_list = []
    if pixelID < 0:
        return superPixel_list
    return an empty list if we aren't calculating for this pixelID
    # pixelInfo_dictionary of the format:
    # key = pixelID
    # value = [pixelList, pixelLong, y_AJ_Pixel, x_AJ_Pixel]
    pixel_list = pixelInfo_dictionary[int(pixelID)]
    x = int(pixel_list[3])
    y = int(pixel_list[2])
    pixelId = int(pixelID)
    # convert to int, if it isn't already
    # Initialize all potential neighbor pixels
    pixelId_1 = -1
    pixelId_2 = -1
    pixelId_3 = -1
    pixelId_4 = -1
    pixelId_5 = -1
    pixelId_6 = -1
    pixelId_7 = -1
    pixelId_8 = -1
    # Calculate the immediate and diagonal neighbors
    topExists = 0 (0 <= y <= 289) # if pixelID is not in top row
    leftExists = 0 (0 <= x <= 292) # if pixelID is not in leftmost column
    rightExists = 0 (0 <= x <= 292) # if pixelID is not in rightmost column
    bottomExists = 0 (y <= 289) # if pixelID is not in bottom row
    if topExists:
        if not in top row
        pixelId_2 = pixelId + 293
        if leftExists:
            if not in leftmost column
            pixelId_4 = pixelId - 1
        if rightExists:
            if not in rightmost column
            pixelId_6 = pixelId + 1
        if bottomExists:
            if not in bottom row
            pixelId_7 = pixelId - 293
    # Calculate the diagonal neighbors
    # if topExists and leftExists:
def macroPixel(pixelID, pixelInfo_dictionary):
    # macroPixel():
    # - takes in a pixel ID
    # - returns a list of all of the pixels that make up the "super pixel"
    # - a "super pixel" consists of the pixel itself, and the 24 pixels surrounding it
    # - to calculate it, this function calculates the super pixel of each diagonal neighbor and
    # - then removes duplicates
    # (1) (2)
    # (3) (4)
    # - a "macro pixel" is 0.5x0.5 mi in dimension
    macroPixel_list = []
    # pixelInfo_dictionary of the format:
    # key = pixel_ID
    # value = [pixel_long, y, A1, A2, A3, A4]
    pixel_list = pixelInfo_dictionary[int(pixelID)]
    x = int(pixel_list[0])
    y = int(pixel_list[1])
    pixel_id = int(pixelID)  # convert to int, if it isn't already
    # Initialize all potential neighbor pixels
    pix_id_1 = x - 1
    pix_id_2 = x - 1
    pix_id_3 = x - 1
    pix_id_4 = x - 1
    # Calculate the immediate and diagonal neighbors
    topExists = (0 <= y < 289)  # if pixelID is not in top row
    if topExists:
        pix_id_1 = pix_id - 293
    if topExists and rightExists:
        pix_id_2 = pix_id + 293 + 1
    if bottomExists and leftExists:
        pix_id_3 = pix_id - 293 - 1
    if bottomExists and rightExists:
        pix_id_4 = pix_id + 293 + 1
    list1 = superPixel(pix_id_1, pixelInfo_dictionary)
    list2 = superPixel(pix_id_2, pixelInfo_dictionary)
    list3 = superPixel(pix_id_3, pixelInfo_dictionary)
    list4 = superPixel(pix_id_4, pixelInfo_dictionary)
    for item in (list1 + list2 + list3 + list4):
        if item not in macroPixel_list:
            macroPixel_list.append(item)
    # print(macroPixel_list)
    return macroPixel_list

def fallWithin_superPixel(pixel1, pixel2, pixelInfo_dictionary):
    pixel1 = int(pixel1)
    pixel2 = int(pixel2)
    if pixel1 == pixel2:  # if same pixel
        return True
    elif abs(pixel1 - pixel2) == 1:  # if horizontally next to each other
        return True
    elif abs(pixel1 - pixel2) == 293:  # if vertically next to each other
        return True
    else:
        superPix1 = superPixel(pixel1, pixelInfo_dictionary)
        superPix2 = superPixel(pixel2, pixelInfo_dictionary)
        # Determine any kind of intersection
        (intersectBoolean, intersectPixelList, intersectMag) = intersectionAnalysis(superPix1, superPix2)
        return intersectBoolean
def failWithinMacroPixel(pixel1, pixel2, pixelInfo_dictionary):
    pixel1 = int(pixel1)
    pixel2 = int(pixel2)
    if pixel1 == pixel2:
        return True
    elif abs(pixel1 - pixel2) == 1 or |
        abs(pixel1 - pixel2) == 2 or |
        abs(pixel1 - pixel2) == 3 or |
        abs(pixel1 - pixel2) == 4:
        return True
    else:
        return failWithinSuperPixel(pixel1, pixel2, pixelInfo_dictionary)

def geoGraphicallyAdjacent(pixel1, pixel2, pixelInfo_dictionary, superORmacro):
    if superORmacro == "macroPixel":
        return failWithinMacroPixel(pixel1, pixel2, pixelInfo_dictionary)
    else:
        return failWithinSuperPixel(pixel1, pixel2, pixelInfo_dictionary)

def corrRidesharing_AVO(dataFILENAME, dataFILEPATH):
    # corrRidesharing_AVO():
    # - input a trip data file
    # - iterates through, row by row, and sums the # of passengers and # of trips
    # - calculates the correlated ridesharing AVO
    # - prints information
    # - returns the correlated ridesharing AVO
    # Find and Open File
    os.chdir(dataFILEPATH)
    file = open(dataFILENAME, "rU")
    # Counters
    tripCount = 0
    passengerCount = 0
    for line in file:
        # Split the string into a list of strings, using comma delimiter
        l = line.split(\"\",)
        # Ignores the header line
        if l[0] == "tripID":
            continue
        tripCount += 1
        # Pull passenger count from line
        numPass = int(l[10])
        # Add count to the passenger total
        passengerCount += numPass
    file.close()
    # AVO from Correlated Ridesharing
    avo = float(pasengerCount) / float(tripCount)
    # print("Passenger Count = \n + str(passengerCount))
    # print("Trip Count = \n + str(tripCount))
    # print("AVO from Correlated Ridesharing = \n + str(avo))
    return avo, passengerCount, tripCount

def reduceTimeWindowDict(oldTimeWindow_dict, TimeWindowSeconds, tripFile_dict):
    newTimeWindow_dict = {}
    for tripID in oldTimeWindow_dict:
        list = value
        newTimeWindow_dict[tripID] = []
        for eachTripIDMatch in list:
            if eachTripIDMatch in the list of all possible matches
                for tripID_possible in list:
                    if the tripID fits in the time window, save it
                        if file in TimeWindow(extractPickupDatetime(tripFile_dict[tripID_possible]), extractPickupDatetime(tripFile_dict[tripID_possible]), TimeWindowSeconds), newTimeWindow_dict[tripID_possible].append(tripID_possible)
        return newTimeWindow_dict
def casualRidesharing_ADO_New(dict, timewindow, timeWindowSeconds, sortedTFList, pixelInfoDictionary, superORMacro, tripFile_dictionary):
    # Create the sorted trip file list by reading in the sorted trip file
    (sortedTFList, b) = create_sortedTripFileList(SORTEDdataFILENAME, SORTEDdataFILEPATH)
    i = 0
    for i in range(0, len(sortedTFList)):
        tripID = extractTripID(sortedTFList[i])
        timeWindow_dict[tripID] = []
        timeWindow_dict[tripID].append(tripID)
        j = i + 1
        if j > len(sortedTFList):
            break
        # Continue
        while j <= timeWindow_dict[tripID].append(extractTripID(sortedTFList[j]))
        print('
' + TimeWindow = ' + str(timeWindowSeconds) + ' seconds and Geo. Boundary = ' + superORMacro)
        print('
' + departures = {}
        passengers = {}
        recordedTripIDs = {}
        summaryStat_list = []
        tripshareCount = {}
        No. of Trips per Departure", "Original Trip Count", "Departure Count", "Departures w/ Tripshare", "% Reduction in Taxi Departures", "% of shared rides"
        To count the frequency of the different number of taxi trips making up tripshares
        tripshareCount = {}
        for j in range(1, 201):
            tripshareCount[j] = 0
            for j in range(0, 361):
                passengerFreqCount = {}
                for j in range(0, len(sortedTFList)):
                    passeng = extractPassengerCount(sortedTFList[j])
                    if trip in recordedTripIDs:
                        continue
                    else:
                        departures[departureIteration] = []
                        departures[departureIteration].append(trip)
                        passengers[departureIteration] = []
                        passengers[departureIteration].append(passeng)
                        recordedTripIDs[trip] = []
                        possibleTripsharesList = dict[timeWindow][trip]
                        for tripPossibility in possibleTripsharesList:
                            if trip != tripPossibility and tripPossibility not in recordedTripIDs:
                                extractDropoffPixel(tripFile_dictionary[tripPossibility]), pixelInfoDictionary, superORMacro):
                                    passengers[departureIteration].append(tripPossibility)
                                    recordedTripIDs[tripPossibility] = []
                                departureIteration += 1
                                tripCount += 1
                                total_passengers += 1
                                passengerAmount += 1
                                count.Unique_Rideshares = 0
                                print('""' + Total Number of Departures = 1 + str(len(departures))")
                                # Calculate and print tripshare count frequency
                                for key, value in departures.items():
                                    additionalTrips = len(value)
                                    tripCount += additionalTrips
                                    tripshareCount[additionalTrips] += 1

115
if len(value) > 1:
    count_Unique_Rideshares += 1

# Calculate passenger total frequency
additionalPassengers = sum(passengers[key])
passengerAmount += additionalPassengers
passengerFreqCount[additionalPassengers] += 1

# Recalculate the total number of passengers (sanity check)
for key, value in passengers.iteritems():
    total_passengers += sum(value)

print("Unique Rideshare Matches = " + str(count_Unique_Rideshares))
print("(count of departures containing more than one trip")

print("Tripshare Count Frequency:")
for key, value in tripshareFreqCount.iteritems():
    if value > 0:
        # of Departures with " + str(key) + ".trips = " + str(value))

print("")

print("Passenger Count Frequency:")
for key, value in passengerFreqCount.iteritems():
    if value > 0:
        print("")

oldRow = tripFileDict[tripID]
newRow = [departureID, dep_origin_trip, tripID, medallion, medallion_ID, hack_license, hack_license_code, pickup_date, pickup_time, dropoff_date, dropoff_time, passenger_count, trip_time_in_secs, trip_distance, pickup_latitude, pickup_longitude, pickup_pixel_ID, pickup_pixel, dropoff_latitude, dropoff_longitude, dropoff_pixel_ID, dropoff_pixel]
wr.writerow(newRow)

if len(value) > 1:
    count_Unique_Rideshares += 1

# Calculate passenger total frequency
additionalPassengers = sum(passengers[key])
passengerAmount += additionalPassengers
passengerFreqCount[additionalPassengers] += 1

# Recalculate the total number of passengers (sanity check)
for key, value in passengers.iteritems():
    total_passengers += sum(value)

print("Unique Rideshare Matches = " + str(count_Unique_Rideshares))
print("(count of departures containing more than one trip")

print("Tripshare Count Frequency:")
for key, value in tripshareFreqCount.iteritems():
    if value > 0:
        # of Departures with " + str(key) + ".trips = " + str(value))

print("")

print("Passenger Count Frequency:")
for key, value in passengerFreqCount.iteritems():
    if value > 0:
        print("")

oldRow = tripFileDict[tripID]
newRow = [departureID, dep_origin_trip, tripID, medallion, medallion_ID, hack_license, hack_license_code, pickup_date, pickup_time, dropoff_date, dropoff_time, passenger_count, trip_time_in_secs, trip_distance, pickup_latitude, pickup_longitude, pickup_pixel_ID, pickup_pixel, dropoff_latitude, dropoff_longitude, dropoff_pixel_ID, dropoff_pixel]
wr.writerow(newRow)
### Step 2****

#### Create PixelSpecific Trip Files****

1. **iteration** := 1
2. **pixelList_input** := []
3. for pixel in **pixelList_toBeSearched**:
   1. **i** := 1
   2. **pixelList_input**.append(pixel)
   3. if i > 10 * **iteration**:
      1. print("\n+ " + "**********")
      2. print("+ ** iteration **")
      3. print("+ " + "**********")
   4. **saveALL_pixelSpecificPickUData(allTRIPDATA, allTRIPDATA_filepath, pixelList_input, pixelSpecific_outputFILEPATH)**
   5. **pixelList_input** := []
   6. **iteration** += 1
7. if len(pixelList_input) <= 0:
   1. print("\n+ " + "**********")
   2. print("" + " Last iteration !!")
   3. print("+ " + "**********")
8. **saveALL_pixelSpecificPickUData(allTRIPDATA, allTRIPDATA_filepath, pixelList_input, pixelSpecific_outputFILEPATH)**

### Step 3****

#### Sort the PixelSpecific Trip Files by Pickup Date-time****

1. **step2_input_filepath** := "/Users/Swoboda/Dropbox/SeniorThesis/PyCharm/3-01_Ridesharing/
2. **step2_output_filepath** := "/Users/Swoboda/Dropbox/SeniorThesis/PyCharm/3-01_Ridesharing/SORTED_tripdata_pixelSpecific/"
3. **step2_input_filepath** := "/Volumes/Swohobs/SeniorThesis/tripData2013/altered_3-01_tripdata/tripdata_pixelSpecific/"
5. For each pixel in the list of files we want to sort:
   1. for pixel in **pixelList_toBeSearched**:
      1. **part1_TIMER** := time.clock()
      2. print("\n+ " + "**********")
      3. print("" + " Trip File Prep")
      4. printTime(part1_TIMER, time.clock(), "Trip File Prep")

### Step 4****

#### Create Inputs****

1. **part2_TIMER** := time.clock()
2. **create inputs for function**
3. **SORTED_data_filepath** := "/Users/Swoboda/Dropbox/SeniorThesis/PyCharm/3-01_Ridesharing/
4. **SORTED_data_filepath** := "/Volumes/Swohobs/SeniorThesis/tripData2013/altered_3-01_tripdata/"
5. **pixelInfo_Filename** := "pixel_528tREDGnoHeader.csv"
6. **pixelInfo_Filepath** := "/Users/Swoboda/Dropbox/SeniorThesis/PyCharm/3-01_Ridesharing/Departures/"
7. **departure_output_FPath** := "/Volumes/Swohobs/SeniorThesis/tripData2013/altered_3-01_tripdata/Departures/"
8. **departure_output_FPath_HARDDRIVE** := "/Volumes/Swohobs/SeniorThesis/tripData2013/altered_3-01_tripdata/Departures/"

### Step 2****

#### Run Correlated and Casual Ridesharing Analysis for every pixel, time window, and geographic boundary****

1. (While creating departure output files, this step also creates a "summaryStat" file for each permutation)
2. **create the PixelInfo Dictionary**
3. **step_pPixInfoTIMER** := time.clock()
4. **dict_pixInfo длиннаеa a create PIXELInfoDictionary_ALL(pixelInfo_Filename, pixelInfo_Filepath)**
5. **step_pPixInfoTIMER END** := time.clock()
6. printTime(step_pPixInfoTIMER, step_pPixInfoTIMER_END, "PixelInfoDict Creation")
7. **create a summary-statistic file to be filled out as the analysis is completed**
8. **summaryStatFILEPATH** := "SummaryStatistics_PennStationRedo.csv"
9. **summaryStatFilePathHARDDRIVE** := "/Volumes/Swohobs/SeniorThesis/PyCharm/3-01_Ridesharing/"
# Write the Header Row

timeWindowList = [300, 120, 60, 30]

# Create the time window list to be iterated through

# Create a list of all pixels in top50percent that haven't already been seen in transit hub list
# Iterate through all of the pixels and create the departure files for each ridesharing analysis
# Also add to the summary statistics file

# Conduct Casual Ridesharing Analysis

# Keep track of a previous timewindow dict, so that we can just chip away at the previous one with the
# bigger TimeWindow

# Conduct Casual Ridesharing Analysis for SuperPixel with a given time window

# Conduct Casual Ridesharing Analysis for MacroPixel with the same time window (to reduce runtime)
import os
c
# HELPER FUNCTIONS

def convertToFloat(x):
    if (x is None) or (x is ''):
        return float(0)
    else:
        return float(x)

def createPixelInfoDictionary_ALL(pixelinfoFILENAME, pixelinfoFILEPATH):
    os.chdir(pixelinfoFILEPATH)
    file = open(pixelinfoFILENAME, "rU")
    # Columns in Pixel file: pixel_ID, pixel_int, pixel_long, y, AJ,Pixel, x_AJ_Pixel
    pixelinfo_dict = {}
    for line in file:
        l = line.split('"
')
        pixelinfo_dict[int(l[0])] = [l[1], l[2], l[3], l[4]]
    file.close()
    return pixelinfo_dict

    d = {}
    for key, value in tripFileDictionary.iteritems():
        # There won't be a header in this tripFile dictionary
        if isinstance(value[20], list):
            for i in value[20]:
                d[i] = pixelInfoDictionary[int(i)]
    return d

def printTime(startTime, endTime, partName):
    duration = endTime - startTime
    if (duration < 1000):
        print('--- Seconds to run " + partName + ": " + str(duration))
    if (duration > 1000):
        print('--- Minutes to run " + partName + ": " + str(duration / 60))
    if (duration > 3600):
        print('--- Hours to run " + partName + ": " + str(duration / 3600))

# Checks if there are intersecting values between two lists
# Returns three things: boolean, a list comprising of all within intersection, the count of intersecting points
def intersectionAnalysis(list1, list2):
    intersection_list = list(set(list1)) & set(list2)
    intersection_boolean = False
    if intersection_list:
        intersection_boolean = True  # if there's an intersection, change to true
    return (intersection_boolean, intersection_list, len(intersection_list))

# Pulls the pickup date from a row of data in a trip file
def extractPickupDate(line):
    line = line.split('"')
```python
# Pulls the pickup time from a row of data in a trip file
def extractPickupTime(line):
    line = line.split('"
')
    if not line[7]:
        return "00:00:00"
    return line[7]

# Creates a dictionary of all of the trip files
# key = tripID, value = row of data in list form
def createTripFile_dict(FILENAME, FILEPATH):
    file = open(FILEPATH, "rU")
    d = {}
    header = []
    for line in file:
        line = line.split('"
')
        if line[0] == "tripID":
            header.append(line[0])
            header.extend([line[1], line[2], line[3], line[4], line[5], line[6], line[7], line[8], line[9], line[10], line[11], line[12], line[13], line[14], line[15], line[16], line[17], line[18], line[19], line[20], line[21], line[22]])

    if line[0] == "tripID":
        dt = datetime.strptime(line[0], "%Y-%m-%d %H:%M:%S")
        if dt in d:
            d[dt].append(line)
        else:
            d[dt] = [line]
    return d, header

# Creates a list where each entry is a list that comprises of a trip
# the indices of this big list are in order from the order of the file being read in
def createSortedTripFile_list(sortedFILENAME, sortedFILEPATH):
    file = open(sortedFILEPATH, "rU")
    tripdata_list = []
    header = []
    for line in file:
        line = line.split('"
')
        if line[0] == "tripID":
            header.append(line)
            tripdata_list.append(line)
    return tripdata_list, header

# Pull the pickup datetime as a datetime object from a row of data
def extractPickupDatetime(row):
    dt = datetime.strptime(row[6], "%Y-%m-%d %H:%M:%S")
    return dt

# Pull the tripID as an integer from a row of data
def extractTripID(row):
    return int(row[0])

# Pull the passenger count as an integer from a row of data
def extractPassengerCount(row):
    return int(row[10])

# Pull the passenger count as an integer from a row of data
def extractDropoffPixel(row):
    return int(row[20])

# Calculate the difference in seconds between two datetime objects
def secondsDifference(dateTime1, dateTime2):
    diff = dateTime2 - dateTime1
    return diff

# Returns true if the difference between the two datetime objects is smaller than the timeWindow
def fits(intimeWindow, dateTime1, dateTime2, timeWindow):
    diff = dateTime2 - dateTime1
    if diff.seconds <= timeWindow
```
```python
def summaryStatisticsFileToDict(dataFILENAME, dataFILEPATH):
    dict = {}
    # Load SummaryStatistics.csv
    file = open(dataFILENAME, "rt")
    for line in file:
        line = line.split( ",")
        if line[0] != "pixel_ID":
            line[10] = line[10].rstrip("\n")
        pixelID = int(line[0])
        timeWindow = int(line[1])
        geoBound = str(line[2])
        dict[(pixelID, timeWindow, geoBound)] = line
    file.close()
    return dict, header

def pullVehicleMilesSaved(pixelID, timeWindow, geoBound, departureFILEPATH):
    os.chdir(departureFILEPATH)
    if geoBound == "macroPixel":
        geo = "MP"
    else:
        geo = "SP"
    filename = str(pixelID) + "," + str(geo) + "," + str(timeWindow) + "," + str(departures.csv)
    print("Opening", filename, "\n")
    file = open(filename, "rU")
    dict_departureTripMiles = {}
    totalMiles = float(0)
    countLines = 0
    for line in file:
        line = line.split( ",")
        countLines += 1
        if line[0] == "departure_ID":
            continue
        if line[0] not in dict_departureTripMiles:
            dict_departureTripMiles[line[0]] = [float(line[14])]
        else:
            dict_departureTripMiles[line[0]].append(float(line[14]))
            totalMiles += float(line[14])
    file.close()
    totalMilesSaved = float(0)
    i = 0
    for key, value in dict_departureTripMiles.items():
        print("before", str(value))
        value.remove(max(value))
        print("after", str(value))
        totalMilesSaved += sum(value)
    return totalMilesSaved, totalMiles
```

The code above defines functions to load summary statistics from a CSV file, pull vehicle miles saved for a specific pixel, time window, and geographic boundary, and calculate total miles and miles saved. It also includes logic to remove the maximum value from a list of floats and print the updated list.
print( TransitHubs = pixelList + superPixel + [300, 120, 60, 30] + list = [300 , 120, 60, 30] + duplicates = [] + not [ pixel in pixelList + toBeSearched = [] ] + if pixel not in pixelList + top50percentANDHubs + pixelList last1947 part + pixelList toBeSearched. append ( pixel )) + else pixelList duplicates. append ( pixel ) + print(pixelList duplicates) + print(len(pixelListtoBeSearched)) + stats, dict, header = summaryStatisticsFileToDict(summaryStatFiles + INPUTPATH, summaryStatFileINPATH) + print("Number of Pixels being searched = \ + str(len(pixelListtoBeSearched))") + for geo in geoList: + for window in timeWindowList: + # Create the Output File + str = "\c:\\summaryStatisticsFiles\" + OUTPUTPATH + str + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + "\" + ""}
B.6 \((CD = 3, dCIR = 0.2)\)

The Python scripts in this section have many similar pieces to that of Section B.5. As mentioned earlier, this simulation code does not include any trip-file preparation. The major difference in the simulation code involves the `common_destination_x_ridesharing()` function and the automatic preparation functions. With regard to the ridesharing analysis Python script, the most significant difference from the analysis code in B.5.2 comes with the new `pullVehicleMilesSaved()` function.

### B.6.1 \((CD = 3, dCIR = 0.2)\) — Ridesharing Simulation

```python
# author: 'Swoboda'
import os
import csv
import time
import numpy as np
import scipy.spatial
import operator
import gc
import math
import datetime

# HELPER FUNCTIONS

# Function to convert a string to a float
def convertToFloat(x):
    if (x is None or x is ''):
        return float(0)
    else:
        return float(x)

# Function to create a dictionary containing the extra pixel information
def createPixelInfoDictionary_ALL(pixelinfoFILENAME, pixelinfoFILEPATH):
    file = open(pixelinfoFILENAME, 'rU')
    pixelinfo_dct = {}
    for line in file:
        l = line.split('\\n')
        lat = float(l[0])
        long = float(l[1])
    pixelinfo_dct[lat, long, y, AJ, Pixel] = {}
    file.close()
    return pixelinfo_dct

# Creates a dictionary of all of the dropoff Pixels in the trip data
    d = {}
    for key, value in tripFileDictionary.iteritems():
        if int(value[20]) not in d:
            d[int(value[20])] = pixelInfoDictionary[int(value[20])]
    return d

# Print Time
def printTime(startTime, endTime, partName):
    duration = endTime - startTime
    if (duration < 1000):
        print('-> Seconds to run %s,%s %s' % (partName, duration))
    if (duration > 100):
        print('-> Minutes to run %s,%s %s' % (partName, duration / 60))
    if (duration > 3600):
        print('-> Hours to run %s,%s %s' % (partName, duration / 3600))

# Checks if there are intersecting values between two lists
def intersectionAnalysis(list1, list2):
    intersection_list = list(set(list1) & set(list2))
    intersection_boolean = False
    if intersection_list:
        intersection_boolean = True
    if there's an intersection, change to true
```

124
The provided code snippet includes several functions for extracting data from trip files. Here's a breakdown of the key components:

- **extractPickupDateTime(row)**: This function takes a row of data and returns the pickup datetime. If the row is not valid, it returns None.

- **extractPickupDate(line)**: This function extracts the pickup date from a line of data. If the date is not present, it returns an empty string.

- **extractPickupCoord(line)**: This function extracts the pickup coordinates from a line of data. It returns a tuple of (pickupX, pickupY).

- **extractDropoffCoords(line)**: This function extracts the dropoff coordinates from a line of data. It returns a tuple of (dropoffX, dropoffY).

- **create_tripFile_dict(FILENAME, FILEPATH)**: This function creates a dictionary where the keys are file names and the values are the file paths.

- **create_sortedTripFile_list(sorted_FILENAME, sorted_FILEPATH)**: This function creates a sorted list of trip files.

The code also includes logic for reading lines from a file, splitting them, and processing the data according to the function names and descriptions provided.
```python
def extractPassengerCount(row):
    return int(row[10])

def extractDropoffPixel(row):
    return int(row[20])

def secondsDifference(datetime1, datetime2):
    diff = datetime2 - datetime1
    return diff.seconds

def isTimeWindowValid(datetime1, datetime2, timeWindow):
    diff = datetime2 - datetime1
    return diff.seconds <= timeWindow

### Distance formula

def distance(x1, y1, x2, y2):
    return sqrt((x1-x2)**2 + (y1-y2)**2)

def closestPointInList(listXY, point):
    return min(listXY, key=lambda x: distance(x, point))

### Minimum Distance of Coordinate Points

def minDistanceOfCoords(startX, startY, listXY):
    if len(listXY) == 1:
        dist = distance(startX, listXY[0][0], startY, listXY[0][1])
        return dist
    if len(listXY) == 2:
        path1 = distance(startX, listXY[0][0], listXY[1][0], listXY[1][1])
        path2 = distance(startX, listXY[0][0], listXY[0][1], listXY[1][1])
        return min(path1, path2)
    if len(listXY) == 3:
        path1 = distance(startX, listXY[0][0], listXY[1][0], listXY[1][1])
        path2 = distance(startX, listXY[0][0], listXY[1][0], listXY[2][1])
        path3 = distance(startX, listXY[0][0], listXY[2][0], listXY[2][1])
        return min(path1, path2, path3)
```

---

**FUNCTIONS**

---

```python
def summaryStatisticsFileToDict(dataFILENAME, dataFILEPATH):
    dict = {}
    # Load SummaryStatistics.csv
    os.chdir(dataFILEPATH)
    file = open(dataFILENAME, "rt")
```
```python
for line in file:
    line = line.split(" ")
    if line[0] == "pixelID":
        header = header + line
    elif line[0] == "timeWindow":
        timeWindow = int(line[1])
    elif line[0] == "geoBound":
        geoBoundary = str(line[1])
    elif line[0] == "dict":
        pixelID = int(line[0])
        timeWindow = int(line[1])
        geoBound = str(line[4])
        dict[(pixelID, timeWindow, geoBound)] = line

print("Opening", filename, "+", "")
file = open(filename, "+")

def pullVehicleMilesSaved(pixelID, timeWindow, geoBoundary, cd, circuity, departureFILEPATH):
    os.chdir(departureFILEPATH)
    filename = str(pixelID) + "(" + str(geoBoundary) + "+" + str(geoBoundary) + "CD" + str(cd) + "dCIR")
    print("Opening", filename, "+")
    file = open(filename, "+")
    # Create a dictionary to only record the trips that make up the skeleton of the departure, as we are
    # not just going to the center of the superPixel or megaPixel that is formed for the rideshare
    # destination
    dict, departure_skeleton = {}
    dict[dep_ID, originID] = {}
    # for every trip, I can calculate what the total coordinate distance would have been by just doing 1.2
    # coord distance of pickup and dropoff.
    totalOriginID, CoordManhatDist = float(0)
    totalRideshareCoordManhatDist = float(0)
    # Create "global" pickupCoord variables
    pickupX = None
    pickupY = None
    loggedPickupCoord = False
    
    for line in file:
        # Split line into valuable information pieces
        line = line.split(" ")
        dep_ID = int[0]
        dest_origin = int[1]
        dest_originator = int[2]
        
        # If the header, skip this line
        if dep_ID == "departure_ID":
            continue
        
        # Extract pickupcoords and dropoffcoords
        pickup = xCoords + str(yCoords)
        pickup = xCoords + yCoords
        x = xCoords
        y = yCoords
        x, y = extractPickupCoordOrDropoffFile(line)
        if not loggedPickupCoordBoolean:
            loggedPickupCoord = x, y
            loggedPickupCoordBoolean = True
        
        # Add this original trip's distance to the total count
        totalOriginID, CoordManhatDist = distance(pickupX, x, pickupY, y) + 1.2
        totalOriginID, CoordManhatDist = distance(pickupX, x, pickupY, y) + 1.2
        totalOriginID, CoordManhatDist = distance(pickupX, x, pickupY, y) + 1.2
        
        # If the trip is a new departure, add the dest originator's coords to the skeleton
        if dep_ID not in dict, departure_skeleton and int(dest_originator) == 1:
            temp_departure[dest_group] = [x, y]
            dict, departure_skeleton[dep_ID] = temp_departure
        
        # Error in your logic
        elif dep_ID not in dict, departure_skeleton and int(dest_originator) == 0:
            raise Exception("Your logic is flawed... This shouldn't happen")
        
        # If the trip belongs to an existing departure and is a destination originator, add it to the skeleton
        if dep_ID in dict, departure_skeleton and int(dest_originator) == 1:
            temp_departure[dest_group] = [x, y]
            dict, departure_skeleton[dep_ID] = temp_departure
        
        # Find the rideshare coord distance (the shortest path from the origin to the destinations)
        rideshare_coord, dist = minDistanceOfCoord(pickupX, pickupY, dest_coordList) + 1.2
```
445

for keyDepID, value_list in dict_keyDepID_value_list.items():
    print("originalCoord_manhattanDist = float(0)"
    if abs(value_list[0]) > 1:
        print("You're getting close to an error!!")
    return coord_milesSaved, originalCoord_manhattanDist
60214, 50529, 49944, 43786, 52584, 46425, 52306, 52068, 50243, 45547, 52003, 50234, 55226, 46725,
51145, 56177, 49935, 54639, 56986, 47022, 54522, 54063, 55531, 58463, 58484, 49558, 56119,
55238, 57878, 52953, 47307, 48185, 57295, 53184, 56414, 56413, 50539, 55530, 45552, 51119, 52587,
52879, 56162, 55864, 55582, 29545, 29546, 29837, 29838, 30130, 30423, 30424, 30430, 30430, 30717, 30719,
30721, 31012

# Create a list of all pixels to be searched
pixelList_toBeSearched = []

def pullVehicleMilesSaved(pixel, window, geog, cd, circuity,):
    return 0

def summaryStatisticsFileToDict(summaryStatFile, newHeader):
    dict = 
    for pixelList in summaryStatFile:
        if pixel not in pixelList:
            pixelList.append(pixel)

    return dict

# Create the Output File
output = open(outputFILENAME, "w")

for pixel in pixelList_toBeSearched:
    coord_milesSaved = pullVehicleMilesSaved(pixel, window, geog, cd, circuity,)
    newStats = stats_dict[(pixel, window, geog)] + [coord_milesSaved, coord_totalMiles]
    wr.writerow(newStats)

output.close()
B.6.2 (CD = 3, dCIR = 0.2) — Ridesharing Analysis

```python
import os
import csv
import time
import numpy as np
import scipy.spatial
import operator
import datetime
import math

# HELPER FUNCTIONS
# 

# Deals with the issue that some cells may be empty
def convertToFloat(x):
    if x is None or x in "":
        return float(0)
    else:
        return float(x)

# Function to create a dictionary containing the extra pixel information
def createPixelInfoDictionary_ALL(pixelinfoFILENAME, pixelinfoFILEPATH):
    os.chdir(pixelinfoFILEPATH)
    file = open(pixelinfoFILENAME, "r")
    # Column List in Pixel file: pixel_id, pixel_lat, pixel_long, y, AJ, Pixel, xAJ, Pixel
    pixelinfo_dict = {}
    for line in file:
        l = line.split(" ")
        i = [4] * 4
        # lat, long, y, x
```
pass
return pixelInfo_dict

def extractPixelCoordinatesFromPixelInfoDict(pixel, pixelInfo_dict):
  list = pixelInfo_dict[pixel]
xCoord = int(list[3])
yCoord = int(list[2])
return xCoord, yCoord

# Creates a dictionary of all of the dropoff Pixels in the trip data
  d = {}
  for key, value in tripFileDictionary.iteritems():  # There won't be a header in this tripFile dictionary
    if int(value[20]) not in d:
      d[int(value[20])] = pixelInfoDictionary[int(value[20])]
  return d

def printTime(startTime, endTime, partName):
  duration = endTime - startTime
  if (duration < 1800):
    print("""
    Seconds to run " + partName + "" + str(duration)
  ")
  elif (duration > 60):
    print("""
    Minutes to run " + partName + "" + str(duration / 60)
  ")
  elif (duration > 3600):
    print("""
    Hours to run " + partName + "" + str(duration / 3600)
  ")

# Checks if there are intersecting values between two lists
def intersectionAnalysis(list1, list2):
  intersection_list = list(set(list1) & set(list2))
  intersection_boolean = False
  for row in intersection_list:
    intersection_boolean = True if there's an intersection, change to true
  return (intersection_boolean, intersection_list)

# Pulls the pickup date from a row of data in a tripfile
def extractPickupDate(line):
  line = line.split("\"")
  if not line[6]:
    return "00:00:00-00-00"
  return line[6]

# Pulls the pickup time from a row of data in a tripfile
def extractPickupTime(line):
  line = line.split("\"")
  if not line[7]:
    return "00:00:00"
  return line[7]

# Creates a dictionary of all the trip files
# key = tripID - value = row of data in list form
def createTripFileDict(FILENAME, FILEPATH):
  file = open(FILENAME, "rU")
  d = {}
  header = []
  for line in file:
    line = line.split("\"")
    line[22] = line[22].rstrip("\n")
    if line[0] == "tripID":
      header.append(line[0])
    header.extend(line[1], line[2], line[3], line[4], line[5], line[6], line[7], line[8], line[9], line[10], line[11], line[12], line[13], line[14], line[15], line[16], line[17], line[18], line[19], line[20], line[21], line[22])
    if line[0] == "tripID":
      d[line[0]] = [line[0], line[1], line[2], line[3], line[4], line[5], line[6], line[7], line[8], line[9], line[10], line[11], line[12], line[13], line[14], line[15], line[16], line[17], line[18], line[19], line[20], line[21], line[22]]
  file.close()
  return d, header

# Creates a list where each entry is a list that comprises of a trip
# the indices of this big list are in order the of time (the order of the file being read in)
def createSortedTripFileList(sortedFILENAME, sortedFILEPATH):
  os.chdir(sortedFILEPATH)
  file = open(sortedFILENAME, "rU")
trip_data_list = []
header = []

for line in file:
    line = line.split(',')
    line[22] = line[22].rstrip('"
')
if line[0] == "trip_ID":
    header = line
if line[8] == "trip_ID":
    trip_data_list.append(line)

file.close()

return trip_data_list, header

def extractPickupDatetime(row):

def extractTripID(row):
    dt = datetime.strptime(date, "%Y-%m-%d %H:%M:%S")
    return dt

def extractTripID(row):
    return int(row[8])

def extractPassengerCount(row):
    return int(row[10])

def extractDropoffPixel(row):
    diff = datetime2 - datetime1
    diff = diff.seconds
    return trip

def extractPickupPixel(row):
    return trip

def distance(x1, x2, y1, y2):
    dist = math.sqrt((x1-x2)^2 + (y1-y2)^2)
    return dist

def secondsDifference(datetime1, datetime2):
    dist = distance(startX, listXY[1][0], startY, listXY[1][1])
    dist = distance(listXY[0][0], listXY[2][0], listXY[0][1], listXY[2][1])
    dist = distance(listXY[2][0], listXY[0][0], listXY[2][1], listXY[0][1])
    dist = distance(startX, listXY[0][0], startY, listXY[0][1])

def distance(x1, x2, y1, y2):
    dist = distance(x1, x2, y1, y2)
    return dist

def fitTimeWindow(dataList, timeWindow):
    return dist

def extractPassengerCount(row):
    return int(row[10])

def extractDropoffPixel(row):
    return int(row[20])

def extractPassengerCount(row):
    return int(row[8])

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def distance(x1, x2, y1, y2):
    return dist

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def extractPassengerCount(row):
    return int(row[10])

def extractDropoffPixel(row):
    return int(row[20])

def distance(x1, x2, y1, y2):
    return dist

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def distance(x1, x2, y1, y2):
    return dist

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def distance(x1, x2, y1, y2):
    return dist

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def distance(x1, x2, y1, y2):
    return dist

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def distance(x1, x2, y1, y2):
    return dist

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def distance(x1, x2, y1, y2):
    return dist

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def distance(x1, x2, y1, y2):
    return dist

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def distance(x1, x2, y1, y2):
    return dist

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def distance(x1, x2, y1, y2):
    return dist

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def distance(x1, x2, y1, y2):
    return dist

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def distance(x1, x2, y1, y2):
    return dist

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def distance(x1, x2, y1, y2):
    return dist

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def distance(x1, x2, y1, y2):
    return dist

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def distance(x1, x2, y1, y2):
    return dist

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def distance(x1, x2, y1, y2):
    return dist

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def distance(x1, x2, y1, y2):
    return dist

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def distance(x1, x2, y1, y2):
    return dist

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def distance(x1, x2, y1, y2):
    return dist

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def distance(x1, x2, y1, y2):
    return dist

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def distance(x1, x2, y1, y2):
    return dist

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def distance(x1, x2, y1, y2):
    return dist

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def distance(x1, x2, y1, y2):
    return dist

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def distance(x1, x2, y1, y2):
    return dist

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def distance(x1, x2, y1, y2):
    return dist

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def distance(x1, x2, y1, y2):
    return dist

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def distance(x1, x2, y1, y2):
    return dist

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def distance(x1, x2, y1, y2):
    return dist

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def distance(x1, x2, y1, y2):
    return dist

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def distance(x1, x2, y1, y2):
    return dist

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def distance(x1, x2, y1, y2):
    return dist

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def distance(x1, x2, y1, y2):
    return dist

def secondsDifference(datetime1, datetime2):
    return diff.seconds <= timeWindow

def distance(x1, x2, y1, y2):
    return dist
def pullPixelSpecificData(dataFILENAME, dataFILEPATH, outputFILENAME, outputFILEPATH, outputHeaderBoolean, pickup_pixel_id, dropoff_pixel_id):

    # FUNCTIONS
    
    def writeLine(output, line):
        output.writerow(line)

    def prettyPrintDataLine(pickup, dropoff, line)
        pass

    # Find and Open File
    os.chdir(dataFILEPATH)
    file = open(dataFILENAME, "r")

    # Create the Output File
    os.chdir(outputFILEPATH)
    output = open(outputFILENAME, "w")

    # Write a header in the file if specified
    if outputHeaderBoolean:
        wr = csv.writer(output, lineterminator="\n")
        wr.writerow(header)

    # Counters
    tripSearchCount = 0
    tripMatchCount = 0

    # If we care about only pickup id
    if pickup_pixel_id >= 0 and dropoff_pixel_id < 0:
        print("Searching for all trips with pixel pickup id = " + str(pickup_pixel_id))
    # Search through all trips and save the one that fit the correct specifications
    for line in file
        tripSearchCount += 1
        # Print progress as it is running
        if tripSearchCount % 2000000 == 0:
            print("...currently at row %d + str(tripSearchCount))
        print("...number of matches = " + str(tripMatchCount))
        # Split the string into a list of strings, using comma delimiter
        line = line.split(",")
        line[22] = int(line[22].strip())
        # Pull trip pixel information
        line[pickup_pixel_id] = int(line[pickup_pixel_id])
        # Save to new file if the id's match
        if line[pickup_pixel_id] == pickup_pixel_id:
            tripMatchCount += 1
            wr.writerow(line)

    # If we only care about dropoff id...
    if pickup_pixel_id < 0 and dropoff_pixel_id >= 0:
        print("Searching for all trips with pixel dropoff id = " + str(dropoff_pixel_id))
    # Search through all trips and save the one that fit the correct specifications
    for line in file
        tripSearchCount += 1
        # Print progress as it is running
        if tripSearchCount % 2000000 == 0:
            print("...currently at row %d + str(tripSearchCount))
        print("...number of matches = " + str(tripMatchCount))
        # Split the string into a list of strings, using comma delimiter
        line = line.split(",")
        line[22] = int(line[22].strip())
        # Pull trip pixel information
        line[dropoff_pixel_id] = int(line[dropoff_pixel_id])
        # Save to new file if the id's match
        if line[dropoff_pixel_id] == dropoff_pixel_id:
            tripMatchCount += 1
            wr.writerow(line)
# If we care about both pickup id and dropoff id...
if pickup_pixel_id >= 0 and dropoff_pixel_id >= 0:
    print("Searching for all trips with pixel pickup id " + str(pickup_pixel_id) + " and pixel dropoff id " + str(dropoff_pixel_id) + "")

# Search through all trips and save the ones that fit the correct specifications
for line in file:
    tripSearch_count += 1
    # Print progress as it is running
    if tripSearch_count % 28000000 == 0:
        print("Currently at row " + str(tripSearch_count))
        print("Number of matches " + str(tripMatch_count))
    # Split the string into a list of strings, using comma delimiter
    l = line.split(",")
    l[22] = l[22].rstrip("\n")
    # Pull trip pixel information
    line_pickup_pixel_id = int(l[1][5])
    line_dropoff_pixel_id = int(l[2][6])
    # Save to new file if both of the set of ids are both matches
    if line_pickup_pixel_id == pickup_pixel_id and line_dropoff_pixel_id == dropoff_pixel_id:
        tripMatch_count += 1
        wr.writerow(1)
    wr.writerow(1)

file.close()
output.close()

print("Iterated through a total of " + str(tripSearch_count) + " rows")
print("Found " + str(tripMatch_count) + " matches")

# Sort the trips by date and time
sorted_dict = sorted(dict.items(), key=lambda e: (e[1][6], e[1][7]))

# Create the Output File
os.chdir(outputPATH)
output = open(outputFILENAME, "a")
wr = csv.writer(output, lineterminator="\n")
wr.writerow(output, lineterminator="\n")

# Write the Header Row
wr.writerow(header)

# Write all the other rows in the sorted tuple
for item in sorted_dict:
    wr.writerow(item[1])

output.close()

print("CSV file saved: " + outputFILENAME)

def superPixel(pixelID, pixelInfo_dictionary):
    # superPixel():
    # - takes in a pixel ID
    # - returns a list of all of the pixels that make up the "super pixel"
    # - a "super pixel" consists of the pixel itself, and the 8 pixels surrounding it
    # - (1) (2) (3)
    # - (4) (5) (6)
    # - (7) (8)
    # - a "super pixel" is 0.3x0.3 mi in dimension
    superPixel_list = []
    if pixelID < 0:
        return superPixel_list
    return superPixel_list

    # pixelInfo_dictionary of the format:
    # key = pixel_ID
    # value = [pixel_list, y_AJPixel, x_AJPixel]
    pix_list = pixelInfo_dictionary[int(pixelID)]
    x = int(pix_list[3])
    y = int(pix_list[2])
    pix_id = int(pixelID)
    # convert to int, if it isn’t already
    # Initialize all potential neighbor pixels
    pix_id_1 = -1
    pix_id_2 = -1
    pix_id_3 = -1
    pix_id_4 = -1
    pix_id_5 = -1
    pix_id_6 = -1
    pix_id_7 = -1
    pix_id_8 = -1
    # Calculate the immediate and diagonal neighbors
    topExists = (0 <= y < 289)
    if pixelID is not in top row
    leftExists = (0 <= x < 292)
    if pixelID is not in leftmost column
    rightExists = (0 <= x < 292)
    if pixelID is not in rightmost column
    bottomExists = (0 <= y < 292)
    if pixelID is not in bottom row
    if topExists:
        if not in top row
        pix_id_2 = pix_id + 293
        if leftExists:
            if not in leftmost column
            pix_id_4 = pix_id - 1
if rightExists: # if not in rightmost column
    pixel_{id,4} = pixel_{id} + 1
if bottomExists: # if not in bottom row
    pixel_{id,7} = pixel_{id} - 203

# Calculate the diagonal neighbors
if topExists and leftExists:
    pixel_{id,1} = pixel_{id,2} - 1
if topExists and rightExists:
    pixel_{id,3} = pixel_{id,2} + 1
if bottomExists and leftExists:
    pixel_{id,6} = pixel_{id,2} - 1
if bottomExists and rightExists:
    pixel_{id,8} = pixel_{id,2} + 1

# Create scratch list of all pixel ids
scratch_list = [pixel_{id,1}, pixel_{id,2}, pixel_{id,3}, pixel_{id,4},
                pixel_{id}, pixel_{id,5}, pixel_{id,6}, pixel_{id,7}, pixel_{id,8}]
scratch_list = [item for item in scratch_list if item > 0]
superPixelList = scratch_list

# Create super pixel list of only those that exist
for item in scratch_list:
    if item >= 0:
        superPixelList.append(item)

# print("Scratch List = ")
# print(scratch_list)
# print("Super Pixel = ")
# print(superPixelList)

return superPixel_list

def macroPixel(pixelID, pixelInfo_dictionary):
    # macroPixel()
    # - takes in a pixel ID
    # - returns a list of all of the pixels that make up the "super pixel"
    # - a "super pixel" consists of the pixel itself, and the 24 pixels surrounding it
    # - to calculate it, this function calculates the super pixel of each diagonal neighbor and
    #   then removes duplicates
    # (1)
    # (x, y) (2)
    # (3)
    # (4)
    # - a "macro pixel" is 0.5x0.5 mi in dimension
    macroPixel_list = []
    # pixelInfo_dictionary of the format:
    # key = pixel_ID
    # value = [pixel_list, pixel_{id,long, y_AJ_Pixel, x_AJ_Pixel}]
    pix_list = pixelInfo_dictionary[int(pixelID)]
    x = int(pix_list[2])
    y = int(pix_list[3])
    pixel_{id} = int(pixelID) # convert to int, if it isn't already
    # Initialize all potential neighbor pixels
    pixel_{id,1} = -1
    pixel_{id,2} = -1
    pixel_{id,3} = -1
    pixel_{id,4} = -1
    # Calculate the immediate and diagonal neighbors
    topExists = (0 <= y <= 289) # if pixelID is not in top row
    leftExists = (0 < x <= 292) # if pixelID is not in leftmost column
    rightExists = (0 <= x <= 292) # if pixelID is not in rightmost column
    bottomExists = (0 < y <= 289) # if pixelID is not in bottom row
    # Calculate the diagonal neighbors
    if topExists and leftExists:
        pixel_{id,1} = (pixel_{id} + 293) - 1
    if topExists and rightExists:
        pixel_{id,2} = (pixel_{id} + 293) + 1
    if bottomExists and leftExists:
        pixel_{id,3} = (pixel_{id} - 293) - 1
    if bottomExists and rightExists:
        pixel_{id,4} = (pixel_{id} - 293) + 1
    list1 = superPixel(pixel_{id,2}, pixelInfo_dictionary)
    list2 = superPixel(pixel_{id,2}, pixelInfo_dictionary)
    list3 = superPixel(pixel_{id,3}, pixelInfo_dictionary)
    list4 = superPixel(pixel_{id,4}, pixelInfo_dictionary)
    for item in (list1 + list2 + list3 + list4):
        if not item in macroPixel_list:
            macroPixel_list.append(item)
    # print(macroPixel_list)
    return macroPixel_list

def secondPixelOfFirst(pixel1, pixel2, pixelInfo_dictionary):
    pixel1 = int(pixel1)
    pixel2 = int(pixel2)
    if pixel1 == pixel2:
        # if same pixel
        return True
    elif abs(pixel1 - pixel2) == 1:
        # if horizontally next to each other
        return True
    elif abs(pixel1 - pixel2) == 293:
        # if vertically next to each other
        return True
    else:
        superPixelList = superPixel(pixel1, pixelInfo_dictionary)
if pixel1 in superPixelList:
    return True
else:
    return False
def secondPix_in_macroPixelOfFirst(pixel1, pixel2, pixelInfo_dictionary):
    pixel1 = int(pixel1)
    pixel2 = int(pixel2)
    if pixel1 == pixel2:
        # if same pixel
        return True
    elif abs(pixel1 - pixel2) == 1:
        # if horizontally next to each other
        return True
    elif abs(pixel1 - pixel2) == 293:
        # if vertically next to each other
        return True
    else:
        macroPixelList = macroPixel(pixel1, pixelInfo_dictionary)
        if pixel2 in macroPixelList:
            return True
        else:
            return False
def destinationIntersects(geog, destination_pixel, queryPixel, pixelInfo_dictionary):
    if geog == "MP"
        return secondPix_in_macroPixelOfFirst(destination_pixel, queryPixel, pixelInfo_dictionary)
    if geog == "SP"
        return secondPix_in_superPixelOfFirst(destination_pixel, queryPixel, pixelInfo_dictionary)
def circuitySatisfied(circuity, origin_pixel, currentDest_list, queryPixel, pixelInfo_dictionary):
    circuity = float(circuity)
    origin_X, origin_Y = extractPixelCoords_fromPixelInfoDict(origin_pixel, pixelInfo_dictionary)
    currentDest_pointList = []
    for dest in currentDest_list:
        x, y = extractPixelCoords_fromPixelInfoDict(dest, pixelInfo_dictionary)
        currentDest_pointList.append((x, y))
    old_dist = minDistance_ofCoords(origin_X, origin_Y, currentDest_pointList)
    query_x, query_y = extractPixelCoords_fromPixelInfoDict(queryPixel, pixelInfo_dictionary)
    newDest_pointList = currentDest_pointList
    newDest_pointList.append((query_x, query_y))
    new_dist = minDistance_ofCoords(origin_X, origin_Y, newDest_pointList)
    if old_dist == 0:
        # some trips go to themselves, and thus the distance would be 0
        return False
    elif abs(new_dist - old_dist) / old_dist <= circuity:
        return True
    else:
        return False
def directionalCircuitySatisfied(circuity, origin_pixel, currentDest_list, queryPixel, pixelInfo_dictionary):
    circuity = float(circuity)
    origin_X, origin_Y = extractPixelCoords_fromPixelInfoDict(origin_pixel, pixelInfo_dictionary)
    currentDest_pointList = []
    if geog == "MP"
        firstDestPoint = currentDest_list[0]
        first_X, first_Y = extractPixelCoords_fromPixelInfoDict(firstDestPoint, pixelInfo_dictionary)
    if abs(origin_X - first_X) > abs(origin_Y - first_Y):
        direction = "vertSplit_left"
    else:
        direction = "vertSplit_right"
    elif abs(origin_X - first_X) < abs(origin_Y - first_Y):
        direction = "horizSplit_bottom"
    else:
        direction = "horizSplit_top"
    if abs(origin_X - first_X) == abs(origin_Y - first_Y) and abs(origin_Y - first_Y) > 0:
        if origin_Y - first_Y > 0:
            direction = "vertSplit_left"
        else:
            direction = "vertSplit_right"
    else:
        direction = "none"
    if direction == "none":  # do the old circuity calculation and ignore directional influence
        for dest in currentDest_list:
            x, y = extractPixelCoords_fromPixelInfoDict(dest, pixelInfo_dictionary)
            currentDest_pointList.append((x, y))
        old_dist = minDistance_ofCoords(origin_X, origin_Y, currentDest_pointList)
        query_x, query_y = extractPixelCoords_fromPixelInfoDict(queryPixel, pixelInfo_dictionary)
        newDest_pointList = currentDest_pointList
        newDest_pointList.append((query_x, query_y))
new\_dist = minDistance\_of\_Coords\((orig\_X, orig\_Y, new\_Dest\_pointList)\)

if old\_dist == 0:
    # some trips go to themselves, and thus the distance would be 0
    return False

elif (abs(new\_dist - old\_dist))/old\_dist <= circuity:
    return True
else:
    return False

elif direction == \"vertSplit\_left\":
    query\_x, query\_y = extractPixelCoords\(fromPixelInfoDict(queryPixel, pixelInfo\_dictionary)\)

if origin\_X - query\_x < 0:
    return False
else:
    for dest in current\_Dest\_List:
        x, y = extractPixelCoords\(fromPixelInfoDict(dest, pixelInfo\_dictionary)\)
        current\_Dest\_pointList.append\([x, y]\)

    old\_dist = minDistance\_of\_Coords\((orig\_X, orig\_Y, current\_Dest\_pointList)\)

    new\_Dest\_pointList = current\_Dest\_pointList

    new\_Dest\_pointList.append\([query\_x, query\_y]\)

    new\_dist = minDistance\_of\_Coords\((orig\_X, orig\_Y, new\_Dest\_pointList)\)

    if old\_dist == 0:
        # some trips go to themselves, and thus the distance would be 0
        return False

    elif (abs(new\_dist - old\_dist))/old\_dist <= circuity:
        return True
    else:
        return False

elif direction == \"vertSplit\_right\":
    query\_x, query\_y = extractPixelCoords\(fromPixelInfoDict(queryPixel, pixelInfo\_dictionary)\)

if origin\_X - query\_x > 0:
    return False
else:
    for dest in current\_Dest\_List:
        x, y = extractPixelCoords\(fromPixelInfoDict(dest, pixelInfo\_dictionary)\)
        current\_Dest\_pointList.append\([x, y]\)

    old\_dist = minDistance\_of\_Coords\((orig\_X, orig\_Y, current\_Dest\_pointList)\)

    new\_Dest\_pointList = current\_Dest\_pointList

    new\_Dest\_pointList.append\([query\_x, query\_y]\)

    new\_dist = minDistance\_of\_Coords\((orig\_X, orig\_Y, new\_Dest\_pointList)\)

    if old\_dist == 0:
        # some trips go to themselves, and thus the distance would be 0
        return False

    elif (abs(new\_dist - old\_dist))/old\_dist <= circuity:
        return True
    else:
        return False

elif direction == \"horizSplit\_bottom\":
    query\_x, query\_y = extractPixelCoords\(fromPixelInfoDict(queryPixel, pixelInfo\_dictionary)\)

if origin\_Y - query\_y < 0:
    return False
else:
    for dest in current\_Dest\_List:
        x, y = extractPixelCoords\(fromPixelInfoDict(dest, pixelInfo\_dictionary)\)
        current\_Dest\_pointList.append\([x, y]\)

    old\_dist = minDistance\_of\_Coords\((orig\_X, orig\_Y, current\_Dest\_pointList)\)

    new\_Dest\_pointList = current\_Dest\_pointList

    new\_Dest\_pointList.append\([query\_x, query\_y]\)

    new\_dist = minDistance\_of\_Coords\((orig\_X, orig\_Y, new\_Dest\_pointList)\)

    if old\_dist == 0:
        # some trips go to themselves, and thus the distance would be 0
        return False

    elif (abs(new\_dist - old\_dist))/old\_dist <= circuity:
        return True
    else:
        return False

elif direction == \"horizSplit\_top\":
    query\_x, query\_y = extractPixelCoords\(fromPixelInfoDict(queryPixel, pixelInfo\_dictionary)\)

if origin\_Y - query\_y > 0:
    return False
else:
    for dest in current\_Dest\_List:
        x, y = extractPixelCoords\(fromPixelInfoDict(dest, pixelInfo\_dictionary)\)
        current\_Dest\_pointList.append\([x, y]\)

    old\_dist = minDistance\_of\_Coords\((orig\_X, orig\_Y, current\_Dest\_pointList)\)

    new\_Dest\_pointList = current\_Dest\_pointList

    new\_Dest\_pointList.append\([query\_x, query\_y]\)

    new\_dist = minDistance\_of\_Coords\((orig\_X, orig\_Y, new\_Dest\_pointList)\)

    if old\_dist == 0:
        # some trips go to themselves, and thus the distance would be 0
        return False

    elif (abs(new\_dist - old\_dist))/old\_dist <= circuity:
        return True
    else:
        return False

raise Exception("Directional circuity logic is messed up.")
def corrRidesharing_AVO(dataFILENAME, dataFILEPATH):
    # corrRidesharing_AVO()
    # - inputs = tripdata file
    # - iterates through, row by row, and sums the # of passengers and # of trips
    # - calculates the correlated ridesharing AVO
    # - prints information
    # - returns the correlated ridesharing AVO
    # Find and Open File
    os.chdir(dataFILEPATH)
    file = open(dataFILENAME, "rU")

    # Counters
    tripCount = 0
    passengerCount = 0

    for line in file:
        # Split the string into a list of strings, using comma delimiter
        l = line.split(',
        # Ignore the header line
        if l[0] == "trip.ID"
            continue

        tripCount += 1
        # Pull passenger count from line
        numPass = int(l[10])
        # Add count to the passenger total
        passengerCount += numPass

    file.close()

    # AVO from Correlated Ridesharing
    avo = float(passengerCount) / float(tripCount)

    # Print AVO:
    print("Passenger Count = " + str(passengerCount))
    print("Trip Count = " + str(tripCount))
    print("AVO from Correlated Ridesharing = " + str(avo))

    return avo, passengerCount, tripCount

def reduce_timeWindowDict(oldTimeWindow_dict, TimeWindow_Seconds, tripFile_dict):
    newTimeWindow_dict = {}
    for tripID_key, value in oldTimeWindow_dict.iteritems():
        list = value
        newTimeWindow_dict[tripID_key] = []
        for eachTripID in list:
            # if the tripID fits in the time window, save it
            if fits_timeWindow(extractPickupDatetime(tripFile_dict[tripID_key]), extractPickupDatetime(newTimeWindow_dict[tripID_key]), TimeWindow_Seconds):
                newTimeWindow_dict[tripID_key].append(tripID)

    return newTimeWindow_dict

def create_sortedTripFileList(SORTEDdataFILENAME, SORTEDdataFILEPATH, TimeWindow_Seconds):
    # This dictionary has a key for every tripID in the file
    # and the value is a list of all possible matches
    # including itself first
    timeWindow_dict = {}

    # Create the sorted trip file list by reading in the sorted trip file
    (sortedTFList, h) = create_sortedTripFiles_list(SORTEDdataFILENAME, SORTEDdataFILEPATH)
    i = 0

    for i in range(0, len(sortedTFList)):
        tripID = extractTripID(sortedTFList[i])
        timeWindow_dict[tripID] = []
        timeWindow_dict[tripID].append(tripID)
        j = i+1
        if j >= len(sortedTFList):
            break
        #continue
        
        while fits_timeWindow(extractPickupDatetime(sortedTFList[i]), extractPickupDatetime(sortedTFList[j]), TimeWindow_Seconds):
            timeWindow_dict[tripID].append(extractTripID(sortedTFList[j]))
            j += 1
        if j >= len(sortedTFList):
            break
        #continue

    return timeWindow_dict, sortedTFList

def saveDeparturesToCSV(departuresDict, outputFILENAME, outputFILEPATH, tripFileDict):
    # Create the Output File
    os.chdir(outputFILEPATH)
    output = open(outputFILENAME, "a")

    wr = csv.writer(output, lineterminator="\n")
    wr.writerow(output, lineterminator="\n")

    # Write the Header Row

wr.header.writerow(header)

# Write the other rows
for key, value in departuresDict.iteritems():
    dict.destinations = value
    departure_ID = int(key)
    group = 1
    for destGroup, tripList in dict.destinations.iteritems():
        i = 0
        for tripID in tripList:
            if i == 0:
                dest_origin_trip = 1
                i += 1
            else:
                dest_origin_trip = 0
            oldRow = triPFileDict[tripID]
            newRow = [departure_ID, group, dest_origin_trip]
            newRow = newRow + oldRow
            wr.writerow(newRow)
            group += 1
        output.close()

def common_destination_sharing(cd, SPOrMP, circuity, dictTimeWindow, timeWindowSeconds, sortedTFList, pixelInfoDictionary, tripFileDictionary):
    print("Common Destination Sharing")
    print("""TimeWindow = " + str(timeWindowSeconds) + " seconds and Gens. Boundary = " + SPOrMP)""
    summaryStatsList = []
    summaryStatsList = ["ADO of Casual", "No. of Trips per Departure", "Original Trip Count", "Departure Count", "Departures w/ Tripshare", "% Reduction in Taxi Departures", "% of shared rides"]

departures = {}
passengers = {}
recordedTripIDs = {}
departure_iteration = 0
for i in range(0, len(sortedTFList)):
    trip_row = sortedTFList[i]
    trip = extractTripID(trip_row)
    if trip in recordedTripIDs:
        continue
    else:
        curr_dep = {}
        curr_dep_passenger = {}
        curr_dep["orderedKeys"] = {}
        # to keep track of the order in which keys were added (we need to check directional circuity through the first key)
        curr_dep["departure"] = trip_row
        curr_dep["dropoffPixel"] = extractDropoffPixel(trip_row)
        curr_dep["Keys"] = append(extractDropoffPixel(trip_row))
        curr_dep["passenger"] = extractPassengerCount(trip_row)
        recordedTripIDs[trip] = {}
        possibleTripsList = dictTimeWindow[trip]
        for tripPossibility in possibleTripsList:
            if trip == tripPossibility and tripPossibility not in recordedTripIDs:
                triPFileDict = [tripFile_dictionary[tripPossibility]]
                for dest in curr_dep:
                    if tripPossibility not in recordedTripIDs and destinationIntersects(SPOrMP, dest, tripPossibility)
                        curr_dep["passenger"] = append(extractPassengerCount(tripFile_dictionary[tripPossibility]))
                        triPFileDict = [tripFile_dictionary[tripPossibility]]
                        if not matchFound_forTriPposibility and len(curr_dep) < cd:
                            # for non-directional circuity policies, the function circuitySatisfied() was used instead!
                            if directionalCircuitySatisfied(circuity, dep_originPixel, curr_dep["orderedKeys"],
                                tripPossibility, pixelInfoDictionary)
                                curr_dep["dropoffPixel"] = extractDropoffPixel(tripPossibility)
                                curr_dep["Keys"] = append(tripPossibility)
                                curr_dep["passenger"] = append(extractPassengerCount(tripFile_dictionary[tripPossibility]))
                                recordedTripIDs[trip] = [{matchFound_forTriPposibility = False}]
                                departures[departure_iteration] = curr_dep
                                passengers[departure_iteration] = curr_dep["passenger"]
                                departure_iteration += 1
                                if curr_dep["passenger"] <= 0
                                    total_passengerCount += 0
                                    countUniqueRidesharing += 0
                                else:
                                    total_tripCount += 1
# Count the number of unique rideshares
for dep, destinationDict in departures.items():
    if len(destinationDict) > 1:
        count_uniqueRideshares += 1

# Count the number of passengers
for dep, destinationDict in passengers.items():
    for item in destinationDict:
        passengerNumList = destinationDict[item]
        total_passengerCount = sum(passengerNumList)
        if total_passengerCount > 1:
            raise Exception("Trip count from TF List and trip count from Departures dictionary do not match.")

print("% of Departures that are shared")

summaryStatList.append(count_uniqueRideshares)  # ADO of passengers
summaryStatList.append(len(departures))  # No. of Departures
summaryStatList.append(count_uniqueRideshares)  # ADO of original trips
summaryStatList.append(sum(count_uniqueRideshares))  # % reduction in taxi departures
summaryStatList.append(len(departures))  # % of departures that are shared

return departures, summaryStatList
```python
timeWindowList = [300, 120, 60, 30]

for pixelInfo in PixelInfoDictionary:
    print("Step 2")
    print("Create the Pixel Info Dictionary")
    print_time(Step_2_TIMER_START, time.clock(), "Step 1")

    # Create the Pixel Info Dictionary
    pixelInfo_TIMER_START = time.clock()
    print("[" + PixelInfoDictionaryCreation + "]")
    print("Print PixelInfo Dictionary Size = " + str(len(dict(pixelInfo)))
    print_time(Step_2_TIMER_END, time.clock())
    print_time(Step_2_TIMER_START, time.clock(), "Step 2")

    # Create a summary-statistic file to be filled out as the analysis is completed
    summaryFileNAME = "SummaryStatistics_top84.csv"
    summaryFileNAME_middle218 = "SummaryStatistics_middle218.csv"
    summaryFileNAME_last1947 = "SummaryStatistics_last1947.csv"

    summarySTATFilePATH = "C:\Users\swoho\Thesis\Analysis\CD3\CHR 0:2\Directional\"
    os.chdir(summarySTATFilePATH)
    summarySTAT_output = open(summarySTATFileNAME_top84, "a")

    summaryStatWR = csv.writer(summarySTAT_output, lineterminator="\n")
    summaryStatWR.writerow(header)

    print_time(step2_TIMER_START, time.clock(), "Step 2")
```

1102.5 pixels_last1947_part1 = "[not included in code for the report because it is too long of list]"

1102.5 pixels_last1947_part2and3 = "[not included in code for the report because it is too long of list]"

1102.5 pxelID Lists to be Used over three computers

1102.5 pixels_top84 = "[not included in code for the report because it is too long of list]"

1102.5 pixels_middle218 = "[not included in code for the report because it is too long of list]"

1102.5 pixels_last1947 = pixels_last1947_part1 + pixels_last1947_part2and3

1102.5 printTime(step1_TIMER_START, time.clock(), "Step 1")

1102.5 print("Step 2")

1102.5 # Create Pixel Dictionary and Summary-Statistic File""

1102.5 step2_TIMER_START = time.clock()

1102.5 @ Create the Pixel Info Dictionary

1102.5 pixelInfo_TIMER_START = time.clock()

1102.5 print("[" + PixelInfoDictionaryCreation + "]")

1102.5 print("Print PixelInfo Dictionary Size = " + str(len(dict(pixelInfo)))

1102.5 print_time(Step_2_TIMER_END, time.clock())

1102.5 print_time(step2_TIMER_START, time.clock(), "Step 2")

1102.5 # Create a summary-statistic file to be filled out as the analysis is completed

1102.5 summaryFileNAME_top84 = "SummaryStatistics_top84.csv"

1102.5 summaryFileNAME_middle218 = "SummaryStatistics_middle218.csv"

1102.5 summaryFileNAME_last1947 = "SummaryStatistics_last1947.csv"

1102.5 summarySTATFilePATH = "C:\Users\swoho\Thesis\Analysis\CD3\CHR 0:2\Directional\"

1102.5 os.chdir(summarySTATFilePATH)

1102.5 summarySTAT_output = open(summarySTATFileNAME_top84, "a")

1102.5 summaryStatWR = csv.writer(summarySTAT_output, lineterminator="\n")


1102.5 summaryStatWR.writerow(header)

1102.5 print_time(step2_TIMER_START, time.clock(), "Step 2")
for pixel in pixelListToBeSearched:
    step_2_time = time.clock()
    # Create tripFile dictionary for the "pixel"
    SORTED_FILENAME = SORTED_tripFile_dict + str(pixel) + "_pickups.csv"
    (dict_tripFile, h) = create_tripFile_dict(SORTED_FILENAME, SORTED_data_filepath)
    print('dict_tripFile' + str(pixel) + 'tripFileDict length = ' + str(len(dict_tripFile)))
    # Calculate Correlated Ridesharing
    (corr_avo, corr_pass, corr_trip) = corrRidesharing_AVO(SORTED_FILENAME, SORTED_data_filepath)
    step_3_iEND = time.clock()
    partName = 'Preparation- Creation for Pixel' + str(pixel)
    printTime(step_2_iSTART, step_3_iEND, partName)
    # Conduct Casual Ridesharing Analysis
    previous_dictTIMEWINDOW = {}
    firstTimeWindow = True
    for timeWindow in timeWindowList:
        print('("\n" + repeat(' + ' " - " + str(timeWindow) + " seconds")' + ' + repeat(' + ' " - " + str(timeWindow) + " seconds")' + ' + repeat(' + ' " - " + str(timeWindow) + " seconds")' + ' + repeat(' + ' " - " + str(timeWindow) + " seconds")' + ' + repeat(' + ' " - " + str(timeWindow) + " seconds")' + ' + repeat(' + ' " - " + str(timeWindow) + " seconds")' + ' + repeat(' + ' " - " + str(timeWindow) + " seconds")' + ' + repeat(' + ' " - " + str(timeWindow) + " seconds")

        if firstTimeWindow:
            start_part = time.clock()
            (dict_TIMEWINDOW, sortedTF) = create_timeWindowMatches_dict(SORTED_FILENAME, SORTED_data_filepath, timeWindow)
            pixelSORTED_TripFileList = sortedTF
            part = time.clock()
            printTime(start_part, start_part + " - " + str(part) + " - " + str(timeWindow) + " - " + str(timeWindow))
            firstTimeWindow = False
            else:
                start_part = time.clock()
                dict_TIMEWINDOW = reduce_timeWindowDict(previous_dictTIMEWINDOW, timeWindow, dict_tripFile)
                start_part = time.clock()
                printTime(start_part, start_part + " - " + str(part) + " - " + str(timeWindow) + " - " + str(timeWindow))
                dict = {} + repeat(' + ' " - " + str(timeWindow) + " seconds")

        # Super Pixel Ridesharing Analysis
        time1 = time.clock()
        (dict_departures, stats_SP) = common_destination_S_ridesharing(commonDest, "SP", circuity,
        dict_TIMEWINDOW, timeWindow, dict_SORTED_TripFileList, dict_pixelInfo, dict_tripFile)
        departures_output_FName_SP = str(pixel) + "SP" + str(timeWindow) + "CD + str(commonDest) + "
        for departures, output_FName in departures_output_FPath:
            saveDeparturesToCSV(departures, output_FName, departures_output_FPath, dict_tripFile)
        additional_stat_info_SP = (pixel, timeWindow, commonDest, circuity, "SP", corr_avo)
        summaryStat_write(additional_stat_info_SP + stats_SP)
        dict = departures.clear()
        gc.collect()
        print("(" + str(timeWindow) + ") (Pixel' + str(pixel) + " + " + str(timeWindow) + ")")
        printTime(time1, time1 + " - " + part1, part1)
        print('"\n")

        # Macro Pixel Ridesharing Analysis
        time2 = time.clock()
        (dict_departures, stats_MP) = common_destination_M_ridesharing(commonDest, "MP", circuity,
        dict_TIMEWINDOW, timeWindow, dict_SORTED_TripFileList, dict_pixelInfo, dict_tripFile)
        departures_output_FName_MP = str(pixel) + "MP" + str(timeWindow) + "CD + str(commonDest) + "
        for departures, output_FName in departures_output_FPath:
            saveDeparturesToCSV(departures, output_FName, departures_output_FPath, dict_tripFile)
        additional_stat_info_MP = (pixel, timeWindow, commonDest, circuity, "MP", corr_avo)
        summaryStat_write(additional_stat_info_MP + stats_MP)
        dict = departures.clear()
        gc.collect()
        print("(" + str(timeWindow) + ") (Pixel' + str(pixel) + " + " + str(timeWindow) + ")")
        printTime(time2, time2 + " - " + part2, part2)

        previous_dictTIMEWINDOW = dictTIMEWINDOW
        summaryStat_write.close()
        # End the file timer
        print('"\n")
        print('"\n")
        printTime(start_time, time.time(), "ENTIRE PROGRAM")
Bibliography


