Thesis

Ákos Balázs

2022

Corvinus University of Budapest

The welfare effect of platforms on secondary ticket markets

Data, model and simulation

Ákos Balázs
Economic Analysis MSc
Industrial Organisation Analyst Specialisation

Supervisor: Barna Bakó

11th December 2022

Abstract

This paper investigates how platform rules affect the magnitude and composition of social surplus on secondary ticket markets. We first analyse a large dataset we have scraped from a popular secondary ticket market platform, TicketSwap. Our key variable in this analysis is the ratio between a ticket's secondary and primary prices. We examine its distribution, and then run fixed-effect regressions to see how other variables influence it. In particular, we find that these price ratios tend to decrease as the event comes closer in time. Next, we design a theoretical model of ticket markets. In this model, buyers and secondary sellers arrive at the market according to two Poisson processes. Arriving secondary sellers price their tickets in a way that maximises their expected payoffs, while arriving buyers simply base their purchase decisions on their reservation prices. We then run simulations based on this model, using two different specifications. The first ('constrained') specification reflects the rules of TicketSwap: the platform takes a service fee after each transaction, and the prices have lower and upper limits. The second ('unconstrained') specification does not include these rules. Having obtained the two simulated datasets, we conduct the same analysis on them as on the TicketSwap data. We find that the constrained artificial dataset behaves very similarly to the one we have scraped from TicketSwap. Hence we argue that our model provides a sufficient description of ticket markets. Finally, we use the two simulated datasets to investigate the welfare implications of TicketSwap's constraints. We find that although both datasets are far from being Pareto-optimal, the constraints result in only a slight decrease of welfare. Most of the inefficiency is caused by the market's dynamic nature, which cannot be eliminated unless the platform uses auctions. However, when we also check the composition of social surplus in the two cases, we can see that TicketSwap's rules redistribute a substantial amount of welfare from the group of secondary sellers to the group of buyers. Hence we conclude that the constraints are almost neutral for society as a whole, but strictly beneficial to buyers.

Contents

List of Tables

List of Figures

Acknowledgements

Supported by the ÚNKP-22-2-I-CORVINUS-68 New National Excellence Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund.

Új Nemzeti Kiválóság Program

Ministry of Culture and Innovation

During my research I received a great amount of help from several people. Most of all, I would like to thank my supervisor Barna Bakó for the support he gave me — regarding both the paper itself and my career in general. Having him as a mentor now is one of the best outcomes of my master's studies. Second, I am very grateful to my friend Attila Török, who basically taught me how to code in Python and guided me throughout the whole webscraping process. He was always extremely helpful and patient, even after the hundredth question I had bothered him with. I am also indebted to Máté Sándor, László Radnai and Tibor Keresztély for their additional help.

Finally, I would like to thank TicketSwap for unknowingly letting me scrape their data. By doing so, I must have violated their terms and conditions, but I have used this dataset for scientific purposes only, so I hope they forgive me for it.

ticketswap

1 Introduction

"*So ya*
Thought ya
Might like to go to the show"
(Waters, 1979)

It often happens that people buy tickets for a certain concert but end up not showing up. Unexpected sicknesses, the lack of time, or simple mood swings are among the most common reasons for this. Similarly, there are people who would like to attend but cannot get tickets — either because they have been sold out or because they are too expensive. Therefore, it is mutually beneficial for both groups to form a secondary market for tickets. This way those who have spare tickets can sell them to those who need them. However, it is not easy for potential buyers and sellers to find each other. And even if they do so, the buyer faces the risk of being scammed by the seller. Hence transaction costs on such markets are very high (Leslie & Sorensen, 2009).

In recent years, this issue has lead to the emergence of online platforms on secondary ticket markets. These platforms offer a safe and convenient way of buying and selling tickets for all kinds of events. A seller can upload her ticket anytime before the event, for a price of her choice, and the first buyer to pay this price receives the ticket immediately. However, this convenience comes with some drawbacks. The dynamic nature of these markets already causes some inefficiency, and users even need to pay service fees to the platform after each transaction. What is more, some platforms also use lower and upper limits on the prices that sellers can set. These rules can be seen as market distortions.

The aim of our research is to see how serious these distortions are, and how they affect the extent and composition of welfare on ticket markets. We take a three-part approach to this problem: first, we analyse empirical data, then create a theoretical model, and finally run simulations. The dataset we use has been scraped from one of the most popular secondary ticket market platforms in Europe, TicketSwap (TicketSwap B.V., 2022a). It contains observations on 353269 tickets for 2847 events near Amsterdam. This dataset is in itself very valuable due to its size and quality. In its analysis, we focus on the factors that influence the secondary-market prices of tickets (relative to their original prices). Using fixed-effect regressions, we show that prices are on average lower when the event is closer in time. In our theoretical model we assume Poisson arrivals for buyers and sellers. Like at TicketSwap, prices are set by sellers, whose preferences are represented by an expected utility function. Meanwhile, buyers decide whether or not they want to buy a certain ticket based on their reservation prices. This is a novel approach that is in itself a contribution to the literature. The simulations we run also rely on this model. To be able to answer our research question, we run them with two specifications: one where the above-mentioned constraints (service fees, price limits) are present, and one where they are not. We find that in the former case, the simulated dataset exhibits the same patterns as the original dataset from TicketSwap, so our model describes these ticket markets properly. Hence we can use the simulated data to answer our research question. We find that the platform's constraints do not result in a substantial decrease of welfare, but they redistribute a huge surplus from secondary sellers to buyers. This finding draws a rather favourable picture of secondary ticket market platforms, especially if we add that they reduce transaction costs. Most of the inefficiency on these markets is caused by their dynamic nature, and not the platform rules.

Our paper is structured as follows. In chapter 2 we give a brief summary of previous research on secondary ticket markets, and show how our paper contributes to this literature. Then in chapter 3 we present our TicketSwap dataset and how we obtained it. Chapter 4 describes our model's assumptions and how we obtain the formula for a key variable. In chapter 5 we show how we used this model to run simulations, and then analyse the two simulated datasets. Next, in chapter 6 we conduct our welfare analysis and draw conclusions from it. Finally, we make a few remarks about the limitations of our research in chapter 7, and then discuss our findings in chapter 8. Our codes are included in appendix A.

2 Literature review

The literature on ticket markets is quite extensive. From the perspective of economics, tickets are perishable goods (Karp & Perloff, 2005; Sweeting, 2012; Waisman, 2021). After the corresponding event ends, they are no longer valuable to consumers. This 'expiration date' is what makes them special in the first place, giving their markets a dynamic aspect. This also means that their prices are subject to dynamic pricing strategies, as Sweeting (2012) shows. In his paper he tests how accurately a dynamic pricing model describes secondary ticket market data of Major League Baseball (MLB) matches in America. He finds that, in line with his model's predictions, sellers tend to lower their prices as the event gets closer in time. These results are reflected in our findings as well, see sections 3.3 and 5.3. However, while Sweeting uses a very simple discrete-time model, our model is a more complex one where time is continuous.

There are also many papers that deal specifically with secondary ticket markets, and how they are linked to primary ones. One of the observations that puzzle economists the most is that secondary prices are usually much higher than primary ones (Courty, 2000, 2003a; Krueger, 2001; Budish & Bhave, in press). This observation indicates that the tickets on the primary market are underpriced, and hence their original sellers do not exhibit profit-maximising behaviour. Several alternative explanations have been given to this surprising result, as reviewed by Krueger (2001).

But what is more important form our perspective is that this gap in the prices leads to an arbitrage opportunity. Many people try to take advantage of this: they buy tickets from the primary market only to resell them on the secondary one. The colloquial term for this behaviour is the rather pejorative 'ticket scalping'. It is generally regarded by the public as outrageous. Even Roth (2007) includes ticket scalping in his list of repugnant transactions. This unfavourable public opinion has lead to the introduction of the so-called anti-scalping laws in many countries and US states.

Needless to say, most economists view these laws as unnecessary restrictions that distort the market and cause inefficiency (Williams, 1994). They argue that the problem is not with ticket scalping, but with the unreasonably low prices on the primary market, which result in tickets getting to people who value them less. From this perspective, ticket scalping is just the market's way of correcting for this inefficiency. But these are just the most general free-market arguments which we always hear from economists whenever someone initiates some market restrictions. If we want to draw conclusions on ticket markets in particular, we need to be more specific than that. In fact, if we look deeper into the literature of ticket markets, we find arguments both for and against anti-scalping laws.

Williams (1994) uses empirical data from the American National Football League (NFL) to test the effects of anti-scalping laws on primary ticket prices. He finds that prices are on average lower where these laws are present, and hence argues that they are bad for primary sellers. However, other studies find just the opposite. Depken (2007) first presents a theoretical model in which ticket scalping has an ambiguous impact on primary market prices, and then also uses MLB and NFL data to test which direction prevails empirically. His results indicate that primary tickets tend to be more expensive in cities where anti-scalping laws have been introduced. Hence, his conclusion is the opposite of what Williams (1994) claims: anti-scalping laws favour primary sellers. More ambiguous results are shown in the papers of Courty (2003b) and Karp and Perloff (2005), which both use theoretical models of a monopoly on the primary market. Courty (2003b) compares several possible strategies and finds that the monopolist cannot do strictly better by allowing resale. Karp and Perloff (2005) take an information-theoretical approach. They show that the results depend on the monopoly's ability to intertemporally price discriminate, and the degree of informational asymmetry in their model. However, they argue that under the most realistic specification (when intertemporal price discrimination is possible and there is a small degree of informational asymmetry), anti-scalping laws are bad for the monopolist.

While the above papers focus on the surplus of *primary sellers*, Leslie and Sorensen (2009) estimate the effects of anti-scalping laws on the aggregate social welfare. Their approach is the one which is the closest to ours, so we describe their results in more detail. First, they develop a theoretical model of buyer decisions which also takes transaction costs into account. This model is similar to ours in some aspects (e.g. it also assumes Poisson arrivals). Next, they use a novel dataset of rock concerts which includes both primary and secondary market data (they claim to be the first ones to have such a dataset). They estimate the parameters of their model on this dataset using a structural model. Their findings indicate that ticket scalping is overall beneficial to society, but high transaction costs may offset these benefits. On the other hand, the group of buyers is strictly worse off due to ticket scalping, so a social planner whose goal is to maximise consumer surplus should impose anti-scalping laws. These results are completely parallel to what we find in chapter 6 (if we treat anti-scalping laws analogously to the constraints on TicketSwap).

There is yet another group of papers in the literature of ticket markets that use a different approach. Instead of arguing whether the current mechanism is efficient or not, they use auction theory to design a socially optimal allocation mechanism. Two papers by Miyashita (2014, 2017) propose an online double auction for markets of perishable goods. However, Miyashita focuses mainly on another type of perishable goods, food. Budish and Bhave (in press) and Waisman (2021) use datasets from actual ticket markets where auctions have been implemented and test empirically whether these mechanisms achieved their desired effect. Budish and Bhave (in press) use observations from a setting where auctions were introduced on the primary market. They find that these auctions tend to improve price discovery, increase primary seller profits and eliminate arbitrage opportunities for scalpers. The dataset of Waisman (2021) is from a secondary market where buyers could choose from auctions and posted prices. The author estimates a structural model on this data and finds that while sellers benefit from menus of different selling mechanisms, buyers would be better off on an auction-only platform.

To sum up, many papers deal with welfare on secondary ticket markets. Most of these papers focus on the effects of anti-scalping laws, while others propose auctions. However, to our knowledge, no paper has addressed the welfare effect of platforms and their restrictive rules. We can of course argue that the constraints on these platforms are analogous to anti-scalping laws (which is what we find eventually) but we need to check this hypothesis first. Therefore, our papers fills in a gap in the existing literature. What is more, the size of our dataset and the fact that it comes from European ticket markets (and not American ones like in all other cases) also makes our research valuable. The novel approach of our model and the simulations based on it are important contributions as well.

3 Data

In this chapter, we present the dataset we have scraped from TicketSwap. But before doing so, in section 3.1 we first show what exactly TicketSwap is. Then in section 3.2 we make a few remarks on how we have obtained the dataset. Finally, in section 3.1 we describe and analyse our data. The corresponding codes are included in appendices A.1, A.2, and A.3.

3.1 About TicketSwap

TicketSwap is an online platform for buying and selling e-tickets (TicketSwap B.V., 2022a). It supports all kinds of events (e.g. concerts, festivals, even exhibitions). Each event has its own unique page on the platform, which is where trading happens. If the page of a certain event has not been created yet, users can create it themselves. This page can be seen as the secondary marketplace for the event's tickets. In the case of an event which has several event types[1], the page has several subpages for each event type. An example for such a subpage (for floor tickets to Roger Waters' Amsterdam concert) is shown in figure 3.1a.

On each page (and subpage), users can see the prices of all currently available ticket listings[2], as well as the last three sold listings. They can also see the number of available and sold tickets. Users who are interested in buying a ticket for a certain event (or event type) can subscribe for ticket alerts on its page. If they do so, they get notified whenever a new ticket becomes available. The number of users subscribed for ticket alerts is also displayed on each page and subpage (in figure 3.1a this is the 'WANTED' field).

A user who would like to sell tickets first needs to upload them to the platform, and select (or create) the corresponding event's page. She also needs to submit the original price of the tickets on the primary market. Then she is asked to set a so-called 'selling price', as can be seen in figure 3.1b. Rather counter-intuitively, this selling price is not what the seller gets after a transaction, nor is it what the buyer pays for the ticket. It is merely a theoretical price between the two, from which the platform calculates the real prices. The seller of the ticket receives only 95% of the selling price, the remaining 5% goes to TicketSwap (in the form of service fees). The buyer also pays 5% of the selling price as service fees, as well as an additional 3% as transaction fees (TicketSwap B.V., 2022a). This in effect means that a buyer will pay $1.05 \cdot 1.03 = 1.0815$ times the selling price, so the final ratio of what the seller gets and what the buyer pays will be $\frac{0.95}{1.0815} \approx 0.8784$. Therefore, TicketSwap collects about 12.16% of what the buyer pays after each transaction.

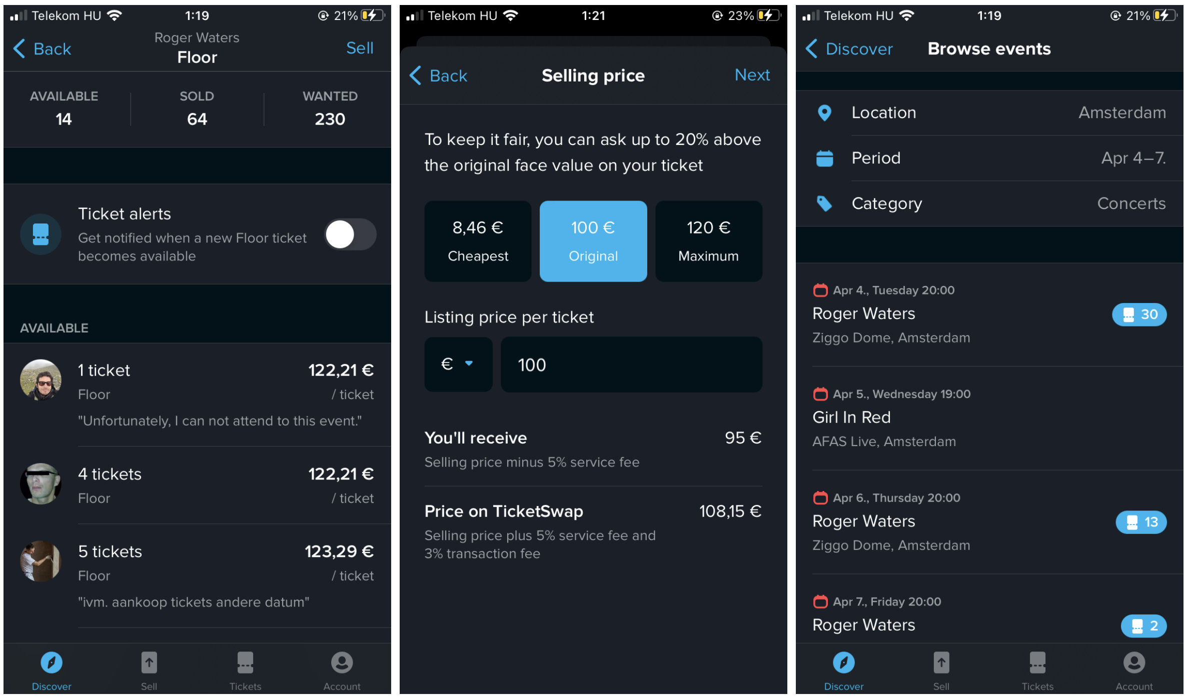
The price that a seller can set is also limited. As stated in figure 3.1b, the selling price must not exceed the ticket's original price by more than 20% (TicketSwap B.V., 2022a). Together with what we have shown in the previous paragraph, this means that the upper limit on the prices that buyers pay is $1.2 \cdot 1.0815 = 1.2978$ times the original price. There is also a lower limit on prices that is not indicated in figure 3.1b. This limit varies from currency to currency, but for euro transactions the minimal selling price that a seller can choose is[3] €5 (TicketSwap B.V., 2022a). Converted to buyer prices, this is $5 \cdot 1.0815 = 5.4075$ euros.

To illustrate these confusing rules with a simple example, assume that a user has bought a ticket for €100, and she would now like to sell it on TicketSwap. Therefore, she can offer this ticket to buyers for

[1]For example, for a festival people can either buy full festival passes or just tickets for a single day. But even some concert venues offer several options (e.g. seated, standing, VIP).

[2]In the TicketSwap terminology, a listing is the set of tickets for a certain event (type) uploaded by the same user at

[3]Some readers might be wondering about what happens when this lower limit exceeds the upper one, i.e. when the original price is below $\frac{5}{1.2} = 4.1\dot{6}$ euros. If this is the case, the selling price has to be exactly €5 (TicketSwap B.V., 2022b). However, such low original prices are extremely rare, so we do not need to be concerned about this issue.


Telekom HU 1:19 21%
Back
Roger Waters
Floor
Sell

AVAILABLE	SOLD	WANTED
14	64	230

Ticket alerts
Get notified when a new Floor ticket becomes available

AVAILABLE

1 ticket	122,21 €
Floor	/ ticket
"Unfortunately, I can not attend to this event."	
4 tickets	122,21 €
Floor	/ ticket
5 tickets	123,29 €
Floor	/ ticket
"ivm. aankoop tickets andere datum"	

Discover Sell Tickets Account

Telekom HU 1:21 23%
Back
Selling price
Next

To keep it fair, you can ask up to 20% above the original face value on your ticket

8,46 €	100 €	120 €
Cheapest	Original	Maximum

Listing price per ticket
€ 100

You'll receive	95 €
Selling price minus 5% service fee	
Price on TicketSwap	108,15 €
Selling price plus 5% service fee and 3% transaction fee	

Telekom HU 1:19 21%
Discover
Browse events

Location	Amsterdam
Period	Apr 4–7.
Category	Concerts

Apr 4., Tuesday 20:00
Roger Waters 30
Ziggo Dome, Amsterdam

Apr 5., Wednesday 19:00
Girl In Red
AFAS Live, Amsterdam

Apr 6., Thursday 20:00
Roger Waters 13
Ziggo Dome, Amsterdam

Apr 7., Friday 20:00
Roger Waters 2

Discover Sell Tickets Account

(a) An event's page (b) Selling a ticket (c) Searching for events

Figure 3.1: Screenshots from the TicketSwap application (TicketSwap B.V., 2022b).

any price between €5.4075 and €129.78. If her ticket is bought, she will receive about 87.84% of what the buyer has paid for it.

As figure 3.1b shows, when a seller is choosing her price, three possible prices are already listed above. These are the the price of the cheapest currently available ticket, the original price at the primary seller, and the upper price limit. Although sellers do not have to choose any of these three options (as we have mentioned, they can choose any price between the lower and upper limits), we will see that they often do so. Hence it is important to keep these options in mind when analysing our dataset.

Another important feature of the TicketSwap webpage is that a user can search for events based on their location, date and category. An example for such a search is shown in figure 3.1c. This feature plays a crucial role in the webscraping process, as we will show in the following section.

3.2 Obtaining the dataset

When we conduct a search on TicketSwap (like in figure 3.1c) or visit a certain event's page (like in figure 3.1a), the data we see comes from a huge dataset stored on TicketSwap's server. Our browser obtains this data by requesting it through the so-called API[4]. Then the server sends back a response with the information we asked for. In the case of TicketSwap, requests must be SQL queries, and the corresponding responses come in `.json` files.

Our webscraping algorithm is built on these foundations. It basically consists of two steps:

1. We post a search query (called `getPopularEvents`) to the API, asking for the most popular 99 events at a certain date and location. From the response, we record the data of each search result into our database of events.

2. For each event in our database, we post a query asking for further data (called

[4] An API (application programming interface) can be thought of as the means of communication between our browser and TicketSwap's server.

`getEventStructuredData`[5]). From the responses, we record the current data of each event, and of the available and sold listings for it.

We have scraped our dataset from TicketSwap simply by repeating this process as many times as we could. Our pool of events grew after every `getPopularEvents` query, and further observations were made on the same events and listings after each `getEventStructuredData` query. In the search queries, we did not specify the category because we wanted to include all kinds of events (concerts, festivals, etc.) in our dataset. However, we did modify the date each time, in order to get events from a wide time interval.

We could have of course also modified the location in the search query each time. However, due to our limited webscraping capacity, we were facing a trade-off here: we could either have a small amount of data from many places, or a lot of data from one specific location. We figured that the latter was better for carrying out a proper analysis, so we have chosen to restrict our research to one specific area. This way we could also avoid having to deal with the possible heterogeneity among different areas[6]. But still, we wanted to have as many observations as possible, so we needed to find the place where the use of TicketSwap was the most widespread. Since TicketSwap was founded in the Netherlands, the Dutch capital seemed like an obvious first guess. After checking several big European cities, we have found that our intuition was right. The circulation of tickets for events around Amsterdam outscored all other cities with a high margin. Therefore, we have used Amsterdam in each of our search queries.

The data we obtained after the webscraping process was 'raw' in the sense that its structure did not make it suitable for further analysis. Hence we needed to 'tidy' it: define new variables, join different data frames, etc. This process is rather technical so we will not go into detail here (the codes can be seen in section A.2). However, what needs to be mentioned is that we also filtered our dataset as part of the tidying process. In particular, we dropped observations where the listing's current selling price exceeded the upper limit of 1.2 times the original price (see section 3.1). We did this because they contradicted TicketSwap's explicitly stated policy of not letting anyone have a mark-up above this limit (TicketSwap B.V., 2022a). Hence we could only see these observations as data imperfections. Similarly, we also excluded observations of listings which were made after the corresponding event had already ended.

3.3 Description and analysis

The dataset we use in this paper was scraped from the TicketSwap API between 14 July 2022, 20:02 and 20 November 2022, 19:37. It contains 376242 observations of 237169 ticket listings, with 353269 tickets inside them. These tickets were uploaded by 165423 different sellers, to the TicketSwap subpages of 3834 event types (related to 2847 events) occurring near Amsterdam. Table 3.1 includes the description of each variable in our dataset[7].

As can be seen from table 3.1, in our dataset we have observations on the ticket listings, and not the tickets themselves. What is more, the very same listings may be recorded several times if their data has changed (e.g. some of the tickets has been sold or the price has been modified) during the webscraping process. Analysing the data in this form would lead to biased results. Instead, we transform our dataset: we replicate each observation as many times as its *NumberSoldNext* variable indicates[8]. This way we obtain an 'unweighted' dataset where the observation units are the tickets themselves: each observation corresponds to one ticket only, and each ticket is counted exactly once. We conduct our further analysis on this unweighted dataset. Table 3.2 includes the summary statistics of its numeric variables.

[5]This query is a modified version of what is posted when we open a certain event's page on TicketSwap.

[6]Although the analysis of such differences could yield interesting results, it would also shift our focus from the main topic of this paper.

[7]The definitions of variables that have the term '*Next*' in their names are a bit more complicated than what is shown in the table. For more detailed definitions, see our codes in appendix A.2.

[8]This also means that an observation whose *NumberSoldNext* is zero will be excluded from our dataset.

Variable	Description
EventID	The unique ID of the currently observed event
EventName	The name of this event
TypeID	The unique ID of the currently observed event *and* event type
TypeName	The name of this event type
Country	The ISO 3166 code of the country where this event takes place at ('NL' for all observations)
City	The name of the city where this event takes place at
Category	The category of this event (e.g. concert, festival, sports)
StartDate	The date and time when this event type starts
EndDate	The date and time when this event type ends
Status	Whether this event is currently active, expired, cancelled, or on hold
NumberOfAvailable	The current number of available tickets for this event type
NumberOfSold	The current number of tickets for this event type that have been sold on Ticketswap
NumberOfAlerts	The current number of buyers subscribed for this event type's ticket alerts
Popular	Whether this event type is currently popular (according to TicketSwap) or not
SoldOut	Whether this event's tickets at the primary seller are currently on sale, sold out, or disabled
TimeRecorded	The date and time of the event type's current observation
ListingID	The unique ID of the currently observed ticket listing
SellerID	The unique ID of this listing's seller
TicketsInListing	The number of tickets in this listing
TicketsStillForSale	The number of tickets in this listing which are still available
OriginalPrice	The original price that the seller has paid for one ticket in this listing (on the primary market)
OriginalCurrency	The currency of *OriginalPrice*
SellerPrice	The seller's chosen 'selling price' (see section 3.1) for one ticket in this listing
SellerCurrency	The currency of *SellerPrice*
TotalPrice	The total price that its buyer will pay for one ticket in this listing (1.0815 times *SellerPrice*)
TotalCurrency	The currency of *TotalPrice* (always the same as *SellerCurrency*)
TimeRecordedTicket	The date and time of the ticket listing's current observation
Sold	Whether this listing is a sold listing (or an available one)
SellerPriceLimit	The upper limit on the selling price (1.2 times *OriginalPrice*) for this listing
RealEnd	The same as *EndDate*, but when its value is missing, *StartDate* is used instead
NumberSoldNext	The number of tickets from this listing that will have been sold when we next observe them (to sold listings that have never been observed as available, we assign their *TicketsInListing*)
PriceRatioNow	The ratio of *TotalPrice* and *OriginalPrice* (if their currencies are the same)
NextPriceRatio	The *PriceRatioNow* of tickets in this listing when we next observe them
NextSeen	The date and time of the same ticket listing's next observation
FirstSeenAvailable	The date and time of the ticket listing's first observation
TimeLeftNow	The amount of time between *TimeRecordedTicket* and *RealEnd* (in seconds, if non-negative)
TimeLeftWhenAvailable	The amount of time between *FirstSeenAvailable* and *RealEnd* (in seconds, if non-negative)
TimeLeftNext	The amount of time between *NextSeen* and *RealEnd* (in seconds, if non-negative)
TimeAvailableFor	The amount of time between *NextSeen* and *FirstSeenAvailable* (in seconds, if non-negative)
EverSold	Whether any of the tickets in the currently observed listings would end up being sold later

Table 3.1: The description of variables in the original TicketSwap dataset

Statistic	N	Mean	St. Dev.	Min	Median	Max
NumberOfAvailable	353 269	76.98	152.56	0	11	1 074
NumberOfSold	353 269	866.19	1 468.79	0	266	10 569
NumberOfAlerts	353 269	950.26	1 655.66	0	408	33 297
TicketsInListing	353 269	2.10	1.70	1	2	28
TicketsStillForSale	353 269	0.99	1.48	0	0	24
OriginalPrice	353 269	56.30	1 738.97	4	49.5	999 999
SellerPrice	353 269	49.60	120.09	5	45.4	29 353
TotalPrice	353 269	53.64	129.87	5.41	49.10	31 745.00
SellerPriceLimit	353 269	67.56	2 086.76	5	59.4	1 199 999
NumberSoldNext	353 269	1.89	1.41	1	2	23
PriceRatioNow	353 269	1.06	0.24	0.0001	1.08	1.30
NextPriceRatio	353 269	1.05	0.25	0.0001	1.08	1.30
TimeLeftNow	353 269	1 328 948.00	3 242 160.00	10.49	289 361.90	38 291 142.00
TimeLeftWhenAvailable	353 269	1 392 132.00	3 258 664.00	10.49	342 142.70	38 291 142.00
TimeLeftNext	353 269	1 148 602.00	3 117 550.00	0.00	180 645.20	38 291 142.00
TimeAvailableFor	353 269	243 529.30	1 067 321.00	0	0	38 291 142

Table 3.2: Summary statistics of numeric variables in the unweighted TicketSwap dataset

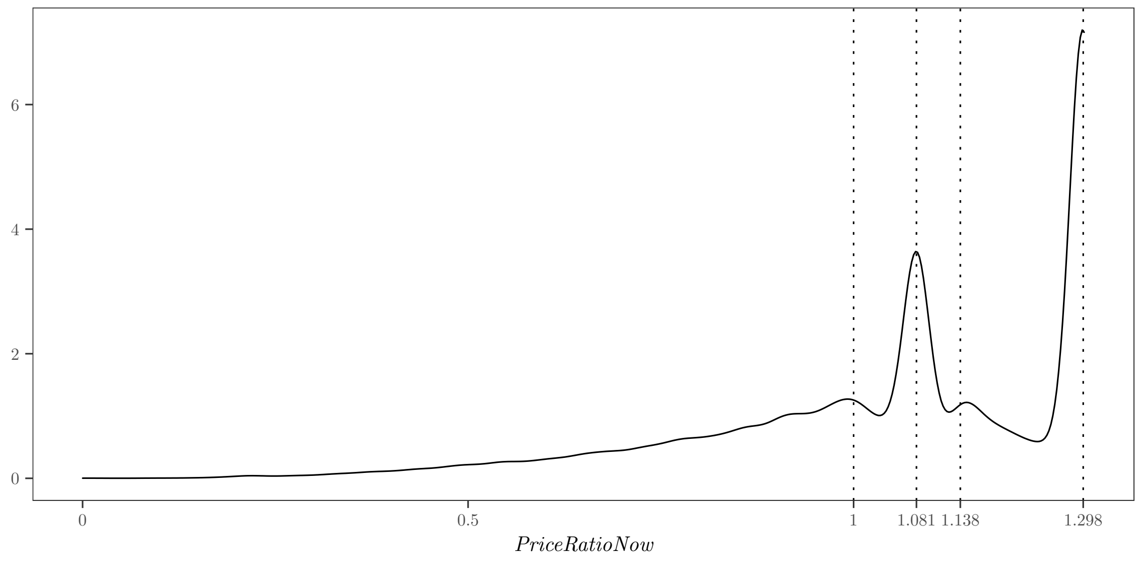


Figure 3.2: Kernel density estimates for *PriceRatioNow*'s distribution in the unweighted TicketSwap dataset.

3.3.1 The distribution of price ratios

The main questions we address in this paper regard the pricing of tickets. However, since the magnitude of prices varies from event to event (not to mention different currencies), we cannot conduct our analysis on the prices themselves. We first need to make them comparable somehow by defining a common scale. This is exactly the reason why we have created the *PriceRatioNow* variable (see table 3.1), which is the ratio of a ticket's total and original price. This *PriceRatioNow* is the key variable for our analysis.

In figure 3.2 we show the distribution of *PriceRatioNow* in the unweighted dataset (using kernel density estimation). We can see that this distribution is multimodal: the density function has four local maxima. Out of these four, the most popular price ratio is the platform's upper limit itself, 1.2978 (see section 3.1). This is not at all surprising since sellers who would like to set an even higher price ratio are bound by this constraint, and hence their best option is to just choose the maximum. What is more, we have shown in figure 3.1b that this upper limit is one of the default options that appear on a seller's screen when she is setting her price. Due to the status quo bias (Samuelson & Zeckhauser, 1988), this feature may push even more sellers to choose the maximal price.

The other three modes of the distribution are 1, 1.0815 and (approximately) 1.1384. These price ratios all correspond to cases where the sellers wanted to make a certain kind of price equal to the ticket's original price (*OriginalPrice*). In particular, a price ratio of 1 means that *TotalPrice*=*OriginalPrice*, a price ratio of 1.0815 means that *SellerPrice*=*OriginalPrice*, and a price ratio of $\frac{1.0815}{0.95} \approx 1.1384$ means that the price that the seller gets back after the transaction is equal to *OriginalPrice*. TicketSwap's user interface (see figure 3.1b) makes it convenient for sellers to set these three price ratios, especially in the case of 1.0815 (which is the default option). But thanks to the 'You'll receive' and 'Price on TicketSwap' fields, a seller can also quite easily set the price ratio to 1.1384 and 1 (respectively). She just has to make sure that these fields display the ticket's original price.

Why sellers choose these three price ratios so often is an important question. The most obvious explanations come from the field of behavioural economics. Each one of the three popular choices can be attributed to some kind of cognitive bias. We discuss these one by one.

First of all, the cases where the price ratio is 1 can be explained by inequity aversion (Fehr & Schmidt, 1999). Sellers may see it 'unfair' to offer their tickets to buyers for more than the original price on the primary market. This may especially be true because buyers and sellers on a secondary ticket market usually come from similar backgrounds (e.g. they are the fans of the same band). In such cases, social preferences are more likely to occur (Chen & Li, 2009) because agents may commit the so-called in-group bias (Brewer, 1979). Nevertheless, sellers may also have completely rational reasons for setting the price ratio to 1. If the event has not been sold out yet[9], no buyer will be willing to buy a ticket for more than its price on the primary market. In such cases it is rational for a seller to offer her ticket to buyers for the same price she has bought it for (or an infinitesimally lower one). We will see in subsection 5.3.1 that even artificial agents from our simulations exhibit this behaviour.

Since a price ratio of 1.0815 is the default option, the obvious explanation for its popularity is the status quo bias (Samuelson & Zeckhauser, 1988). Sellers may stick to this default option because this way they do not have to make calculations and then type in a number, they just click on 'Next' (see figure 3.1b) and their listing is ready. However, we can also think of another possible explanation. Sellers who choose this price ratio may be misinterpreting the rules of TicketSwap. They might mistakenly believe that this so-called 'selling price' is what they will get (or what buyers will pay), and not just a theoretical price. Although the rules are stated on the webpage (see figure 3.1b), they are described in a rather confusing manner. The term 'selling price' is itself misleading. Hence there may be some users who do not 'waste' their time and mental capacity on thoroughly reading and interpreting the rules.

Finally, sellers who set the price ratio to 1.1384 probably want to get back the price they have paid on the primary market. This behaviour can be explained in a prospect-theoretical model (Kahneman & Tversky, 1979; Tversky & Kahneman, 1992) where sellers commit the sunk cost fallacy (Arkes & Blumer, 1985). If a seller does not realise that the price she has paid for the tickets is a sunk cost, her reference point[10] will be her state of wealth before buying them. From this perspective, any price ratio below 1.1384 would be seen as a relative loss. Therefore, if she is loss averse, she will never set a price ratio below 1.1384.

3.3.2 Price ratios over time

The next aspect of the unweighted dataset we investigate is how the price ratios change as the event gets closer in time. Figure 3.3 shows a scatterplot of the *PriceRatioNow* and *TimeLeftNow* variables[11]. This figure is not that suitable for analytical purposes as some tickets in our dataset were uploaded very early (more than an entire year before the event), while most tickets were uploaded in the last days. However, even from this figure we can see that as the event gets closer, the volatility of price ratios increases. The tickets uploaded early tend to have high price ratios, but as time goes by, lower-priced tickets also start occurring. As a consequence, the mean price ratio seems to decrease with time, which is consistent with the findings of Sweeting (2012). Another easily noticeable aspect in the figure is that the dots form two clear horizontal lines. These lines are at the two most popular price ratios (1.0815 and 1.2978) we have dealt with in subsection 3.3.1.

The colours in figure 3.3 show the value of *EverSold* for each ticket. We can see that the dots are mostly blue, indicating that most of the uploaded tickets finally ended up being sold. Indeed, out of the 353269 tickets in our dataset, only 45901 remained unsold.

To obtain a figure that is easier to interpret, we ignore the tickets uploaded very early and 'zoom in' to the last weeks, where most of the listings are created. In particular, in figure 3.4 we show how the price ratios change in the last 5 million seconds (roughly two months) before the events. This figure gives us a much clearer picture. We can see that as time passes, the price ratios are indeed getting lower (and

[9]I.e. tickets for it can still be bought from the primary market.

[10]In prospect theory, an agent's reference point is the state of wealth to which she compares her possible payoffs.

[11]The latter is used with a minus sign because as time passes, the time left until the event becomes shorter.

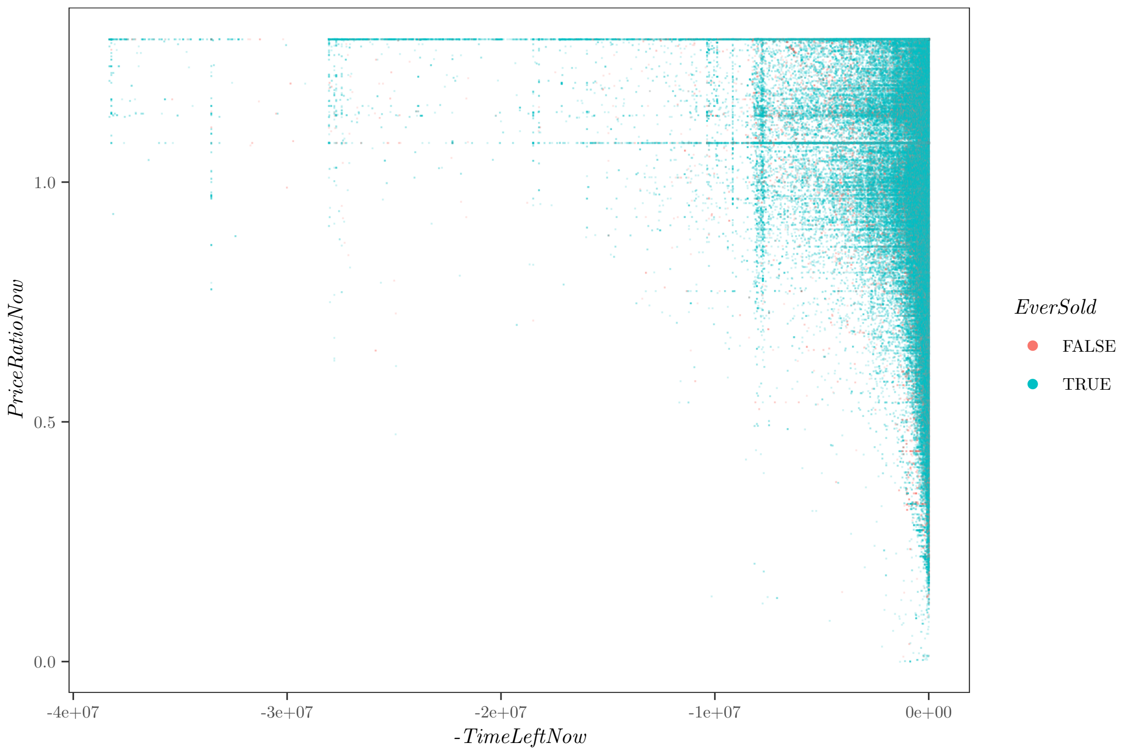


Figure 3.3: Scatterplot of *PriceRatioNow* and -*TimeLeftNow* in the unweighted TicketSwap dataset. Colours indicate whether the ticket was later sold or not.

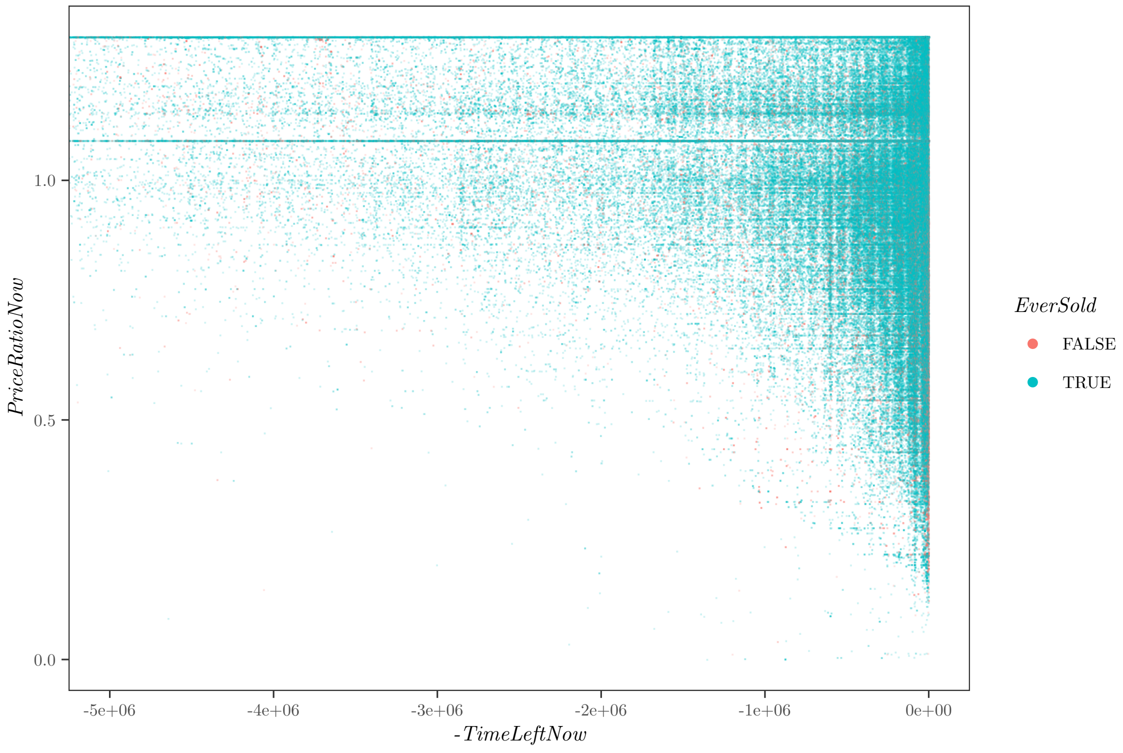


Figure 3.4: Scatterplot of *PriceRatioNow* and -*TimeLeftNow* in the unweighted TicketSwap dataset, in the last 5 million seconds. Colours indicate whether the ticket was later sold or not.

more volatile) on average. We can also clearly see the two horizontal lines at 1.0815 and 1.2978. What is more, even the other two modes of the distribution (1 and 1.1384, see figure 3.2) become noticeable.

Another important aspect of figure 3.4 is that the frequency of ticket uploads appears to be fluctuating. We can clearly see that the dots form vertical 'stripes' in the plot: denser areas are followed by sparser ones, and vice versa. This is due to the daily seasonality in the data. Not so surprisingly, most people upload their tickets at daytime. This causes fluctuation in the *TimeRecordedTicket* variable. But since most events happen in the evening, this fluctuation appears in the *TimeLeftNow* variable as well.

3.3.3 Regressions

In this subsection we run several regressions to see which factors determine the value of the *PriceRatioNow* variable. We consider seven specifications, starting from the simplest possible model and then increasing the complexity. Table 3.3 includes the results of these regressions.

Model (1) is a simple linear OLS regression, with only *TimeLeftNow* (and the constant) as explanatory variables. Based on what we have seen in figures 3.3 and 3.4, we expect the coefficient of *TimeLeftNow* to be significantly positive. It indeed is, implying that the price ratios are on average higher when the event is far from the current date.

However, figures 3.3 and 3.4 suggest that our data is clearly heteroscedastic. The variance of price ratios appears to be much higher when the *TimeLeftNow* variable is small. If we check this hypothesis using a Breusch-Pagan test (Breusch & Pagan, 1979) on model (1), the p-value we get is almost 0. This means that the standard errors from the first regression are biased estimates. Therefore, in the other six regressions we use heteroscedasticity-consistent (HC) standard errors. In the second column we can see that the results from model (1) are not really affected by this robust estimation. The coefficient of *TimeLeftNow* stays significant in model (2) as well.

Next, we include *NumberOfAvailable*, *NumberOfSold* and *NumberOfAlerts* as additional explanatory variables. The intuition behind this is that they may provide significant information about the ticket market to the seller when she is setting her price. In particular, we expect *NumberOfAvailable* to have a negative effect on the prices, as this variable captures the size of the supply side on the market. If sellers need to compete with more other sellers, they will set smaller prices. Similarly, the *NumberOfAlerts* variable is expected to have a positive effect, because it captures the demand's magnitude. The case of *NumberOfSold* is ambiguous. In theory, it should not affect the prices because it is just a measure of the market's intensity *in the past*. However, if the intensity is constant over time, this variable can also work as a proxy for the current demand and/or supply on the market. Hence it may affect the price ratios in both directions. The results of the model (3) appear to be in line with our intuitions. The coefficients of *NumberOfAvailable* and *NumberOfAlerts* are both significant, and their signs are also as we expected. *NumberOfSold* appears to have a positive effect on the prices, which would mean that it proxies the demand more than the supply. Nevertheless, we will see that this changes in the next regressions.

In the above three models we did not control for unobserved heterogeneity between event types. However, the level of price ratios may differ from event to event. Therefore, in the next regressions we include fixed effects for each *TypeID*. We can see that this increases the explanatory power of our models substantially. The adjusted R^2 from model (3) more than doubles when we use fixed effects in model (4).

Comparing the results of the third and fourth models, we can see that the sign of *NumberOfSold*'s coefficient is different. Since the fixed effect regression is clearly a more accurate model, this means that the unobserved heterogeneity between event types leads to a biased estimate. The real *ceteris paribus* effect of *NumberOfSold* is in fact negative. However, when we look at this issue from an economist's perspective, including *NumberOfSold* in the regression is itself rather questionable. We have argued that

	PriceRatioNow						
	(1)	(2)	(3)	(4)	(5)	(6)	(7)
Constant	1.0***	1.0***	1.1***				
	(0.0004)	(0.0004)	(0.0005)				
TimeLeftNow	$1 \cdot 10^{-8}$***	$1 \cdot 10^{-8}$***	$8 \cdot 10^{-9}$***	$3 \cdot 10^{-8}$***	$3 \cdot 10^{-8}$***	$1 \cdot 10^{-8}$***	$2 \cdot 10^{-8}$***
	$(1 \cdot 10^{-10})$	$(1 \cdot 10^{-10})$	$(1 \cdot 10^{-10})$	$(3 \cdot 10^{-10})$	$(2 \cdot 10^{-10})$	$(7 \cdot 10^{-10})$	$(7 \cdot 10^{-10})$
NumberOfAvailable			−0.0008***	−0.0007***	−0.0007***	−0.0005***	−0.0005***
			$(3 \cdot 10^{-6})$	$(5 \cdot 10^{-6})$	$(5 \cdot 10^{-6})$	$(6 \cdot 10^{-6})$	$(6 \cdot 10^{-6})$
NumberOfSold			$4 \cdot 10^{-5}$***	$-7 \cdot 10^{-6}$***		$-1 \cdot 10^{-5}$***	
			$(3 \cdot 10^{-7})$	$(5 \cdot 10^{-7})$		$(8 \cdot 10^{-7})$	
NumberOfAlerts			$2 \cdot 10^{-5}$***	$5 \cdot 10^{-5}$***	$5 \cdot 10^{-5}$***	$3 \cdot 10^{-5}$***	$2 \cdot 10^{-5}$***
			$(3 \cdot 10^{-7})$	$(7 \cdot 10^{-7})$	$(7 \cdot 10^{-7})$	$(2 \cdot 10^{-6})$	$(2 \cdot 10^{-6})$
SoldOut = ON_SALE						0.004	−0.003
						(0.004)	(0.004)
SoldOut = SOLD_OUT						0.04***	0.02***
						(0.003)	(0.003)
TypeID fixed effects	No	No	No	Yes	Yes	Yes	Yes
Number of *TypeID*s	–	–	–	3 833	3 833	155	155
Standard-Errors	IID	HC	HC	HC	HC	HC	HC
Observations	353 269	353 269	353 269	353 269	353 269	65 959	65 959
Size of the 'effective' sample	353 267	353 267	353 264	349 432	349 433	65 798	65 799
R^2	0.0354	0.0354	0.2664	0.5785	0.5782	0.4818	0.4800
Adjusted R^2	0.0354	0.0354	0.2664	0.5738	0.5736	0.4805	0.4787
Within R^2				0.1434	0.1429	0.1594	0.1565
F-test p-value	0	0	0	0	0	0	0

*Signif. Codes: ***: 0.01 **: 0.05 *: 0.1*

Table 3.3: Regression results for the unweighted TicketSwap dataset.

this variable could work as a proxy for the supply or demand on the market[12]. But the *NumberOfAvailable* and *NumberOfAlerts* variables already capture both the supply and the demand side in a much more straightforward way. It is unnecessary (and wrong) to include an additional proxy for *the same* mechanism that we already have variables for. Hence, even though its effect is statistically significant in model (4), for these economic reasons we exclude *NumberOfSold* from model (5). Fortunately, this does not change the estimates of the other coefficients substantially.

Finally, in models (6) and (7) we include the *SoldOut* variable, because we believe that the prices increase when the primary seller runs out of tickets. Model (6) is the statistically superior regression (where *NumberOfSold* is included), while model (7) is the economically superior one (where *NumberOfSold* is excluded). The problem with these regressions is that only a small portion of event pages are connected to the primary seller's webpage, so we have very few observations on *SoldOut*. Hence the external validity of these regressions is rather questionable. Nevertheless, we can see that the price ratios are indeed higher when the primary tickets are sold out (compared to the baseline category, *SoldOut* = DISABLED). On the other hand, the other dummy variable (indicating whether the tickets are on sale) is not significantly different from the baseline. We can also see that the sign and significance of other coefficients is unaffected: they are the same as in models (4) and (5).

Now that we have described each model, we should choose the one that has the best specification. As we have argued, the fixed effects are crucial to control for unobserved heterogeneity between event types, while including the *SoldOut* variable reduces the sample too much. And even though model (4) is slightly better than model (5) in the statistical sense, *NumberOfSold* should be excluded for the above-mentioned economic reasons. Hence we believe that model (5) has the best specification.

Nevertheless, which model we choose is almost completely indifferent. Fortunately, all seven specifica-

[12]The negative coefficient in model (4) implies that it proxies the supply side more. This is actually more intuitive than what model (3) implied. On most markets there are more buyers than sellers, and the number of transactions depends more on the smaller group's size.

tions are consistent in the main finding: as the event gets closer in time, the tickets become cheaper. This is the same as what Sweeting (2012) has found. We have also found that the number of available tickets (the supply) decreases the prices, while the number of ticket alerts (the demand) increases them. This is also consistently true in all regressions where we included these variables. The results for *NumberOfSold* are ambiguous, but we have argued that including this variable in the regressions does not make too much economic sense. Finally, we have found that the prices are higher when the tickets are sold out on the primary market. These results are all quite strong (significant on all usual levels), and in line with our intuition.

4 Model

In this chapter we present our own mathematical model of a certain event's ticket market. This model is designed to resemble the markets on TicketSwap as closely as possible. In section 4.1 we specify our model by describing the assumptions we make, then in section 4.2 we derive a certain probability which plays a key role in our model.

4.1 Assumptions

4.1.1 Arrivals in time

In our model we assume that time is continuous. Before the secondary market for tickets is formed[13], tickets can only be purchased from the primary seller (e.g. the organiser of a concert). We refer to the moment of the secondary market's formation as 'time zero'. At time zero, we assume that only the primary seller is present at the market. After time zero, buyers and secondary sellers start entering randomly. These arrivals at the market happen one-by-one, according to two independent Poisson processes. We denote the parameters of the two corresponding Poisson distributions by $\lambda \in \mathbb{R}_+$ for buyer arrivals and $\mu \in \mathbb{R}_+$ for (secondary) seller arrivals. This means that the probability of $k \in \mathbb{N}$ buyers arriving within a time interval of length $t \in \mathbb{R}_+$ is $\frac{(\lambda \cdot t)^k \cdot e^{-\lambda \cdot t}}{k!}$ (while for sellers the same is $\frac{(\mu \cdot t)^k \cdot e^{-\mu \cdot t}}{k!}$).

We assume that after the event itself occurs, the tickets for it become worthless for buyers, and therefore this market ceases to exist. We normalise the amount of time between time zero and the event to 1 unit, meaning that the market exists for exactly one unit of time. We of course define λ and μ in a way that corresponds to this scaling.

4.1.2 The primary seller

In this paper we are focusing on secondary ticket markets, so it may seem strange that the primary ticket seller is present in our model. However, distinguishing between the primary and secondary markets is mostly just a matter of definition. In real life these are so interrelated that ignoring the primary seller in a model would be a huge mistake. Instead, we are going to treat the market for the event's tickets as one entity which is not separable into two parts.

Nevertheless, we treat the primary seller's decision as exogenously given. We assume that it has already set a price ($p_p \in \mathbb{R}_+$) before time zero, and this price cannot be changed. We also know that the primary seller started selling its tickets earlier, so at time zero it only has a limited amount of available tickets left. We denote this amount by $N_p \in \mathbb{N}$.

4.1.3 The behaviour of buyers

We assume that each buyer arriving at this market would like to buy exactly one ticket for the event. We model their decision very simply: we assume that they have a reservation price $p_r \in \mathbb{R}_+$, and that whenever they see an available ticket for less than that (either at the primary seller or on the secondary market), they instantly buy it. They are not strategic thinkers in the sense that they do not wait for cheaper tickets to come. However, they still want to minimise their expenses, so if there are more than one available tickets for less than their reservation price, they always choose the cheapest option[14].

[13] At TicketSwap this happens when a user creates a page for it.

[14] And if several tickets have exactly the same price, they pick one randomly.

Just like at TicketSwap, we allow arriving buyers to subscribe for alerts when a new ticket becomes available. What is more, we assume that any buyer who enters the market instantly subscribes if no currently available ticket is acceptable to her. From our model's perspective this means that a buyer will always be aware of recently arrived secondary sellers until she actually decides to buy a ticket. (She is of course also aware of the price at the primary seller.) Therefore it can happen that when a new ticket becomes available, there are several subscribed buyers who would want to buy it (but only one can). We assume that in these situations nature picks one of the potential buyers randomly, and she will be the lucky one to get the ticket. This is basically the case on TicketSwap as well, where the ticket goes to the fastest buyer to click on the notification on her phone. Due to the harsh competition between buyers, sometimes even milliseconds make a difference, so we can reasonably argue that this is just a matter of pure luck.

We assume that the reservation prices of potential buyers in the whole population follow a certain distribution with cumulative distribution function[15] F, and that the reservation price of a buyer is independent from her arrival time. In our model's context, this is equivalent to the assumption that when a buyer arrives, her reservation price is randomly drawn from F.

4.1.4 The behaviour of secondary sellers

We assume that each secondary seller has exactly one ticket she wants to sell. In the moment of her arrival at the market, she has to set the price (p) she would like to offer the ticket for. Like at TicketSwap, this price must fall between two limits: $p \in [\underline{p}, \overline{p}]$. In our model these limits ($\underline{p}, \overline{p} \in \mathbb{R}_+$, $\underline{p} < \overline{p}$) have been set exogenously by the secondary ticket platform. We assume that once a seller has chosen a price, she cannot change it later, nor can she take the ticket off the platform. She cannot postpone this decision either, she has to submit the price right at the moment of her arrival.

Just like at TicketSwap, in our model we assume that secondary sellers do not get the total price that a buyer pays for their ticket. The platform collects a given ratio from the price of any transaction that happens on the secondary market. We assume this ratio to be exogenously given (or previously set by the platform), and denote it by $\tau \in [0, 1]$. Therefore, if a buyer buys a ticket for price $p \in \mathbb{R}_+$ on the secondary market, its seller only gets $(1 - \tau) \cdot p$ in return.

We assume that secondary sellers are risk averse, and that their preferences over lotteries can be represented by an expected utility function of the von Neumann–Morgenstern form (von Neumann & Morgenstern, 1944/1947). More precisely, we assume that for two possible payoffs x_1 and x_2 with corresponding probabilities P_1 and P_2, the expected utility of seller i is:

$$P_1 \cdot {x_1}^{\alpha_i} + P_2 \cdot {x_2}^{\alpha_i} \tag{4.1}$$

Here $\alpha_i \in [0, 1]$ is a given parameter measuring how risk averse the ith seller is. This may of course differ from seller to seller. Like the reservation prices of buyers, these α parameters also follows a certain distribution (we denote its CDF by G) in the population of sellers, and this distribution is also independent from the sellers' arrival times. In the model's context this means that when a secondary seller enters the market, her α is drawn randomly from G.

In our model a secondary seller perceives her possible payoffs relative to the situation where she fails to sell her ticket. In a prospect-theoretical context (Kahneman & Tversky, 1979; Tversky & Kahneman, 1992), this can be seen as her reference point. Strictly speaking, our model is not a prospect-theoretical one since we do not include the weighting function from the specification of Tversky and Kahneman

[15]In fact, this F function is closely related to the aggregate demand for the tickets. If we normalise the mass of buyers to one unit, the aggregate demand function is $D(p) = 1 - F(p)$.

(1992), nor do we assume loss aversion[16]. However, it may be interesting to apply the prospect-theoretical framework to our model and see how this changes the results. Maybe in that case it would be even better to assume that sellers compare their payoffs to the price they originally paid for the ticket (at the primary seller). This would be a more natural reference point, especially if sellers commit the sunk cost fallacy (Arkes & Blumer, 1985) by not realising that the price they paid for their ticket is already gone. This is a good direction for further research, but in this paper we stick to expected utility theory.

If the secondary seller perceives her payoffs this way, her payoff will by definition be zero when her ticket remains unsold. Similarly, when she succeeds in selling it, her payoff will be its price minus the share paid to the platform. Hence if she decides to offer her ticket to buyers for a price of $p \in \mathbb{R}_+$, her expected utility becomes:

$$P(\text{Ticket is not sold for price } p) \cdot 0^{\alpha_i} + P(\text{Ticket is sold for price } p) \cdot ((1-\tau) \cdot p)^{\alpha_i} \tag{4.2}$$

The first term in this sum is zero, so it can be left out. Secondary sellers maximise this expected utility by setting their price p. Therefore seller i's utility maximisation problem can be written as:

$$\max_{p \in [\underline{p}, \overline{p}]} \quad P(\text{Ticket is sold for price } p) \cdot ((1-\tau) \cdot p)^{\alpha_i} \tag{4.3}$$

We assume that the values of τ, $\underline{p}$, $\overline{p}$, p_p, N_p and λ are common knowledge among sellers, and so is the F function. At the moment of their arrival sellers also learn the current number of buyers who are subscribed for ticket alerts, and the prices of all currently available tickets. On the other hand, a buyer's reservation price (p_r) is private information, as well as the risk aversion parameter (α) of a seller.

Note that in the above list of commonly known values we did not include the μ parameter and the G function. This is because we assume that when a secondary seller makes her decision, she does not consider the possibility of other sellers entering the market *after* her own arrival. She only takes the prices of sellers already present at the market into account. In other words, she mistakenly believes that $\mu = 0$ and hence no more seller arrivals will occur. This means that we should rewrite the ith secondary seller's utility maximisation problem from (4.3) in a more precise way:

$$\max_{p \in [\underline{p}, \overline{p}]} \quad P(\text{Ticket is sold for price } p \mid 0 \text{ sellers arrive}) \cdot ((1-\tau) \cdot p)^{\alpha_i} \tag{4.4}$$

4.2 The probability from the utility maximisation problem

In the previous section we have sufficiently described our model, so ideally we could now turn to our simulations based on it. However, in order to do so, we first need to obtain a closed-form formula for the probability in (4.4). This is what we set out to do in this section.

How we can obtain this probability depends on the relation between the current seller's chosen price (p) and the prices of other currently available tickets on the (primary or secondary) market. Let us denote these other prices by $\{p_1, p_2, ..., p_N\}$, where $N \in \mathbb{N}$ can of course be zero if there are no tickets available at the moment of our seller's arrival. Without loss of generality, we assume that the indices indicate the order of these prices[17], i.e. $p_1 \leq p_2 \leq ... \leq p_N$.

We need to consider three cases. Our seller can either undercut all other sellers ($p < p_1$), set a unique price above the lowest one ($p > p_1$ and $p \notin \{p_1, p_2, ..., p_N\}$), or set a price that is exactly the same as one of the previous prices ($p \in \{p_1, p_2, ..., p_N\}$). We consider these possibilities one-by-one.

[16]Under these assumptions we do not even have to deal with losses since the worst thing that can happen to a seller is having zero payoffs.

[17]Some prices may of course be equal to each other (e.g. on the primary market they all are). These ties are just broken randomly in the indexing process.

4.2.1 Case I: $p < p_1$

We first deal with the case when our seller undercuts every other seller. We have assumed that buyers choose the ticket with the lowest acceptable price, which is in this case p. Since our seller believes that no more sellers will arrive after her, she must also think that her ticket will remain the cheapest for as long as it is on the market. This means that if at least one buyer arrives whose reservation price is above p, the ticket will be sold. What is more, it will also be sold if at the moment of our seller's arrival there is a subscribed buyer on the market with a reservation price above p. It is therefore easier for us to obtain the probability of the complementary event, $P(\text{Ticket is not sold for price } p \mid 0 \text{ sellers arrive})$. This can only happen if all buyers who arrive later than our seller, and all subscribed buyers already present at the market have a reservation price below p.

The probability of one reservation price falling below p is by definition $F(p)$. Suppose we have $k \in \mathbb{N}$ new buyers entering the market after our seller's arrival. The probability that all of their reservation prices fall below p is therefore $(F(p))^k$.

The case of previously subscribed buyers is a bit different since here the seller has additional information about them, and she can use this information to update her beliefs. The seller knows that these buyers entered the market earlier, so a ticket for a price of p_1 must have been available for them. But they did not buy it, which means that their reservation prices must all be smaller than p_1. Therefore, our seller can use Bayes' theorem (Bayes & Price, 1763) to obtain a conditional probability instead of an unconditional one. This updated probability of one previously subscribed buyer's reservation price (p_r) falling below p is:

$$P(p_r < p \mid p_r < p_1) = \frac{P(p_r < p)}{P(p_r < p_1)} = \frac{F(p)}{F(p_1)} \tag{4.5}$$

Let us denote the number of subscribed buyers at the moment of our seller's arrival by $n \in \mathbb{N}$. To obtain the updated probability of all previously subscribed buyers having a reservation price below p, we simply have to raise the expression in (4.5) to the nth power. Putting this together with the buyers arriving later, the probability of our seller's ticket not being sold for p (conditional on k buyers and zero sellers arriving) becomes:

$$P(\text{Ticket is not sold for price } p \mid k \text{ buyers and 0 sellers arrive}) = (F(p))^k \cdot \left(\frac{F(p)}{F(p_1)}\right)^n \tag{4.6}$$

Let us denote the amount of time between our seller's arrival and the event by $t \in \mathbb{R}_+$. Since we assumed that buyers arrive according to a Poisson process with parameter λ, the probability of k arrivals happening in this time interval is $\frac{(\lambda \cdot t)^k \cdot e^{-\lambda \cdot t}}{k!}$ for any $k \in \mathbb{N}$. Hence we can use Bayes' theorem again[18]:

$$\begin{aligned}
&P(\text{Ticket is not sold for price } p \mid 0 \text{ sellers arrive}) = \\
&\quad = \sum_{k=0}^{\infty} P(k \text{ buyers arrive}) \cdot P(\text{Ticket is not sold for price } p \mid k \text{ buyers and 0 sellers arrive}) = \\
&\quad = \sum_{k=0}^{\infty} \frac{(\lambda \cdot t)^k \cdot e^{-\lambda \cdot t}}{k!} \cdot (F(p))^k \cdot \left(\frac{F(p)}{F(p_1)}\right)^n = \\
&\quad = e^{-\lambda \cdot t} \cdot \left(\frac{F(p)}{F(p_1)}\right)^n \cdot \sum_{k=0}^{\infty} \frac{(\lambda \cdot t \cdot F(p))^k}{k!} = \\
&\quad = e^{\lambda \cdot t \cdot F(p) - \lambda \cdot t} \cdot \left(\frac{F(p)}{F(p_1)}\right)^n
\end{aligned}$$

This is the probability of the complementary event. We can now subtract this from 1 to obtain the

[18]In the last line we also make use of the identity that $\sum_{k=0}^{\infty} \frac{x^k}{k!} = e^x \; \forall x \in \mathbb{R}$.

final formula for the probability in (4.4) when $p < p_1$:

$$P(\text{Ticket is sold for price } p \mid 0 \text{ sellers arrive}) = 1 - e^{\lambda \cdot t \cdot F(p) - \lambda \cdot t} \cdot \left(\frac{F(p)}{F(p_1)} \right)^n \tag{4.7}$$

4.2.2 Case II: $p > p_1$ and $p \notin \{p_1, p_2, ..., p_N\}$

The next case is when our seller's price is not equal to any of the previously submitted prices but is above the lowest one. Let us denote the number of transactions on the market after our seller's arrival by $m \in \mathbb{N}$, and the number of tickets currently available for a cheaper price than p by[19] $J \in \mathbb{N}$. Since buyers always prefer cheaper tickets, our seller's ticket will not be sold unless all of these J tickets are sold before it. Our seller also believes that no more sellers will arrive after her arrival, so she must think that her ticket will remain on the $J + 1$st place of the buyers' preference list for the rest of the market's existence. Hence if someone does buy it, that will necessarily be the $J + 1$st transaction happening on this market. What is more, the fact that *there is* a $J + 1$st transaction already implies that our seller's ticket is sold. In other words, our seller thinks that her ticket will be sold if and only if $m > J$.

We also know that previously subscribed buyers are not interested in buying any of these $J+1$ tickets because their reservation prices are all below p_1 (see the reasoning in subsection 4.2.1). Therefore, if some buyers do buy these tickets, they all have to arrive at the market *after* our seller's arrival. Suppose that the number of buyer arrivals after our seller's arrival is $k \in \mathbb{N}$. Together with our seller's belief that no other sellers will enter after her, the above reasoning also implies that no transactions will happen involving previously subscribed buyers. This gives her a constraint on m: $m \leq k$.

Like in the previous case, the complementary event's probability is easier to obtain, so we will focus on scenarios when our seller's ticket is not sold, i.e. $m \leq J$. When $k \leq J$, this must necessarily be the case (because together with $m \leq k$ this implies that $m \leq k \leq J$). Hence we know that:

$$P(\text{Ticket is not sold for price } p \mid k \leq J \text{ buyers and } 0 \text{ sellers arrive}) = 1 \tag{4.8}$$

Things get a bit more complicated when $k > J$. The order of the k buyer arrivals will also play a role. The cheapest ticket will be sold to the first buyer whose reservation price is above p_1. But before her, any amount of buyers with reservation prices below p_1 can arrive, no transaction will happen. Similarly, the second transaction will occur when a buyer arrives with a reservation price above p_2. But between the first and second transactions, any number of other buyers may arrive as long as their reservation prices are smaller than p_2. And so on, the mth transaction will occur when a buyer arrives with a reservation price above p_m. And even this buyer may be followed by as many additional buyers as we like, as long as their reservation prices remain below[20] p'_{m+1}, there will not be an $m + 1$st transaction. Therefore, if we have k buyer arrivals and m transactions happening after our seller's arrival, the reservation prices of these arriving buyers need to follow a very specific pattern. This pattern is shown in figure 4.1.

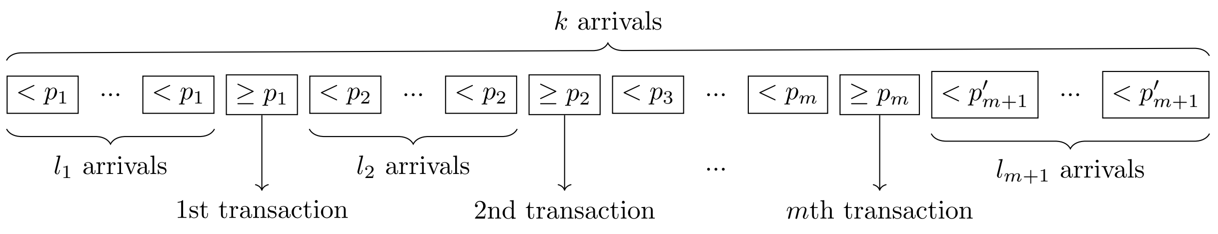


Figure 4.1: The reservation prices of the k buyers arriving after our seller (when m transactions happen).

[19] We of course know that $J \leq N$ and $p_J < p$ (and $p < p_{J+1}$ if $J \neq N$).

[20] We define this p'_{m+1} as being p when $m = J$, and p_{m+1} otherwise. We need this definition because when $m = J$, the $m + 1$st smallest price is our seller's price p (and not p_{J+1}).

We can see in figure 4.1 that we have denoted the number of buyer arrivals between our seller's arrival and the first transaction by $l_1 \in \mathbb{N}$, the number of buyer arrivals between the first and second transactions by $l_2 \in \mathbb{N}$, and so on. For given values of $\boldsymbol{l} = (l_1, l_2, ..., l_{m+1})$ and m, we can easily obtain the probability that the reservation prices follow the pattern of figure 4.1. We simply need to use the distribution of reservation prices F:

$$P(\text{Reservation prices follow the pattern above}) = F(p'_{m+1})^{l_{m+1}} \cdot \prod_{i=1}^{m} F(p_i)^{l_i} \cdot (1 - F(p_i)) \tag{4.9}$$

Since k arrivals happen in total, by summing up the elements of the $\boldsymbol{l}$ vector and adding them to the number of transactions m, the result needs to be k. (This is indicated in figure 4.1 as well.) By rearranging we get:

$$\sum_{i=1}^{m+1} l_i = k - m \tag{4.10}$$

In the terminology of combinatorics, the set of natural-valued $\boldsymbol{l}$ vectors that satisfy equation (4.10) is referred to as the set of weak $(m+1)$-compositions of $(k-m)$ (Hankin, 2006). Let us denote this set by C^{k-m}_{m+1}:

$$C^{k-m}_{m+1} = \left\{ \boldsymbol{l} \in \mathbb{N}^{m+1} \,\middle|\, \sum_{i=1}^{m+1} l_i = k - m \right\} \tag{4.11}$$

Using this definition we can obtain the same probability as in (4.9), but now for any $\boldsymbol{l} \in C^{k-m}_{m+1}$, treating only m and k as given. We have shown that this will be the probability that m transactions happen, conditional on $k > J$ buyer arrivals and 0 seller arrivals occurring after our seller's arrival.

$$\begin{aligned} &P(m \text{ transactions happen } | k > J \text{ buyers and 0 sellers arrive}) = \\ &\qquad = \sum_{\boldsymbol{l} \in C^{k-m}_{m+1}} F(p'_{m+1})^{l_{m+1}} \cdot \prod_{i=1}^{m} F(p_i)^{l_i} \cdot (1 - F(p_i)) \end{aligned}$$

We have shown that when zero sellers arrive, our seller's ticket will be unsold if and only if $m \leq J$. We can now obtain the probability of this event for a given k (above J):

$$\begin{aligned} &P(\text{Ticket is not sold for price } p \mid k > J \text{ buyers and 0 sellers arrive}) = \\ &\quad = P(\text{Less than } J \text{ transactions happen} \mid k > J \text{ buyers and 0 sellers arrive}) = \\ &\quad = \sum_{m=0}^{J} P(m \text{ transactions happen} \mid k > J \text{ buyers and 0 sellers arrive}) = \\ &\quad = \sum_{m=0}^{J} \sum_{\boldsymbol{l} \in C^{k-m}_{m+1}} F(p'_{m+1})^{l_{m+1}} \cdot \prod_{i=1}^{m} F(p_i)^{l_i} \cdot (1 - F(p_i)) \end{aligned} \tag{4.12}$$

Since we know that k follows a Poisson distribution, we can now use Bayes' theorem (Bayes & Price,

1763) on equations (4.8) and (4.12) to get:

$$
\begin{aligned}
&P(\text{Ticket is not sold for price } p \mid 0 \text{ sellers arrive}) = \\
&= \sum_{k=0}^{\infty} P(k \text{ buyers arrive}) \cdot P(\text{Ticket is not sold for price } p \mid k \text{ buyers and } 0 \text{ sellers arrive}) = \\
&= \left(\sum_{k=0}^{J} P(k \text{ buyers arrive}) \cdot P(\text{Ticket is not sold for price } p \mid k \text{ buyers and } 0 \text{ sellers arrive}) \right) + \\
&\quad + \sum_{k=J+1}^{\infty} P(k \text{ buyers arrive}) \cdot P(\text{Ticket is not sold for price } p \mid k \text{ buyers and } 0 \text{ sellers arrive}) = \\
&= \left(\sum_{k=0}^{J} \frac{(\lambda \cdot t)^k \cdot e^{-\lambda \cdot t}}{k!} \cdot 1 \right) + \\
&\quad + \sum_{k=J+1}^{\infty} \frac{(\lambda \cdot t)^k \cdot e^{-\lambda \cdot t}}{k!} \cdot \sum_{m=0}^{J} \sum_{\boldsymbol{l} \in C_{m+1}^{k-m}} F(p'_{m+1})^{l_{m+1}} \cdot \prod_{i=1}^{m} F(p_i)^{l_i} \cdot (1 - F(p_i))
\end{aligned}
$$

Now we simply need to subtract this from 1, and we get the desired probability for the case when $p > p_1$ and $p \notin \{p_1, p_2, ..., p_N\}$.

$$
\begin{aligned}
&P(\text{Ticket is sold for price } p \mid 0 \text{ sellers arrive}) = 1 - \left(\sum_{k=0}^{J} \frac{(\lambda \cdot t)^k \cdot e^{-\lambda \cdot t}}{k!} \right) - \\
&- \left(\sum_{k=J+1}^{\infty} \frac{(\lambda \cdot t)^k \cdot e^{-\lambda \cdot t}}{k!} \cdot \sum_{m=0}^{J} \sum_{\boldsymbol{l} \in C_{m+1}^{k-m}} F(p'_{m+1})^{l_{m+1}} \cdot \prod_{i=1}^{m} F(p_i)^{l_i} \cdot (1 - F(p_i)) \right)
\end{aligned}
\tag{4.13}
$$

4.2.3 Case III: $p \in \{p_1, p_2, ..., p_N\}$

The last case is when our seller sets a price that is exactly equal to some of the previously submitted prices. In particular, let us assume that $p_J < p = p_{J+1} = p_{J+2} = ... = p_{J+d}$ for some $J \in \{1, 2, ..., N\}$ and $d \in \{1, 2, ..., N - J\}$. In other words, there is a 'tie' of $d + 1$ prices (including p), and there are J other tickets which are cheaper than them.

This might be seen as the most complicated case since we need to handle this tie somehow. We have assumed that buyers are completely indifferent among equally-priced tickets, and so when they need to decide between such options, they just pick one randomly (with equal probabilities). This assumption is (in effect) equivalent to the one that nature breaks the tie by replacing the indifferences in the buyers' preference ordering with a randomly assigned order[21]. Therefore, in our case there is a $\frac{1}{d+1}$ chance that our seller's ticket will be $J + 1$st in the buyers' preference order, a $\frac{1}{d+1}$ chance that it will be $J + 2$nd, and so on, a $\frac{1}{d+1}$ chance that it will be $J + d + 1$st. Formally, we can write this as:

$$
P(\text{Ticket is } J + j\text{th}) = \frac{1}{d+1} \quad \forall j \in \{1, 2, ..., d + 1\} \tag{4.14}
$$

Note that once this random ordering has happened, the situation we find our seller in is equivalent to the previous case from subsection 4.2.2. There is a vector of previously submitted prices that buyers prefer to hers, only now this vector has $J + j$ elements instead of J, and the additional j tickets have the same price as our seller's. Therefore we can use the probability from (4.13), we only need to replace J with $J + j$, and our default indexing with the new preference ordering assigned by nature. Let us denote

[21]This order is not (necessarily) the same as the order of indices, which we also assigned randomly when ties occurred.

the price of the ith ticket in this new preference ordering by $\widehat{p_i}$. Hence we have:

$$P(\text{Ticket is sold for price } p \mid 0 \text{ sellers arrive and ticket is } J+j\text{th}) = 1 - \left(\sum_{k=0}^{J+j} \frac{(\lambda \cdot t)^k \cdot e^{-\lambda \cdot t}}{k!} \right) - \\ - \left(\sum_{k=J+j+1}^{\infty} \frac{(\lambda \cdot t)^k \cdot e^{-\lambda \cdot t}}{k!} \cdot \sum_{m=0}^{J+j} \sum_{\boldsymbol{l} \in C_{m+1}^{k-m}} F(\widehat{p_{m+1}})^{l_{m+1}} \cdot \prod_{i=1}^{m} F(\widehat{p_i})^{l_i} \cdot (1 - F(\widehat{p_i})) \right) \tag{4.15}$$

Now we can simply use Bayes' theorem (Bayes & Price, 1763) on equations (4.14) and (4.15) to get:

$$P(\text{Ticket is sold for price } p \mid 0 \text{ sellers arrive}) = \\ = \sum_{j=1}^{d+1} P(\text{Ticket is } J+j\text{th}) \cdot P(\text{Ticket is sold for price } p \mid 0 \text{ sellers arrive and ticket is } J+j\text{th}) = \\ = \sum_{j=1}^{d+1} \frac{1}{d+1} \cdot \left(1 - \left(\sum_{k=0}^{J+j} \frac{(\lambda \cdot t)^k \cdot e^{-\lambda \cdot t}}{k!} \right) - \right. \\ \left. - \left(\sum_{k=J+j+1}^{\infty} \frac{(\lambda \cdot t)^k \cdot e^{-\lambda \cdot t}}{k!} \cdot \sum_{m=0}^{J+j} \sum_{\boldsymbol{l} \in C_{m+1}^{k-m}} F(\widehat{p_{m+1}})^{l_{m+1}} \cdot \prod_{i=1}^{m} F(\widehat{p_i})^{l_i} \cdot (1 - F(\widehat{p_i})) \right) \right) = \\ = 1 - \frac{1}{d+1} \cdot \left(\sum_{j=1}^{d+1} \sum_{k=0}^{J+j} \frac{(\lambda \cdot t)^k \cdot e^{-\lambda \cdot t}}{k!} \right) - \\ - \frac{1}{d+1} \cdot \left(\sum_{j=1}^{d+1} \sum_{k=J+j+1}^{\infty} \frac{(\lambda \cdot t)^k \cdot e^{-\lambda \cdot t}}{k!} \cdot \sum_{m=0}^{J+j} \sum_{\boldsymbol{l} \in C_{m+1}^{k-m}} F(\widehat{p_{m+1}})^{l_{m+1}} \cdot \prod_{i=1}^{m} F(\widehat{p_i})^{l_i} \cdot (1 - F(\widehat{p_i})) \right)$$

Therefore we have obtained the probability from problem (4.4) for the case when $p \in \{p_1, p_2, ..., p_N\}$:

$$P(\text{Ticket is sold for price } p \mid 0 \text{ sellers arrive}) = 1 - \frac{1}{d+1} \cdot \left(\sum_{j=1}^{d+1} \sum_{k=0}^{J+j} \frac{(\lambda \cdot t)^k \cdot e^{-\lambda \cdot t}}{k!} \right) - \\ - \frac{1}{d+1} \cdot \left(\sum_{j=1}^{d+1} \sum_{k=J+j+1}^{\infty} \frac{(\lambda \cdot t)^k \cdot e^{-\lambda \cdot t}}{k!} \cdot \sum_{m=0}^{J+j} \sum_{\boldsymbol{l} \in C_{m+1}^{k-m}} F(\widehat{p_{m+1}})^{l_{m+1}} \cdot \prod_{i=1}^{m} F(\widehat{p_i})^{l_i} \cdot (1 - F(\widehat{p_i})) \right) \tag{4.16}$$

5 Simulations

Now that we have described our model and obtained its key variable, we can finally turn to our simulations based on it. However, in chapter 4 we did not set the values of our model's parameters, nor did we define the two distribution functions. This was all fine in a theoretical model where one tries to be as general as possible. But if we want to run simulations based on this model, we need to be specific. Therefore, in section 5.1 we specify the parametrisation of our simulations. Then in section 5.2 we include the basic outline of the simulating algorithm (for a more detailed description, see our codes in appendix A.4). Finally, in section 5.3 we describe and analyse the two artificial datasets we obtain after running the simulations. The codes for this analysis are included in appendix A.5.

5.1 Parametrisation

First of all, we assume that the reservation prices (p_r) of buyers always follow a uniform distribution on the $[0,1]$ interval[22], i.e.:

$$F(x) = P(p_r < x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \in [0,1] \\ 1 & \text{if } x > 1 \end{cases} \tag{5.1}$$

This does not mean that the maximal reservation price is *nominally* one, it is just a normalisation. We define this maximal reservation price as one unit, and measure other prices relative to it.

Similarly, we assume that the risk aversion parameters (α) of sellers also follow a uniform distribution on $[0,1]$. This means that we define the G function as:

$$G(x) = P(\alpha < x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \in [0,1] \\ 1 & \text{if } x > 1 \end{cases} \tag{5.2}$$

To make our artificial datasets similar to the one we have scraped from Ticketswap, we simulate the ticket markets of several events. But of course different events have different characteristics. The frequency of buyer and seller arrivals, the initial number of primary tickets, and the price at the primary seller are all factors which may vary from event to event. Since we want our simulations to mirror this heterogeneity, we assume that each event has a unique set of parameters[23] λ, μ, p_p and N_p. These parameters are all generated randomly for each event, using independent (continuous or discrete) uniform distributions again. More precisely, we use the following assumptions:

$$P(\lambda < x) = \begin{cases} 0 & \text{if } x < 0 \\ \frac{x}{20} & \text{if } x \in [0,20] \\ 1 & \text{if } x > 1 \end{cases} \qquad P(\mu < x) = \begin{cases} 0 & \text{if } x < 0 \\ \frac{x}{15} & \text{if } x \in [0,15] \\ 1 & \text{if } x > 1 \end{cases}$$
$$P(p_p < x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \in [0,1] \\ 1 & \text{if } x > 1 \end{cases} \qquad P(N_p = x) = \begin{cases} \frac{1}{6} & \text{if } x \in \{0,1,2,3,4,5\} \\ \\ 0 & \text{otherwise} \end{cases} \tag{5.3}$$

[22]This also implies that we have a linear demand function, see footnote 15.

[23]Nevertheless, we assume no heterogeneity in the F and G functions. They always follow a uniform distribution, as described above in equations (5.1) and (5.2).

We still have three more parameters which we have not specified yet: τ, $\underline{p}$ and $\overline{p}$. These are set exogenously by the secondary ticket platform. To see how these parameters affect the outcome, we consider two different specifications. The first one is where these values are the same as at TicketSwap. We refer to this specification as the 'constrained' one since these rules may be seen as market constraints. In particular, the rules of TicketSwap (see section 3.1) imply that $\tau = 1 - \frac{0.95}{1.05 \cdot 1.03}$ and $\overline{p} = 1.2 \cdot 1.05 \cdot 1.03 \cdot p_p$. The lower price limit ($\underline{p}$) should be equal to $5 \cdot 1.05 \cdot 1.03 = 5.4075$ euros, but since we normalised the maximal reservation price to one unit, we do not know how much €5.4075 is on this scale. What is more, the maximal reservation price may differ from event to event, and hence the value of $\underline{p}$ will be heterogenous among events after the normalisation. Therefore we use another uniform distribution to randomise its values[24]:

$$P(\underline{p} < x) = \begin{cases} 0 & \text{if } x < 0 \\ \frac{x}{p_p} & \text{if } x \in [0, p_p] \\ 1 & \text{if } x > p_p \end{cases} \tag{5.4}$$

The second specification we consider is 'unconstrained'. Under this setting, there is no platform on the secondary market. Secondary sellers can freely set any price they want, and they will get the full price that buyers pay to them. In the model's context this means that $\tau = 0$, $\underline{p} = 0$, and[25] $\overline{p} \geq 1$.

5.2 The outline of the algorithm

Our simulations of one event's ticket market are carried out according to the following steps:

1. We randomise the values of λ, μ, p_p and N_p, using their distributions from equation (5.3).
2. We randomise the two Poisson processes for buyer and secondary seller arrivals, with parameters λ and μ (respectively). We record the time of each arrival.
3. We randomise the reservation price of each arriving buyer, using equation (5.1).
4. We randomise the risk aversion parameter of each arriving secondary seller, using equation (5.2).
5. We go through the agents (buyers and secondary sellers) one by one, in the order of their arrivals. Using the constrained specification of τ and $\overline{p}$ (and randomising $\underline{p}$ from (5.4)), we determine what each agent's decision would be on a constrained market:
   - If the current agent is a buyer, we check if any of the tickets currently available at the time of her arrival have a price below her reservation price. If there is no such ticket, nothing happens (the buyer just subscribes for ticket alerts). But if some tickets are acceptable to her, we randomly pick one of them and that ticket will be 'matched' (sold) to the buyer.
   - If the current agent is a secondary seller, we numerically solve her utility maximisation problem from (4.4) (using formulas (4.7), (4.13), and (4.16), together with the definitions of the variables in them). This way we obtain the seller's optimal price. Then we check if there are any previously subscribed buyers whose reservation prices are above this price. If there are, one of them is selected randomly and matched to the current seller. If not, the seller simply uploads her ticket to the platform.

[24]As can be seen from the equation, here we implicitly assumed that $\underline{p} \leq p_p$.

[25]Since the maximal reservation price is one unit, no buyer would buy a ticket for more than one unit. This also means that no seller would try to sell it for more, and hence an upper limit above one unit will never be binding. Saying that $\overline{p} \geq 1$ is basically equivalent to saying that $\overline{p} = \infty$.

6. We go through the same agents again and repeat the same process, only now under the unconstrained settings.

7. We record the outcomes of the constrained and unconstrained markets in two separate datasets.

We repeat this process several times, obtaining two artificial datasets (one with and one without constraints) of the very same events' ticket markets.

5.3 Description and analysis

Both of our artificial datasets contain the data of 10874 events' ticket markets. On these markets there are overall 110005 buyers, 81144 secondary sellers, and 27058 primary tickets[26]. These sum up to 218207 observations, which are stored in a single data frame (in both the constrained and unconstrained cases).

Table 5.1 includes the description of each variable in the two artificial datasets. Some of these variables are analogous to variables from our model (see chapter 4) or from the TicketSwap dataset (see table 3.1). These analogies are also shown in the table. We can also see that some variables are defined only for secondary sellers, while one is defined only for buyers.

In our analysis we want to focus on the pricing decision of secondary sellers, like we did in section 3.3. Therefore we drop the observations on buyers and primary tickets, and keep only the 81144 observations that correspond to secondary sellers. This way we obtain two filtered datasets, one with and one without constraints. Tables 5.2 and 5.3 (respectively) include their basic summary statistics (only for numeric variables). We conduct our analysis on these filtered datasets. Our main goal is to compare them with each other and the TicketSwap dataset, so we use the same approach as in section 3.3.

5.3.1 The distribution of price ratios

Like in the previous analysis (see section 3.3), our key variable is the price ratio. Already from tables 5.2 and 5.3 we can see that this variable is on average higher in the unconstrained case than in the constrained one. This is not at all surprising since once we remove the upper limit on the price ratios, some sellers will choose higher prices than that. However, the median (which is more robust than the mean) of *PriceRatio* is almost the same in the two datasets. Therefore, the large difference in the means can be attributed to some outliers. We can see that the maximum price ratio is very high, which also points to this direction. To get a clearer picture about the distribution of price ratios in the two artificial datasets, we use kernel density estimation again. The estimates are shown in figure 5.1 for both datasets (colour-coded). In the unconstrained case the distribution's 'tail' is 'cut down'[27] at a price ratio of 1.5.

In figure 5.1 we can see that while the price ratios all fall between 0 and 1.2978 in the constrained dataset, in the unconstrained case they follow a long-tailed distribution. This is in line with our expectations: since the price ratios have to fall between two limits in the constrained case, they are of course more concentrated inside these limits. This also explains why the red curve runs above the blue one in most cases. However, for small price ratios the opposite is true: the unconstrained dataset shows a higher density. This is probably because of the lower limit on the prices. The smaller the price ratio, the more likely it is that it will fall below the lower limit. As a consequence, we can also see that the red line is increasing rather steeply in the beginning.

The constrained dataset, which has been simulated using the rules of TicketSwap, appears to resemble the TicketSwap dataset in several aspects. Two of the four modes from figure 3.2 (shown here using dotted lines) also appear in figure 5.1. The first one is the upper limit itself, 1.2978. In subsection 3.3.1 we

[26]Tickets from the primary seller are stored in separate rows, even when they are for the same event.

[27]We have seen in table 5.3 that the maximum price ratio is huge in the unconstrained dataset, so a figure where the whole range is plotted would not be useful for analytical purposes.

		Analogies	
Variable	Description	Model	TicketSwap
Arrival	The amount of time between the arrivals of the current agent and the previous agent of the same kind		
TimeLeftNow	The amount of time between the current arrival and the event	t	*TimeLeftNow*
Buyer	Whether the current agent is a buyer or not		
*ReservationPrice***	The reservation price of the current buyer	p_r	
*alpha**	The risk aversion parameter of the current secondary seller	α	
Price	The total price of the ticket offered (if she is a seller) or bought (if she is a buyer) by the current agent	p	*TotalPrice*
*Objective**	The maximised expected utility of the current secondary seller		
Matched	Whether the current agent has been matched to any other agent in a transaction		*EverSold*
MatchedWith	The *ID* of the agent to whom the current agent is matched		
TimeLeftWhenMatched	The amount of time between the moment when this agent is matched and the event		*TimeLeftNext*
lambda	The rate of buyer arrivals	λ	
mu	The rate of secondary seller arrivals	μ	
NumberOfPrimaryTickets	The number of available tickets left on this event's primary market at time zero	N_p	
OriginalPrice	The original price of tickets on this event's primary market	p_p	*OriginalPrice*
LowerLimit	The lower price limit on this event's secondary ticket market	$\underline{p}$	
UpperLimit	The upper price limit on this event's secondary ticket market	$\min(\overline{p}, 1)$	
EventID	The unique ID of the current event (type)		*TypeID*
*NumberOfAvailable**	The current number of this event's available tickets	N	
*NumberOfMatched**	The number of transactions on this event's market between time zero and the current arrival		
*NumberOfAlerts**	The current number of buyers subscribed to this event's alerts	n	*NumberOfAlerts*
*AvgPrice**	The current average price of this event's available tickets		
*MinPrice**	The current minimum price of this event's available tickets		
*NumberOfAvailableSecondary**	The current number of this event's available tickets on the secondary market		*NumberOfAvailable*
*NumberOfMatchedSecondary**	The number of transactions on this event's secondary market between time zero and the current arrival		*NumberOfSold*
*AvgPriceSecondary**	The current average price of available tickets on this event's secondary market		
*MinPriceSecondary**	The current minimum price of available tickets on this event's secondary market		
ID	The unique ID of the current agent		
PriceRatio	The ratio of *Price* and *OriginalPrice*	$\frac{p}{p_p}$	*PriceRatioNow*
*AvgPriceRatio**	The current average price ratio of this event's available tickets		
*MinPriceRatio**	The current minimum price ratio of this event's available tickets		
*SoldOut**	Whether this event's tickets on the primary market are currently sold out or not		*SoldOut*
*AvgPriceRatioSecondary**	The current average price ratio of available tickets on this event's secondary market		
*MinPriceRatioSecondary**	The current minimum price ratio of available tickets on this event's secondary market		

*: Defined only for secondary sellers, **: Defined only for buyers

Table 5.1: The description of variables in the two artificial datasets

Statistic	N	Mean	St. Dev.	Min	Median	Max
Arrival	81 144	0.11	0.12	0.0000	0.07	1.00
TimeLeftNow	81 144	0.50	0.29	0.0000	0.50	1.00
alpha	81 144	0.50	0.29	0.0000	0.50	1.00
Price	81 144	0.40	0.23	0.0001	0.40	1.00
Objective	81 144	0.48	0.27	0.0000	0.50	1.00
TimeLeftWhenMatched	42 155	0.40	0.26	0.0000	0.38	1.00
lambda	81 144	10.13	5.81	0.001	10.30	20.00
mu	81 144	9.92	3.59	0.09	10.55	15.00
NumberOfPrimaryTickets	81 144	2.48	1.71	0	2	5
OriginalPrice	81 144	0.50	0.29	0.0001	0.50	1.00
LowerLimit	81 144	0.25	0.22	0.0000	0.18	0.99
UpperLimit	81 144	0.62	0.33	0	0.7	1
NumberOfAvailable	81 144	4.84	3.96	0	4	28
NumberOfMatched	81 144	2.62	2.88	0	2	27
NumberOfAlerts	81 144	2.43	3.13	0	1	28
AvgPrice	69 178	0.50	0.25	0.0001	0.52	1.00
MinPrice	69 178	0.45	0.24	0.0001	0.46	1.00
NumberOfAvailableSecondary	81 144	3.05	3.33	0	2	23
NumberOfMatchedSecondary	81 144	1.93	2.39	0	1	27
AvgPriceSecondary	58 435	0.48	0.24	0.0001	0.51	1.00
MinPriceSecondary	58 435	0.42	0.23	0.0001	0.43	1.00
PriceRatio	81 144	0.89	0.30	0.01	0.92	1.30
AvgPriceRatio	69 178	0.98	0.19	0.11	1.00	1.30
MinPriceRatio	69 178	0.90	0.26	0.02	0.99	1.30
AvgPriceRatioSecondary	58 435	0.98	0.21	0.11	0.98	1.30
MinPriceRatioSecondary	58 435	0.88	0.28	0.02	0.93	1.30

Table 5.2: Summary statistics of numeric variables in the constrained artificial dataset of secondary sellers

Statistic	N	Mean	St. Dev.	Min	Median	Max
Arrival	81 144	0.11	0.12	0.0000	0.07	1.00
TimeLeftNow	81 144	0.50	0.29	0.0000	0.50	1.00
alpha	81 144	0.50	0.29	0.0000	0.50	1.00
Price	81 144	0.45	0.22	0.0000	0.48	0.94
Objective	81 144	0.59	0.27	0.00	0.66	1.00
TimeLeftWhenMatched	42 839	0.37	0.25	0.0000	0.33	1.00
lambda	81 144	10.13	5.81	0.001	10.30	20.00
mu	81 144	9.92	3.59	0.09	10.55	15.00
NumberOfPrimaryTickets	81 144	2.48	1.71	0	2	5
OriginalPrice	81 144	0.50	0.29	0.0001	0.50	1.00
LowerLimit	81 144	0.00	0.00	0	0	0
UpperLimit	81 144	1.00	0.00	1	1	1
NumberOfAvailable	81 144	4.93	3.70	0	4	26
NumberOfMatched	81 144	2.53	2.60	0	2	18
NumberOfAlerts	81 144	2.53	3.21	0	1	28
AvgPrice	73 820	0.56	0.21	0.0001	0.60	1.00
MinPrice	73 820	0.47	0.24	0.0000	0.50	1.00
NumberOfAvailableSecondary	81 144	3.17	3.19	0	2	23
NumberOfMatchedSecondary	81 144	1.81	2.20	0	1	16
AvgPriceSecondary	63 107	0.56	0.20	0.0003	0.60	0.97
MinPriceSecondary	63 107	0.45	0.23	0.0000	0.49	0.87
PriceRatio	81 144	5.45	104.01	0.0000	0.91	6 930.64
AvgPriceRatio	73 820	5.66	113.16	0.001	1.00	6 346.49
MinPriceRatio	73 820	4.68	100.80	0.0000	1.00	6 045.25
AvgPriceRatioSecondary	63 107	6.46	122.37	0.001	1.00	6 346.49
MinPriceRatioSecondary	63 107	5.30	109.01	0.0000	0.96	6 045.25

Table 5.3: Summary statistics of numeric variables in the unconstrained artificial dataset of secondary sellers

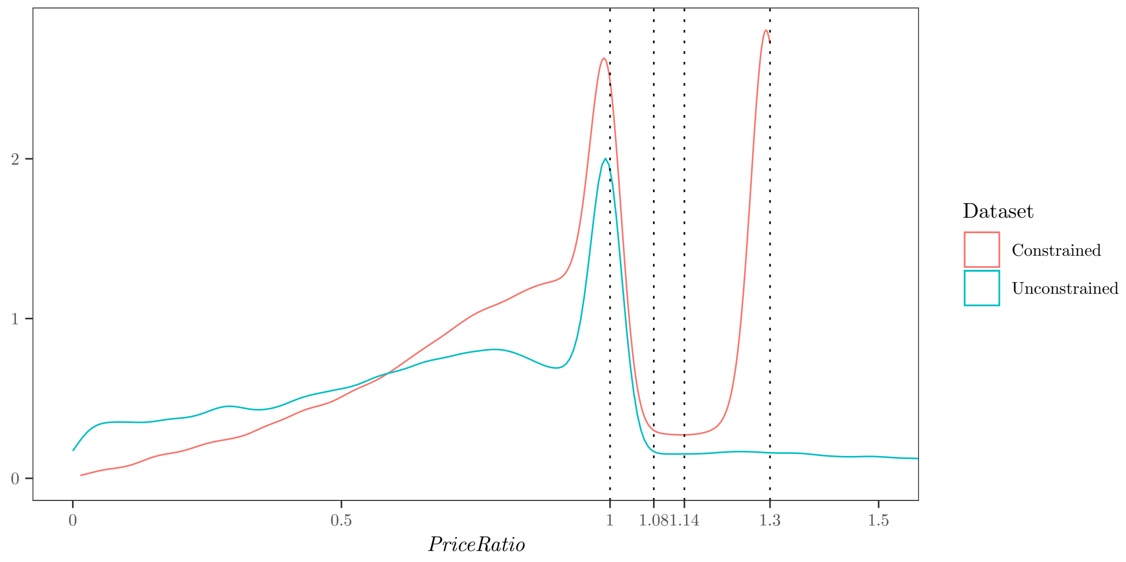


Figure 5.1: Kernel density estimates for *PriceRatio*'s distribution in the constrained (red) and unconstrained (blue) artificial datasets of secondary sellers, for price ratios below 1.5.

argued that the reason for this price ratio's popularity is that sellers who would like to set a higher price are bound by the upper limit. Translated to our model's context, this simply means that the utility maximisation problem from (4.4) yields a corner solution. Therefore the finding that the same logic applies to the simulated dataset as well is rather intuitive.

The 'red' distribution's other mode is at the price ratio of 1. This is most probably also the result of corner solutions. We have seen in section 4.2 that sellers need to optimise on several subintervals when maximising their expected utility. These subintervals are separated by the prices of currently available tickets. But since tickets on the primary market have the same price (p_p), at least two of the subintervals will always have p_p as an endpoint (as long as the primary tickets are not sold out). If the final optimum is a corner solution on this subinterval, then the price ratio will be approximately[28] 1.

Nevertheless, the remaining two modes of *PriceRatioNow*'s distribution in the TicketSwap dataset (1.0815 and 1.1384, see figure 3.2) do not appear in the constrained artificial dataset. In subsection 3.3.1 we could only give behavioural explanations to why these price ratios are popular. Therefore, since our model did not incorporate any of these behavioural aspects, it is not surprising that these modes do not show up here.

Now we turn to the distribution of price ratios in the unconstrained dataset. We can see in figure 5.1 that the mode of 1 appears here as well. The explanation we have given to this price ratio's popularity above also implies to this case (as it had nothing to do with the constraints). However, the figure also shows that the density at 1 is not as high as in the constrained dataset. This means that when we remove the constraints, less sellers will set this price ratio. Indeed, the constraints tighten the interval of price ratios that sellers can choose from, so price ratios 'in the middle' will become more popular in the constrained case. It is also noticeable from figure 5.1 that the 'blue' distribution has another mode, approximately at 0.8. But unlike in the previous cases, there is nothing special about this price ratio that would explain its relative popularity.

[28]When one of the previously submitted prices is p_p, a price of $p = p_p$ falls in case III (see subsection 4.2.3), but a price of $p = p_p \pm \varepsilon$ falls in case II (see subsection 4.2.2). Hence the objective function is not continuous at $p = p_p$, so the optimum may not exist. If this is the case, the seller might want to set an infinitesimally higher/lower price than p_p. What is more, if a previous secondary seller has set a price of $p_p \pm \varepsilon$, then the next secondary seller's optimal choice may be to set a price of $p_p \pm \varepsilon \pm \delta$, and so on. As a consequence, it can happen that the price ratios of consecutive sellers are always increasing/decreasing with an infinitesimally small amount. In fact, even figure 5.1 supports this hypothesis: the actual mode of the distribution appears to be slightly lower than 1.

Another important aspect of figure 5.1 is that the price ratio of 1 seems to work as some kind of cut-off in both distributions. There is a sudden drop in the density both before and after[29] the 'hump' at 1, and the level of density is much higher right before the 'hump' at 1 than right after it. This means that a price slightly above the original one is less likely to be chosen than a price slightly below it.

5.3.2 Price ratios over time

Now that we have described their distribution, we turn to how price ratios change over time (like we did in subsection 3.3.2 for the TicketSwap dataset). Figures 5.2 and 5.3 show a scatterplot of the *PriceRatio* and *TimeLeftNow* variables in the constrained and unconstrained datasets of secondary sellers (respectively). In the unconstrained case we 'zoomed in' to price ratios below 1.5 (like in figure 5.1), to avoid having to plot the outliers.

From figure 5.2 we can see that the constrained artificial dataset again behaves similarly to the TicketSwap dataset. Like in figures 3.3 and 3.4, the price ratios tend to decrease as the event gets closer in time. The horizontal lines also appear at the two modes of the distribution (see figure 5.1). Therefore, like the one from the paper of Sweeting (2012), our model also captures the negative trend in the prices. We can also see that the volatility of price ratios increases with time, though not as much as in the TicketSwap dataset. However, this simulated dataset also exhibits a 'new' aspect which the original one does not. The upper right corner of the figure looks like as if someone has taken a bite of it, indicating that when the event gets really close, sellers do not choose high price ratios anymore. We have not observed this behaviour in the TicketSwap dataset.

The other scatterplot, figure 5.3 shows a different picture. Since we do not have an upper limit on the price ratios in this case, here we do not see a horizontal line at 1.2978. However, as we could expect from figure 5.1, the other horizontal line (at the price ratio of 1) remains, and the dots are much less dense above this line than below it. Due to the fact that we have excluded the dots above 1.5 from the figure, we cannot say much about how the mean and the variance of price ratios changes over time. But based on what we see in this figure, the general finding that the mean decreases and the volatility increases seems to apply to this dataset as well.

We can see in both figures (5.2 and 5.3) that many dots are red, indicating that a significant ratio of the tickets remains unsold in the end. This is again an aspect that is different from what we see in the TicketSwap dataset (figures 3.3 and 3.4), where almost all dots were blue. Indeed, about half of the tickets uploaded by secondary sellers remain unsold in both artificial datasets, while in the TicketSwap dataset the same ratio is only about 13%. What is more, these red dots are more concentrated in the bottom of the figure, while in the upper regions we mostly see blue dots. This finding may seem rather strange and counter-intuitive at the first glance, as it implies that tickets with high price ratios end up sold more often. However, this is most probably just a result of reverse causality: sellers set lower prices *precisely because* they know that the current event is not popular. They try to increase their small chances of selling the tickets by decreasing their price.

5.3.3 Regressions

Now that we have seen the distribution of price ratios and how they change over time, what is left is to run some regressions. We consider the same seven specifications as in the analysis of the TicketSwap dataset (see table 3.3), using the analogous variables from the artificial datasets (see table 5.1). The regression results are included in table 5.4 for the constrained, and in table 5.5 for the unconstrained dataset.

[29]In the constrained case, the sudden drop after the 'hump' at 1 is followed by yet another hump because of the upper limit on the price ratios.

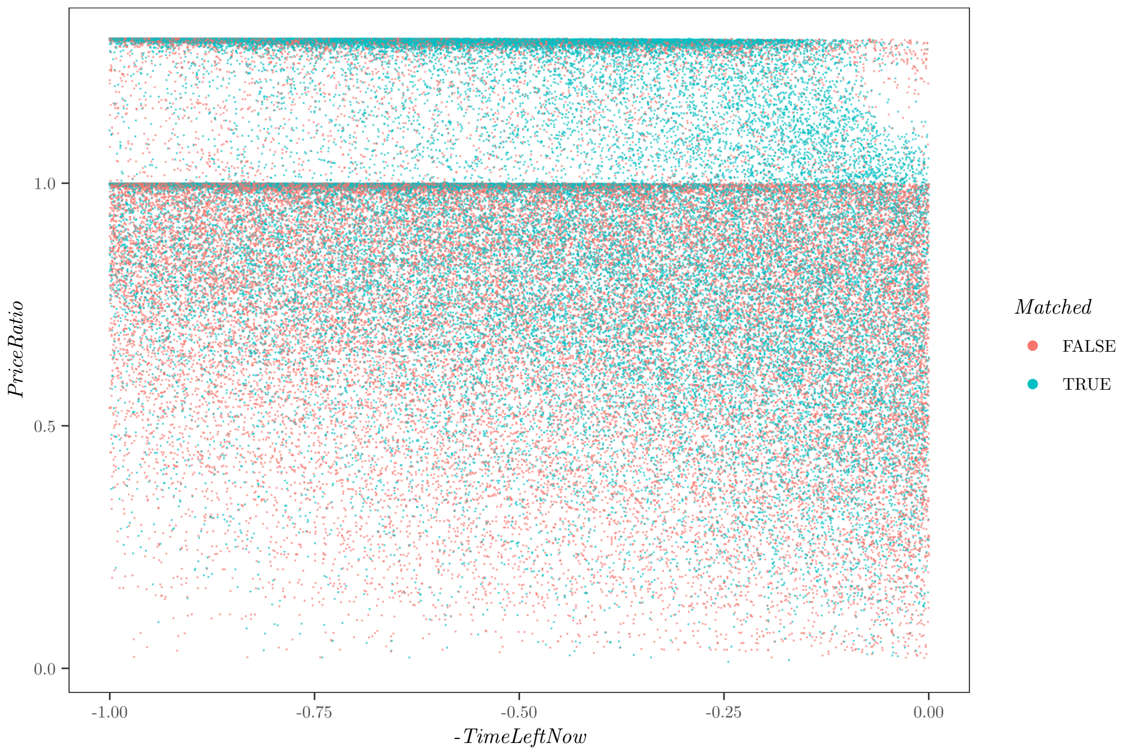


Figure 5.2: Scatterplot of *PriceRatio* and -*TimeLeftNow* in the constrained artificial dataset of secondary sellers. Colours indicate whether the ticket was later sold or not.

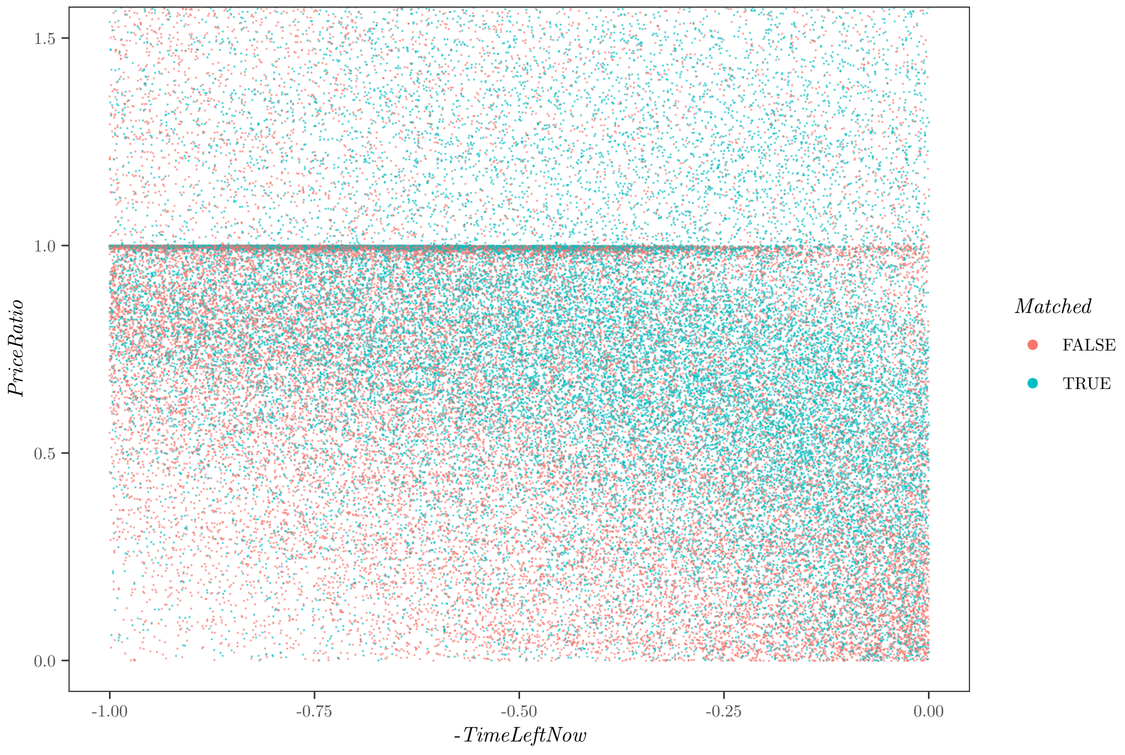


Figure 5.3: Scatterplot of *PriceRatio* and -*TimeLeftNow* in the unconstrained artificial dataset of secondary sellers, for price ratios below 1.5. Colours indicate whether the ticket was later sold or not.

	PriceRatio						
	(1)	(2)	(3)	(4)	(5)	(6)	(7)
Constant	0.7***	0.7***	0.7***				
	(0.002)	(0.002)	(0.004)				
TimeLeftNow	0.3***	0.3***	0.4***	0.2***	0.4***	0.2***	0.4***
	(0.003)	(0.003)	(0.005)	(0.005)	(0.004)	(0.005)	(0.004)
NumberOfAvailableSecondary			−0.02***	−0.007***	−0.002***	−0.007***	−0.001***
			(0.0003)	(0.0003)	(0.0003)	(0.0003)	(0.0003)
NumberOfMatchedSecondary			0.02***	−0.02***		−0.02***	
			(0.0005)	(0.0005)		(0.0005)	
NumberOfAlerts			0.01***	0.007***	0.007***	0.007***	0.007***
			(0.0003)	(0.0003)	(0.0003)	(0.0003)	(0.0003)
SoldOut = TRUE						0.06***	0.05***
						(0.003)	(0.003)
EventID fixed effects	No	No	No	Yes	Yes	Yes	Yes
Number of *EventID*s	–	–	–	10 182	10 182	10 182	10 182
Standard-Errors	IID	HC	HC	HC	HC	HC	HC
Observations	81 144	81 144	81 144	81 144	81 144	81 144	81 144
Size of the ‘effective’ sample	81 142	81 142	81 139	70 958	70 959	70 957	70 958
R^2	0.0950	0.0950	0.1855	0.8450	0.8411	0.8459	0.8420
Adjusted R^2	0.0950	0.0950	0.1854	0.8227	0.8183	0.8238	0.8193
Within R^2				0.3872	0.3720	0.3910	0.3755
F-test p-value	0	0	0	0	0	0	0

*Signif. Codes: ***: 0.01 **: 0.05 *: 0.1*

Table 5.4: Regression results for the constrained artificial dataset of secondary sellers.

	PriceRatio						
	(1)	(2)	(3)	(4)	(5)	(6)	(7)
Constant	3.9***	3.9***	−4.8**				
	(0.7)	(0.6)	(2.1)				
TimeLeftNow	3.1**	3.1**	11.5***	−2.8**	0.7	−3.4***	0.4
	(1.3)	(1.3)	(2.4)	(1.1)	(0.5)	(1.2)	(0.5)
NumberOfAvailableSecondary			0.6***	−0.6***	−0.4***	−0.6***	−0.4***
			(0.2)	(0.1)	(0.09)	(0.1)	(0.08)
NumberOfMatchedSecondary			0.5**	−0.5***		−0.6***	
			(0.2)	(0.1)		(0.1)	
NumberOfAlerts			0.8***	−0.07	−0.07	−0.02	−0.03
			(0.2)	(0.06)	(0.06)	(0.07)	(0.07)
SoldOut = TRUE						−2.5***	−2.2**
						(0.9)	(0.9)
EventID fixed effects	No	No	No	Yes	Yes	Yes	Yes
Number of *EventID*s	–	–	–	10 182	10 182	10 182	10 182
Standard-Errors	IID	HC	HC	HC	HC	HC	HC
Observations	81 144	81 144	81 144	81 144	81 144	81 144	81 144
Size of the ‘effective’ sample	81 142	81 142	81 139	70 958	70 959	70 957	70 958
R^2	0.0001	0.0001	0.0006	0.9538	0.9538	0.9538	0.9538
Adjusted R^2	0.0001	0.0001	0.0006	0.9471	0.9471	0.9472	0.9471
Within R^2				0.0033	0.0027	0.0036	0.0030
F-test p-value	0.0131	0.0131	0.0000	0	0	0	0

*Signif. Codes: ***: 0.01 **: 0.05 *: 0.1*

Table 5.5: Regression results for the unconstrained artificial dataset of secondary sellers.

We can see in table 5.4 that the constrained artificial dataset produces the same regression results as the TicketSwap dataset. The signs of the coefficients are exactly the same in all seven regressions as in their analogies from table 3.3. These results are significant on all the usual levels. However, the unconstrained dataset does not behave this similarly to the TicketSwap data. We can see in table 5.5 that even the main finding that the price ratios decrease with time becomes questionable when we include *EventID* fixed effects. And we can see ambiguous results for the other variables as well. What is more, none of the seven models could reproduce all of our general findings[30] from subsection 3.3.3. The reason for this may be that there are many outliers in the unconstrained dataset, as we have shown.

[30]By the term 'general finding' we refer to the results which were consistent in the seven models. These are the significantly positive coefficient for *TimeLeftNow*, *NumberOfAlerts* and the *SoldOut* dummy, and the significantly negative coefficient for *NumberOfAvailable(Secondary)*.

6 Results

In this chapter we draw the final conclusions from our paper's findings. First, in section 6.1 we argue that the constrained artificial dataset behaves very similarly to the TicketSwap dataset, and hence our model provides a sufficient description for the behaviour of secondary ticket sellers. Then in section 6.2 we use the two artificial datasets to see how the constraints affect social surplus. Finally, in section 6.3 we show what our results imply for TicketSwap.

6.1 Comparing the real and artificial datasets

When we compare our analyses of the TicketSwap dataset (section 3.3) and the constrained artificial dataset (section 5.3), we can see that their results are very similar in almost all cases. The two distributions of price ratios (in figures 3.2 and 5.1) are analogous in many aspects, even two of the modes are identical. The scatterplots of *TimeLeftNow* and *PriceRatio(Now)* (figures 3.3, 3.4 and 5.2) also show similar patterns. Finally, the signs of the regression coefficients (tables 3.3 and 5.4) are exactly the same in all seven specifications (and these are all significant results at 1%). These findings imply that our model (and the simulations based on it) have served their purpose: we were successful in capturing several aspects of secondary seller behaviour on TicketSwap.

However, there are of course some aspects in which the constrained artificial dataset differs from the one we have scraped from TicketSwap. For instance, we have seen that two modes of the distribution from figure 3.2 still do not appear in figure 5.1. This is because in our model we did not include the behavioural reasons behind the popularity of these price ratios. We could of course modify the model in a way that incorporates these aspects as well[31], but we believe that even this rational-agent model is sufficient for our purposes. Another worrying aspect is that the ratio of tickets remaining unsold is much higher in the artificial data. This is mainly the result of the parametrisation of our simulations[32] (see section 5.1). In section 7.3 we show how this might limit our research.

Although we should keep these limitations in mind, overall the analogies between the two datasets are convincing. We can never expect a model to perfectly reflect what we see in the empirical data, but this one appears to work quite well. Therefore, we can use our model and simulations to make additional conclusions about TicketSwap (or, even more generally, secondary ticket markets). In the next section, this is exactly what we do. Whether these conclusions indeed apply to real markets is of course still questionable, and we will never have clear evidence of that. All we can say is that based on the analyses in sections 3.3 and 5.3, we have good reasons to think that our findings can be generalised.

6.2 Welfare analysis

Our main research question in this paper is how the rules of TicketSwap affect social welfare. Using the scraped TicketSwap dataset only, we would not be able to answer this question. This dataset does not include any information about the utility that buyers derive from buying a ticket, so it cannot be used for welfare analysis. However, if we accept the above proposition that our simulations provide a sufficient description of secondary ticket markets, we can use them instead. The two artificial datasets contain the reservation price of each buyer, and that can be seen as a utility measure. Therefore, we can conduct our welfare analysis on these datasets and then draw the conclusions for TicketSwap.

[31]For example, a prospect-theoretical model (Kahneman & Tversky, 1979; Tversky & Kahneman, 1992) could explain why people choose the price ratio of 1.1384. This is indeed a good direction for further research, as we have already suggested in subsection 4.1.4.

[32]Most markets on TicketSwap are extremely buyer-heavy, so ideally we should randomise λ to be much higher than μ.

The structure of our artificial datasets allows us to compute each agent's utility gain from transacting on the market, measured in units of money. For agents who do not take part in any transaction (i.e. whose *Matched* is FALSE), this gain is of course zero. For a buyer who does buy a ticket, the utility gain is her reservation price minus the price she pays, $p_r - p$. A secondary seller's gain from selling her ticket is the share that she receives from its price[33], $(1-\tau)\cdot p$. Finally, the gain of a primary seller from selling one ticket is simply the original price itself[34], p_p.

Since these individual utilities are all measured on a common scale (money), they can be added to each other. This way we can obtain the aggregate surplus of buyers, primary and secondary sellers. We can also easily compute the profit of the secondary market platform[35]: we just have to multiply the surplus of secondary sellers by $\frac{\tau}{1-\tau}$. Then we can sum up these group-wise surpluses to get the social surplus (or welfare) generated on these ticket markets. These results are shown in figure 6.1, for both the constrained and unconstrained artificial datasets.

The arguments above also imply that the amount of welfare generated in a transaction is always equal to the buyer's reservation price (p_r). This amount is then divided between the buyer, the (primary or secondary) seller and the platform. Hence the aggregate social surplus is always equal to the sum of reservation prices of buyers who have a ticket. This means that in a Pareto-optimal allocation, each event's tickets should belong to the buyers with the highest reservation prices. Any other allocation can be Pareto-improved if a buyer who does not have a ticket buys it from someone whose reservation price is lower (for not less than this lower reservation price). In figure 6.1 we show what would be the social surplus if our simulated agents managed to reach such a Pareto-optimal allocation. We do not indicate how this surplus is distributed between the groups because it does not matter. Any redistribution can be Pareto-optimal as long as the tickets remain at the buyers with the highest reservation prices.

6.2.1 The magnitude of the social surplus

We can see from figure 6.1 that the aggregate social surplus in the constrained case is lower than in the unconstrained case. This means that the constraints have a negative effect on welfare, which is not surprising. The share that the platform collects (τ) and the two limits on the prices ($\underline{p}$ and $\overline{p}$) all cause market distortions. τ works exactly like a tax, which is known to have a deadweight loss. $\underline{p}$ keeps some prices at an artificially high level, so these tickets may go to waste if no buyer is willing to pay that much. Similarly, $\overline{p}$ keeps some prices artificially low, so buyers with low reservation prices might 'snap up' these tickets earlier than others who need them more. Therefore, this finding is completely in line with the basic economic intuition.

Figure 6.1 also shows that the Pareto-optimal case yields a higher social surplus than the other two cases. This is not surprising either, since the Pareto-optimum must by definition outscore any suboptimal allocation. In the previous paragraph we have given several arguments why the constrained case is suboptimal. The reason why the unconstrained case is suboptimal as well (even though the above arguments do not apply to it) is the dynamic nature of these ticket markets. For example, consider a market where only two buyer arrivals and zero secondary seller arrivals occur. Suppose that on this market there is a ticket at the primary seller, and its price is acceptable to both arriving buyers. In a

[33] The price that a secondary seller originally paid for her ticket is just a sunk cost, so we do not take that into account.

[34] Here we implicitly assume that the variable costs related to one additional guest turning up at the event are negligible. This assumption is of course debatable. For example, the costs of cleaning the venue are probably higher after a full-house concert than after a half-house one. Nevertheless, we will see in figure 6.1 that the surplus of primary sellers is almost the same in the constrained and unconstrained cases. Therefore, if the costs are proportionate to the surplus, they should also be approximately equal in the two cases.

[35] Similarly to the case of primary sellers, here we assume that the platform's variable costs are negligible. This is again a questionable assumption: maintaining a server is more expensive if the platform has more users. However, these costs probably depend on the number of *arrivals* (which is the same in the two cases), and not the number of *transactions*. Hence we do not need to worry about this either.

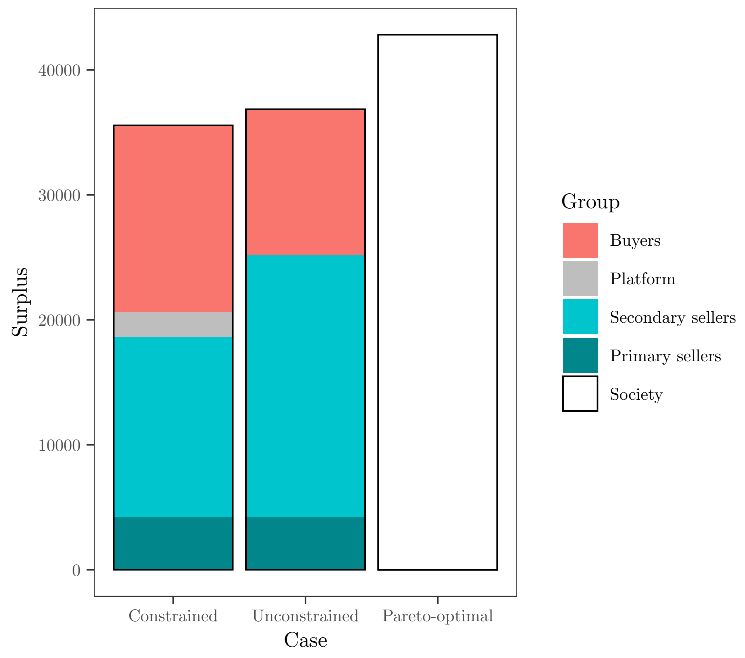


Figure 6.1: The aggregate surplus of each group in the two artificial datasets, compared to the Pareto-optimal social surplus.

Pareto-efficient allocation, this ticket should go to the buyer with the higher reservation price. However, if it is the buyer with the lower reservation price who arrives earlier, she can take the ticket from the other buyer, hence causing inefficiency. One could argue that on a free market this would not be a problem because buyers who had bought a ticket can also become secondary sellers. In theory, it would be rational for any buyer to try to resell her purchased ticket for a price above her reservation price. After all, if it is bought for that much, she would be better off than by going to the event. However, in real life, people do not work that way. Due to the endowment effect (Thaler, 1980; Kahneman, Knetsch & Thaler, 1990), they are not likely to resell goods they have already acquired.

To sum up, the order of the three social surpluses is exactly what basic economics suggests. But the differences between these values are more interesting. Although the welfare in the unconstrained case is indeed higher than in the constrained one, the difference is not that substantial. By removing the constraints, we only improve the aggregate social surplus by about 4%. In contrast, the Pareto-optimal welfare is about 20% higher than the constrained case. This means that, despite our claims about their distortive nature, these constraints do not matter that much. At the same time, the difference between the welfares of the Pareto-optimal and unconstrained cases is quite big. Therefore, we can conclude that most of the inefficiency is caused by the market's dynamic nature, and not the constraints themselves.

6.2.2 The composition of the social surplus

Having discussed how the magnitude of welfare varies across the three cases, now we turn to its composition. For the constrained and unconstrained cases[36], figure 6.1 shows how the social surplus is shared between the groups of agents. The only group whose surplus appears to be unaffected by the constraints is the group of primary sellers. For the other groups, we can observe a radical welfare redistribution. First of all, the platform does not receive any surplus in the unconstrained case, while in the constrained case it does. This is of course because the platform cannot collect any revenue when τ is zero. We can also see that in the unconstrained case the surplus of secondary sellers is more than twice as large as the surplus

[36]In the Pareto-optimal case we do not know these shares because any redistribution of welfare is efficient.

of buyers, but roughly the same in the constrained case. In other words, the constraints redistribute a substantial amount of welfare from buyers to sellers. This is probably because of the upper limits on the prices. The redistribution is so drastic that, even despite the slight decrease in the aggregate welfare, the surplus of buyers in the constrained case is much higher. Therefore, the constraints are beneficial for the group of buyers.

6.3 Implications for TicketSwap

This section applies the theoretical findings from above to a more practical context, and examines what they imply for TicketSwap. First of all, our results from subsection 6.2.1 indicate that TicketSwap has a substantial 'deadweight loss'. Figure 6.1 shows that the constrained case (which is analogous to TicketSwap, see 6.1) is far from being Pareto-optimal. However, we have also shown that the platform's rules are responsible for only a slight decrease of welfare. Most of the deadweight loss is the result of the dynamic nature of ticket markets, which is a given that TicketSwap can hardly change. In theory, it could eliminate the dynamic aspect by collecting the tickets of all arriving secondary sellers, and then giving them out to the highest-bidding buyers in an auction just before the event[37]. Mechanisms like the Vickrey–Clarke–Groves auction (Vickrey, 1961; Clarke, 1971; Groves, 1973) would even enable the platform to reach the Pareto-optimal outcome. This solution is similar to what has been proposed by several authors in the literature (Miyashita, 2014, 2017; Waisman, 2021). However, it is disputable whether such an allocation mechanism could be applied to these ticket markets. Buyers may dislike the idea that they have to wait until the last moment to find out whether they received a ticket or not. What is more, the final price in these auctions would probably also be higher than in the constrained case (where there are upper limits). These issues may cause some buyers to leave TicketSwap and try to purchase tickets elsewhere. This is clearly not something that the platform would like, so trying this mechanism would be risky. But it may still work, so it would be interesting to see how it turns out. This is a good direction for further research as well.

Nevertheless, if we accept that TicketSwap cannot change the dynamic nature of ticket markets, our results draw a rather favourable picture of the platform. The deadweight loss that can be attributed to its rules is almost negligible. What is more, TicketSwap has a serious advantage which was not included in our model. It substantially reduces the transaction costs of buying and selling tickets. Taking this into account, its net effect on welfare is almost certainly positive.

In section 6.2.2 we have also shown that the rules of TicketSwap result in a serious redistribution of welfare: buyers get a much higher surplus due to the upper limits on the prices. Why the platform uses these upper limits is an interesting question. One could even argue that by artificially lowering prices, TicketSwap reduces its own profits, so this behaviour is irrational. The obvious explanation of course is that the same public outrage that has lead to anti-scalping laws (Williams, 1994) is responsible for this phenomenon as well. However, the framework of two-sided markets provides yet another rational explanation for this. For a platform on a two-sided market, it is optimal to set a lower price for the group whose demand is more elastic (Rochet & Tirole, 2003; Armstrong, 2006). This way it can attract more agents from that group and hence make a higher profit. Therefore, if it is the group of buyers that has a more elastic demand[38], the upper price limit can be seen as an implicit form of platform pricing. From this perspective, trying to keep prices on a lower level is just an effort to attract more buyers to TicketSwap. However, our data suggests that the platform is 'too successful' in this: on most markets there are much more buyers than sellers. Therefore, in theory, the platform should try to attract sellers instead of buyers.

[37]Then the platform could distribute the auction's proceeds between the sellers (and itself).

[38]I.e. they have more alternative ways of acquiring tickets than buyers have of selling them.

7 Limitations

Although the approach we used in this paper is quite sophisticated, our research (like any) of course has some limitations. These limitations may result from the imperfect structure of our dataset, the unrealistic assumptions of our model, or the arbitrary choices in our simulations. In this chapter we cover these three issues one by one.

7.1 Data imperfections

Throughout the analysis of the TicketSwap dataset (see section 3.3) we have treated the *TimeLeftNow* variable as the time left until the event when the current ticket was *uploaded*. However, this *TimeLeftNow* was derived from the *TimeRecordedTicket* variable, which shows the time when we *observed* the ticket. This is of course related to when it was uploaded, but clearly not the same. In fact, the time between two observations of the same event was about one and a half hours on average. This means that when we see the value of *TimeLeftNow* for a certain ticket, all we know is that it was uploaded at some point in the last 1.5 hours before that value. Therefore, the *TimeLeftNow* variable is not perfectly accurate. Moreover, the same goes for the other variables as well. The explanatory variables from our regressions (see table 3.3) all show the event's data when the current ticket was *observed*, and not when it was *uploaded*.

The basic problem is that the `.json` files we download from the TicketSwap API do not contain any information about the tickets at the time of their upload, only at the moment of our current request. As a result, we cannot observe these events in continuous time, like it would be ideal for a proper analysis. The best we can do instead is take snapshots of each event's ticket market as frequently as possible. This is exactly how our webscraping algorithm is designed (see section 3.2 or appendix A.1).

This is a serious limitation on our analysis, which relied on the assumption that the values we observe in the data are the same values that the sellers saw when they were uploading their tickets. If this is not true, our results become questionable. For example, when two ticket listings for a certain event are first observed in the same snapshot, we cannot even tell which one was uploaded earlier[39].

However, this does not mean that our of our results are worthless. After all, the time window of 1.5 hours is not that large, especially as we have been scraping the dataset for several months. At this magnitude, a few hours of inaccuracy is not that big of a deal. Also, since we treat time as a continuous variable, the inaccurate order of tickets from the same snapshot does not cause problems either. The amount of time between recording two tickets from the same snapshot is measured in milliseconds, so it definitely does not bias our regression results.

7.2 Assumptions in our model

In order to make our model solvable, we needed to simplify some things by making somewhat unrealistic assumptions in section 4.1. In the current section we highlight these assumptions and argue why we needed them. First of all, the assumption of Poisson arrivals seems very unrealistic at the first glance since it would mean that the rate of arrivals is constant over time. We have seen in the data that this is clearly not the case since most arrivals occur in the last days before the event. However, this is not a great problem since it is only a matter of scaling. We could easily transform the time variable in a way that corresponds to what we see in the dataset and interpret the results accordingly. We can think of time in our model as being measured on a relative scale that implies constant arrival rates. This of course

[39]In our analysis we try to make use of the order in which they are listed in the `.json` file we download from the API. However, there is no way to tell if that is indeed the order in which they are uploaded.

means that two time intervals with the same length on this relative scale may actually have different lengths if we measure them in real time. But this does not make any difference in the results we get.

Nevertheless, by assuming that buyer and seller arrivals *both* occur according to a Poisson process on the *same* relative time scale we do make a limiting assumption. We actually assume that these two arrival rates follow the same patterns, i.e. when buyers arrive relatively rarely, sellers also arrive relatively rarely. This might not be the case in real life, but the structure of our data is not sufficient to actually check it. All we can say is that we have no solid arguments implying that the two arrival rates should follow significantly different patterns, so at least this assumption is not counter-intuitive.

The assumption that secondary sellers always have exactly one ticket to sell is also clearly not true. People rarely go to these events alone, and hence they often buy several tickets at once. Therefore, if they decide to sell these tickets, they may sell them together. We could of course modify our model by assuming that some secondary sellers have several tickets to sell, but this would mean that we need to consider cases where only some of these tickets are sold. Although such a modification would be feasible, we believe that it would not be worth the effort. Our model would just get even more complicated while the results would not change that much in return. What is more, we would need to make further arbitrary assumptions about how to randomise the number of tickets that a certain seller wants to sell. Therefore we stick to our assumption that each secondary seller has one ticket to sell. The good news is that this is not as far from reality as we would think: in the TicketSwap dataset, around 65% of the listings include one ticket only.

The same assumption for buyers is not that much of a problem. The behaviour of one buyer trying to buy several tickets is equivalent to the behaviour of several buyers (with equal reservation prices) trying to buy just one. This means that if we wanted to include buyers looking for several tickets, we would only need to modify the process of arrivals. But this would still require further arbitrary assumptions, so we do not make this modification either. Doing so may be a good direction for further research.

Our next questionable assumption is that buyers do not think strategically. It would be a rational thing to wait for a cheaper ticket to come even if there are already some available tickets at acceptable prices. But this behaviour would be very risky, especially on a buyer-heavy market (where λ is much greater than μ). After all, cheaper tickets are not certain to come, and even if they do so, other buyers might be faster and take those cheaper tickets. What is more, the original tickets may also be taken by other buyers in the meantime. This uncertainty (together with the harsh competition between buyers) may indeed incentivise buyers to instantly go for any available ticket which is acceptable for them. However, there may be some buyers who are willing to take these risks. And the less buyer-heavy a market is, the more likely that such buyers will occur.

Nevertheless, if we look at our data we can see that seller-heavy markets seem to be very rare. Out of the observations we have made on the events, only around 7% showed that the number of available tickets was larger than the number of ticket alerts at the moment. What is more, the number of available tickets was exactly zero in more than 20% of the observations. These statistics give us a hint that most secondary ticket markets are probably buyer-heavy and hence only really brave buyers are incentivised to wait for cheaper tickets. Therefore our assumption of buyers instantly buying any acceptable ticket may not be that far from reality. But as long as the buyers are not all extremely risk-averse, strategic thinking will still occur to some extent. Hence this is still a limitation we need to take into account.

Another limitation comes from the assumption that the reservation price of a buyer is independent from her arrival time. One might argue that a buyer whose reservation price is high is more desperate to buy tickets and hence arrives earlier. Another argument may be that the mass of potential buyers is not infinite, so a buyer's entry always changes the distribution of reservation prices in the population outside of the market. These are of course valid concerns, but as long as the distribution does not change radically over time, our model provides a good approximation.

We also assumed that once a seller has set a price, she cannot change it later. Although this is not the case in real life[40], we need this assumption to make our model work. If we did not assume this, our seller would be able to maximise her utility by choosing a *function* (instead of a single number) which returns an optimal price for each moment in the future. Fortunately, we can see in the dataset that most people do not tend to change the price of their listed ticket that often. The price remains unchanged in around 85% of the listings in our dataset (although this is partly because some tickets are sold so quickly that sellers do not even have time to change their price).

The assumption that the parameters and distribution functions are common knowledge among sellers is of course is again a simplification, especially in the case of λ and F. However, having even more imperfect information in our model would make it impossible to handle. And even if sellers are not able to precisely quantify these values, they must have at least an approximate intuition of what they can be.

Finally, the most unrealistic assumption in our model is that secondary sellers believe that no more sellers will enter the market after them. We cannot really defend this assumption with solid arguments, but we need it to make the model solvable. If we let the sellers consider later seller arrivals too, the probability from the utility maximisation problem would be impossible to obtain. Therefore we can only hope that this assumption does not bias our results too much. A good thing is that we at least know the direction of the bias: by not taking later seller entries into account, sellers overestimate the probability of their tickets being sold. This means that they choose higher prices than they would if they knew the true value of μ.

7.3 The parametrisation of our simulations

In section 5.1, when choosing how to randomise the parameters in our simulations, we needed to make important decisions which may have had huge influence on our final results. Ideally, we could base these decisions on some kind of empirical findings, but since the literature on this topic is scarce, we were not able to. In most cases we could not use the TicketSwap dataset either, because it did not include any observations on latent variables like reservation prices or risk aversion parameters. Therefore, our decisions were determined to be arbitrary, which of course means a significant limitation to our research.

We have decided to use uniform distributions in all cases precisely because this seemed like the ‘least arbitrary’ choice. Had we assumed anything else (say, a normal distribution), we would have needed to make further arbitrary assumptions about the parameters describing this distribution (in the normal case, its mean and variance). In the case of uniform distribution, the only things we need to specify are the two endpoints of the interval. This is much easier, especially when we can use some natural ‘limits’ on the parameter values (like in the case of α, where the values need to fall between 0 and 1 by definition), or when we can normalise the maximum value to 1 (like in the cases of p_r and p_p). Similarly, in equation (5.4) we implicitly assumed that $\underline{p}$ never exceeds the ticket’s original price from the primary market (p_p). This is a rather plausible assumption (it is true for almost all observations in the TicketSwap dataset).

In the remaining three randomisations (concerning λ, μ and N_p), our reason for choosing 20, 15 and 5 (respectively) as maximal values was purely technical. Larger values for these parameters would have resulted in exceeding our computational capacity[41]. Therefore we went for the largest values which our computer could handle in a reasonable amount of time and chose them as upper bounds.

Needless to say, these values most probably do not follow uniform distributions in real life, nor do λ, μ and N_p fall below 20, 15 and 5. But these assumptions are the best we can do with the computational capacity of a laptop, and no empirical findings to base our results on.

[40] At TicketSwap, sellers can change their prices or delete their listing anytime they want.

[41] Obtaining the set of compositions (see (4.11)) for integers of this magnitude is a difficult task for a single computer.

8 Discussion

Our research question in this paper was how the rules of platforms on secondary ticket markets affect social welfare. We investigated this question by analysing a dataset, designing a model and then running simulations based on it. Each of these three steps is valuable in its own right, as well as the results we draw from them. The dataset we have scraped from TicketSwap is already very large, and it will grow even further as we continue scraping it. What is more, we can easily modify our codes to collect data from other cities as well (and not just from Amsterdam). This way we could also see whether there are spatial differences in seller behaviour.

The model we have designed is also an important contribution to the literature. Further research could use it for many other purposes, from evaluating certain anti-scalping laws to analysing seller behaviour. We just need to change the settings a bit and then run simulations in the same way. The model can also be extended in many ways. As we have mentioned above, applying the prospect-theoretical framework would be an obvious extension. But we could include any other behavioural aspect as well. Ideally, it would also be nice to remove our limiting assumptions (see section 7.2), but we fear that such a model would be impossible to handle.

Our simulations could also be improved if we knew more about the parameters of our model. Further research could try to estimate their values somehow, but this of course requires empirical data with a suitable structure. It is possible that researchers would need to conduct experiments to collect such data.

Our research has shown that TicketSwap's constraints alone do not result in a significant loss of welfare. We have also claimed that even this slight decrease is probably offset by the fact that the platform reduces transaction costs on the secondary market. This is our paper's key result, suggesting that these platforms are overall beneficial to society. A closely related finding is that that the reason why these markets are still far from the Pareto-optimum lies in their dynamic nature. We argued that this could be eliminated if the platforms used auctions to decide who gets tickets. Although it is questionable whether this could work in real life, designing an optimal auction is still a good direction for further research. Finally, we have shown that while keeping aggregate welfare at roughly the same level, TicketSwap's constraints shift a large surplus from secondary sellers to buyers. We have mentioned platform pricing as a possible reason for that, but further research could dig deeper into this issue.

References

Arkes, H. R. & Blumer, C. (1985). The psychology of sunk cost. *Organizational Behavior and Human Decision Processes*, *35*(1), 124–140.

Armstrong, M. (2006). Competition in two-sided markets. *The RAND Journal of Economics*, *37*(3), 668-691.

Bayes, T. & Price, R. (1763). An essay towards solving a problem in the doctrine of chances. By the late Rev. Mr. Bayes, F.R.S. communicated by Mr. Price, in a letter to John Canton, A.M.F.R.S. *Philosophical Transactions of the Royal Society of London*, *53*, 370–418.

Bergé, L. (2018). Efficient estimation of maximum likelihood models with multiple fixed-effects: the R package FENmlm. *CREA Discussion Papers*(13).

Breusch, T. S. & Pagan, A. R. (1979). A simple test for heteroscedasticity and random coefficient variation. *Econometrica*, *47*(5), 1287–1294.

Brewer, M. B. (1979). In-group bias in the minimal intergroup situation: A cognitive-motivational analysis. *Psychological Bulletin*, *86*(2), 307–324.

Budish, E. & Bhave, A. (in press). Primary-market auctions for event tickets: Eliminating the rents of "Bob the Broker"? *American Economic Journal: Microeconomics*.

Chen, Y. & Li, S. X. (2009). Group identity and social preferences. *American Economic Review*, *99*(1), 431–57.

Clarke, E. H. (1971). Multipart pricing of public goods. *Public Choice*, *11*(1), 17–33.

Courty, P. (2000). An economic guide to ticket pricing in the entertainment industry. *Recherches Économiques de Louvain/Louvain Economic Review*, *66*(2), 167–192.

Courty, P. (2003a). Some economics of ticket resale. *Journal of Economic Perspectives*, *17*(2), 85–97.

Courty, P. (2003b). Ticket pricing under demand uncertainty. *The Journal of Law and Economics*, *46*(2), 627–652.

Depken, C. A. (2007). Another look at anti-scalping laws: Theory and evidence. *Public Choice*, *130*(1), 55–77.

Fehr, E. & Schmidt, K. M. (1999). A theory of fairness, competition, and cooperation. *The Quarterly Journal of Economics*, *114*(3), 817–868.

Groves, T. (1973). Incentives in teams. *Econometrica*, *41*(4), 617–631.

Hankin, R. K. S. (2006). Additive integer partitions in R. *Journal of Statistical Software, Code Snippets*, *16*.

Kahneman, D., Knetsch, J. L. & Thaler, R. H. (1990). Experimental tests of the endowment effect and the Coase theorem. *Journal of Political Economy*, *98*(6), 1325–1348.

Kahneman, D. & Tversky, A. (1979). Prospect theory: An analysis of decision under risk. *Econometrica*, *47*(2), 263–291.

Karp, L. & Perloff, J. M. (2005). When promoters like scalpers. *Journal of Economics & Management Strategy*, *14*(2), 477–508.

Krueger, A. B. (2001). Supply and demand: An economist goes to the Super Bowl. *Milken Institute Review*, *3*(2), 22–29.

Leslie, P. & Sorensen, A. (2009). *The welfare effects of ticket resale.* Working paper, National Bureau of Economic Research.

Miyashita, K. (2014). Online double auction mechanism for perishable goods. *Electronic Commerce Research and Applications*, *13*(5), 355–367.

Miyashita, K. (2017). Incremental design of perishable goods markets through multi-agent simulations. *Applied Sciences*, *7*(12), 1300.

R Core Team. (2022). R: A language and environment for statistical computing [Computer software manual]. Vienna. Retrieved from https://www.R-project.org/

Rochet, J. C. & Tirole, J. (2003). Platform competition in two-sided markets. *Journal of the European Economic Association*, *1*(4), 990-1029.

Roth, A. E. (2007). Repugnance as a constraint on markets. *Journal of Economic Perspectives*, *21*(3), 37–58.

Samuelson, W. & Zeckhauser, R. (1988). Status quo bias in decision making. *Journal of Risk and Uncertainty*, *1*(1), 7–59.

Sweeting, A. (2012). Dynamic pricing behavior in perishable goods markets: Evidence from secondary markets for major league baseball tickets. *Journal of Political Economy*, *120*(6), 1133–1172.

Thaler, R. H. (1980). Toward a positive theory of consumer choice. *Journal of Economic Behavior & Organization*, *1*(1), 39–60.

TicketSwap B.V. (2022a). *Platform Agreement and Ticket Agreement.* Retrieved 12th November 2022, from https://www.ticketswap.com/content/conditions

TicketSwap B.V. (2022b). *TicketSwap.* (Version 4.30.0) [Mobile application]. App store. Retrieved 12th November 2022, from https://apps.apple.com/app/ticketswap/id932337449

Tversky, A. & Kahneman, D. (1992). Advances in prospect theory: Cumulative representation of uncertainty. *Journal of Risk and Uncertainty*, *5*(4), 297–323.

van Rossum, G. & Drake, F. L. (2009). Python 3 reference manual [Computer software manual]. Scotts Valley: CreateSpace. Retrieved from https://www.python.org/

Vickrey, W. (1961). Counterspeculation, auctions, and competitive sealed tenders. *The Journal of Finance*, *16*(1), 8–37.

von Neumann, J. & Morgenstern, O. (1947). *Theory of Games and Economic Behavior* (2nd ed.). Princeton: Princeton University Press. (Original work published 1944)

Waisman, C. (2021). Selling mechanisms for perishable goods: An empirical analysis of an online resale market for event tickets. *Quantitative Marketing and Economics*, *19*(2), 127–178.

Waters, R. (1979). In the flesh [Song recorded by Pink Floyd]. In *The Wall.* Harvest Records.

Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L. D., François, R., . . . Yutani, H. (2019). Welcome to the tidyverse. *Journal of Open Source Software*, *4*(43), 1686.

Williams, A. T. (1994). Do anti-ticket scalping laws make a difference? *Managerial and Decision Economics*, *15*(5), 503–509.

A Codes

A.1 Webscraping

This appendix includes our codes for scraping data from TicketSwap. For the webscraping process we use the Python programming language (van Rossum & Drake, 2009). We first need to import the required packages.

```python
import numpy as np
import pandas as pd
import requests
import json
import time
import pyarrow
import datetime
from datetime import timedelta
import random
import os
```

Next, we create several functions (one for each variable) for extracting the permanent[42] data of events in our dataset. These functions follow the structure of the `.json` files we obtain after posting a search query (`getPopularEvents`) to the TicketSwap API (see later). To make sure that some missing information does not result in an error, the functions are all defined using the `try-except` clauses.

```python
def GetID(edge):
    try:
        return edge["node"]["id"]
    except:
        pass

def GetName(edge):
    try:
        return edge["node"]["slug"]
    except:
        pass

def GetCategory(edge):
    try:
        return edge["node"]["category"]
    except:
        pass

def GetCountry(edge):
    try:
        return edge["node"]["country"]["code"]
    except:
        pass

def GetCity(edge):
    try:
        return edge["node"]["location"]["city"]["name"]
    except:
```

[42]By permanent data, we refer to the basic characteristics (e.g. name, start date) of an event that do not change over time. In contrast, other variables like the number of available tickets are called ‘changing data’.

```
        pass

def GetStartDate(edge):
    try:
        return edge["node"]["startDate"]
    except:
        pass

def GetEndDate(edge):
    try:
        return edge["node"]["endDate"]
    except:
        pass

def GetStatus(edge):
    try:
        return edge["node"]["status"]
    except:
        pass
```

Similarly, we define functions that collect the changing data of a certain event. These functions follow the structure of a different `.json` file, the one we obtain after posting the `getEventStructuredData` query (see later).

```
def GetStatus2(EventData):
    try:
        return EventData["data"]["node"]["status"]
    except:
        pass

def GetSoldOut(EventData):
    try:
        return EventData["data"]["node"]["externalPrimaryTicketShops"][0]["state"]
    except:
        pass

def GetNumberOfAvailable(edge):
    try:
        return edge["node"]["availableTicketsCount"]
    except:
        pass

def GetNumberOfSold(edge):
    try:
        return edge["node"]["soldTicketsCount"]
    except:
        pass

def GetNumberOfAlerts(edge):
    try:
        return edge["node"]["ticketAlertsCount"]
    except:
        pass

def GetPopular(edge):
    try:
        return edge["node"]["isPopular"]
    except:
        pass
```

Finally, we create functions that collect data about the (available or sold) ticket listings themselves. These functions also follow the structure of the `getEventStructuredData` query's result (see later).

```
def GetTicketsInListing(edge):
    try:
        return edge["node"]["numberOfTicketsInListing"]
    except:
        pass

def GetTicketsStillForSale(edge):
    try:
        return edge["node"]["numberOfTicketsStillForSale"]
    except:
        pass

def GetSellerID(edge):
    try:
        return edge["node"]["seller"]["id"]
    except:
        pass

def GetOriginalPrice(edge):
    try:
        return edge["node"]["price"]["originalPrice"]["amount"]
    except:
        pass

def GetOriginalCurrency(edge):
    try:
        return edge["node"]["price"]["originalPrice"]["currency"]
    except:
        pass

def GetTotalPrice(edge):
    try:
        return edge["node"]["price"]["totalPriceWithTransactionFee"]["amount"]
    except:
        pass

def GetTotalCurrency(edge):
    try:
        return edge["node"]["price"]["totalPriceWithTransactionFee"]["currency"]
    except:
        pass

def GetSellerPrice(edge):
    try:
        return edge["node"]["price"]["sellerPrice"]["amount"]
    except:
        pass

def GetSellerCurrency(edge):
    try:
        return edge["node"]["price"]["sellerPrice"]["currency"]
    except:
        pass
```

The GetPermanentData, GetChangingData, and GetTicketData functions simply use the above-defined functions to create a data frame row containing the extracted variables. As implied by their names, GetPermanentData and GetChangingData are for extracting the permanent and changing data of an event, while GetTicketData is for collecting data about a ticket listing.

```
def GetPermanentData(edge, edge2):
    df = pd.DataFrame(
        {
            "EventID": [GetID(edge)],
            "EventName": [GetName(edge)],
            "TypeID": [GetID(edge2)],
            "TypeName": [GetName(edge2)],
            "Country": [GetCountry(edge)],
            "City": [GetCity(edge)],
            "Category": [GetCategory(edge)],
            "StartDate": [GetStartDate(edge2)],
            "EndDate": [GetEndDate(edge2)],
            "Status": [GetStatus(edge)],
        }
    )
    return df

def GetChangingData(EventData, edge, eventid):
    df = pd.DataFrame(
        {
            "EventID": [eventid],
            "TypeID": [GetID(edge)],
            "NumberOfAvailable": [GetNumberOfAvailable(edge)],
            "NumberOfSold": [GetNumberOfSold(edge)],
            "NumberOfAlerts": [GetNumberOfAlerts(edge)],
            "Popular": [GetPopular(edge)],
            "SoldOut": [GetSoldOut(EventData)],
            "TimeRecorded": [datetime.datetime.now()],
        }
    )
    return df

def GetTicketData(edge, edge2, eventid):
    df = pd.DataFrame(
        {
            "EventID": [eventid],
            "TypeID": [GetID(edge)],
            "ListingID": [GetID(edge2)],
            "SellerID": [GetSellerID(edge2)],
            "TicketsInListing": [GetTicketsInListing(edge2)],
            "TicketsStillForSale": [GetTicketsStillForSale(edge2)],
            "OriginalPrice": [GetOriginalPrice(edge2)],
            "OriginalCurrency": [GetOriginalCurrency(edge2)],
            "TotalPrice": [GetTotalPrice(edge2)],
            "TotalCurrency": [GetTotalCurrency(edge2)],
            "SellerPrice": [GetSellerPrice(edge2)],
            "SellerCurrency": [GetSellerCurrency(edge2)],
            "TimeRecorded": [datetime.datetime.now()],
        }
    )
    return df
```

The GetNewEvents function is used for adding new events to our PermanentData dataset. Its arguments are the unique ID of the city we are looking for events in, the date of the events (this can either be a particular date or 'ANYTIME'), whether or not we are looking for highlighted events, and the session of internet communication.

In the function we first store the whole search query as SearchArgs. This query is a modified version of what is automatically posted to the API when we conduct a search on TicketSwap. Due to the slightly different syntax, we need to distinguish between searches where the exact date is specified and searches where only 'ANYTIME' is given. After waiting for a random amount of time between 1 and 4 seconds, we try to post this query to the API and store the response as Search. If this trial is not successful, the programme waits for one hour and tries again. If this is still not successful, it waits another two hours, and then a whole day. These waiting times are needed because TicketSwap limits the frequency of queries posted to its API, and if a certain IP address exceeds this limit, it is banned from the server for some time. This is how TicketSwap tries to distinguish humans from bots (like this programme). Luckily for us, these methods are not perfect, and the waiting times still enable us to efficiently scrape data.

Once we obtain the query's response in a .json file (Search), we can start adding new events to our PermanentData dataset. First, we create an empty data frame (NewPermanentData) which has the same columns as PermanentData. Next, we open the .feather file containing the PermanentData dataset[43]. Then we go through the events in Search one by one. If the current event is already in PermanentData, we do not do anything. But if it is not, we use the GetPermanentData function to extract data on all of its event types from Search. We append the resulting row to NewPermanentData. Just like before, we use try-except clauses to avoid errors. Finally, after we have dealt with all events in Search, we append NewPermanentData to PermanentData and save PermanentData in our working directory as PermanentData.feather.

```
def GetNewEvents(CityID, Date, IsHighlighted, sess):
    if Date == "ANYTIME":
        SearchArgs = {
            "operationName": "getPopularEvents",
            "variables": {
                "cityId": CityID,
                "period": "ANYTIME",
                "first": 99,
                "highlighted": IsHighlighted,
            },
            "query": """
            query getPopularEvents(
              $first: Int,
              $after: String,
              $highlighted: Boolean,
              $period: Period,
              $date: DateTime,
              $category: EventCategory,
              $cityId: ID,
              $locationId: ID,
              $nearby: GeopointFilter) {
                activeEvents(
                  first: $first
```

[43]This code is designed in a way that assumes that we already have some data saved in .feather files. This was of course not true when we started the whole webscraping process, so our first codes were slightly different.

```
                  after: $after
                  period: $period
                  date: $date
                  orderBy: {field: BOOST_VALUE, direction: DESC}
                  filter: {
                    locationId: $locationId,
                    category: $category,
                    highlighted: $highlighted,
                    city: $cityId,
                    nearby: $nearby
                    }
                  ) {edges {
                       node {
                         ...eventList
                         }
                       }
                     }
                }
            fragment eventList on Event {
              id
              slug
              category
              status
              types(first: 99) {
                edges {
                  node {
                    id
                    slug
                    startDate
                    endDate
                    }
                  }
                }
              country {
                ...country
                }
              location {
                city {
                  name
                  }
                }
              }
            fragment country on Country {
              code
              }
          """,
        }
    else:
        SearchArgs = {
            "operationName": "getPopularEvents",
            "variables": {
                "cityId": CityID,
                "date": Date,
                "first": 99,
                "highlighted": IsHighlighted,
            },
            "query": """
            query getPopularEvents(
              $first: Int,
              $after: String,
              $highlighted: Boolean,
              $period: Period,
              $date: DateTime,
              $category: EventCategory,
              $cityId: ID,
              $locationId: ID,
              $nearby: GeopointFilter) {
                activeEvents(
                  first: $first
                  after: $after
                  period: $period
                  date: $date
                  orderBy: {field: BOOST_VALUE, direction: DESC}
                  filter: {
                    locationId: $locationId,
                    category: $category,
                    highlighted: $highlighted,
                    city: $cityId,
                    nearby: $nearby
                    }
                  ) {edges {
                       node {
                         ...eventList
                         }
                       }
                     }
                }
            fragment eventList on Event {
              id
              slug
              category
              status
              types(first: 99) {
                edges {
                  node {
                    id
                    slug
                    startDate
                    endDate
                    }
                  }
                }
              country {
                ...country
                }
              location {
                city {
                  name
                  }
                }
              }
            fragment country on Country {
              code
              }
          """,
        }
```

```
try:
    time.sleep(random.uniform(1, 4))
    Search = sess.post(
        "https://api.ticketswap.com/graphql/public",
        json=SearchArgs,
        headers={"User-Agent": "Mozilla/5.0"},
    )
    Search = json.loads(Search.content)
except:
    try:
        print("WAITED!")
        time.sleep(3600)
        Search = sess.post(
            "https://api.ticketswap.com/graphql/public",
            json=SearchArgs,
            headers={"User-Agent": "Mozilla/5.0"},
        )
        Search = json.loads(Search.content)
    except:
        try:
            print("WAITED 2!")
            time.sleep(3600 * 2)
            Search = sess.post(
                "https://api.ticketswap.com/graphql/public",
                json=SearchArgs,
                headers={"User-Agent": "Mozilla/5.0"},
            )
            Search = json.loads(Search.content)
        except:
            print("WAITED A DAY!")
            time.sleep(3600 * 24)
            Search = sess.post(
                "https://api.ticketswap.com/graphql/public",
                json=SearchArgs,
                headers={"User-Agent": "Mozilla/5.0"},
            )
            Search = json.loads(Search.content)

NewPermanentData = pd.DataFrame(
    columns=[
        "EventID",
        "EventName",
        "TypeID",
        "TypeName",
        "Country",
        "City",
        "Category",
        "StartDate",
        "EndDate",
        "Status",
    ]
)

PermanentData = pd.read_feather("PermanentData.feather")

try:
    for edge in Search["data"]["activeEvents"]["edges"]:
        if GetID(edge) in PermanentData["EventID"].values:
            pass

        else:
            try:
                for edge2 in edge["node"]["types"]["edges"]:
                    NewPermanentData = NewPermanentData.append(
                        GetPermanentData(edge, edge2), ignore_index=True
                    )
            except:
                pass
except:
    print("PROBLEM WITH SEARCH!")

PermanentData = PermanentData.append(NewPermanentData, ignore_index=True)
PermanentData.astype(str).to_feather("PermanentData.feather")
```

The `UpdateAll` function is used for updating the data of events already in our datasets. To make our code as efficient as possible, we store our data in four separate datasets (including `PermanentData`). They will be merged into a single dataset later in section A.2. As implied by their names, `PermanentData` and `ChangingData` are for storing the permanent and changing data of events[44], while `AvailableTickets` and `SoldTickets` are for storing data on (available and sold) ticket listings. The `UpdateAll` function first opens these four datasets from the previously saved `.feather` files. Next, it creates three empty data frames (`NewChangingData`, `NewAvailableTickets` and `NewSoldTickets`). Then it goes through the active[45] events in `PermanentData` one by one, and performs several tasks on them.

For each active event, the function stores a so-called `getEventStructuredData` query as `EventDataArgs`. This query is a modified version of what is posted to the TicketSwap API every time a user loads a certain event's page. `EventDataArgs` is then posted to the TicketSwap API, and the response is stored as `EventData`. Like in the `GetNewEvents` function, we use waiting times to make sure that TicketSwap does not identify us as bots. If the current event's status is still active[46], we go through each of its event types one by one, and use the `GetChangingData` function to extract their changing data in a data frame row. We append this row to `NewChangingData`. Then we also go through each available and sold ticket listing for the current event type, and use the `GetTicketData` function to extract their data. These rows are appended to the `NewAvailableTickets` and `NewSoldTickets` data frames. To be as cautious as possible, we use the `try-except` clauses again, and print out the current event ID if any of these operations are unsuccessful.

Once we are done with all active events, we save the `PermanentData` dataset as `PermanentData.feather` in our working directory. Then we append the `NewChangingData`, `NewAvailableTickets` and `NewSoldTickets` data frames to the `ChangingData`, `AvailableTickets` and `SoldTickets` datasets. In each case, we convert all variables to strings, and drop rows which differ only in the `TimeRecorded` variable. Finally, we save them as `.feather` files in our working directory.

```
def UpdateAll(sess):
    PermanentData = pd.read_feather("PermanentData.feather")
    ChangingData = pd.read_feather("ChangingData.feather")
    AvailableTickets = pd.read_feather("AvailableTickets.feather")
    SoldTickets = pd.read_feather("SoldTickets.feather")
```

[44] The only exception is the `Status` variable, which is changing over time but is stored in the `PermanentData` dataset for practical reasons.
[45] An event is active if tickets for it are currently being sold on the platform.
[46] If it is not, we modify its row in the `PermanentData` dataset accordingly.

```
    NewChangingData = pd.DataFrame(
        columns=[
            "EventID",
            "TypeID",
            "NumberOfAvailable",
            "NumberOfSold",
            "NumberOfAlerts",
            "Popular",
            "SoldOut",
            "TimeRecorded",
        ]
    )

    NewAvailableTickets = pd.DataFrame(
        columns=[
            "EventID",
            "TypeID",
            "ListingID",
            "SellerID",
            "TicketsInListing",
            "TicketsStillForSale",
            "OriginalPrice",
            "OriginalCurrency",
            "TotalPrice",
            "TotalCurrency",
            "SellerPrice",
            "SellerCurrency",
            "TimeRecorded",
        ]
    )

    NewSoldTickets = NewAvailableTickets

    ActivesOnly = PermanentData.query("Status == 'ACTIVE'")

    for eventid in ActivesOnly["EventID"].unique():

        EventDataArgs = {
            "operationName": "getEventStructuredData",
            "variables": {"id": eventid},
            "query": """
            query getEventStructuredData($id: ID!){
              node(id: $id) {
                ... on Event {
                  status
                  externalPrimaryTicketShops {
                    ... externalPrimaryTicketShop
                    }
                  types(first: 99) {
                    edges {
                      node {
                        id
                        isPopular
                        availableTicketsCount
                        soldTicketsCount
                        ticketAlertsCount
                        availableListings: listings(
                          first: 99,
                          filter: {listingStatus: AVAILABLE}
                          ) {
                              ...listings
                              }
                        soldListings: listings(
                          first: 99,
                          filter: {listingStatus: SOLD}
                          ) {
                              ...listings
                              }
                        }
                      }
                    }
                  }
                }
              }
            fragment externalPrimaryTicketShop on ExternalPrimaryTicketShop {
              state
              }
            fragment listings on ListingConnection {
              edges {
                node {
                  ...listingList
                  }
                }
              }
            fragment listingList on PublicListing {
              id
              numberOfTicketsInListing
              numberOfTicketsStillForSale
              status
              seller {
                id
                }
              price {
                originalPrice {
                  ...money
                  }
                totalPriceWithTransactionFee {
                  ...money
                  }
                sellerPrice {
                  ...money
                  }
                }
              }
            fragment money on Money {
              amount
              currency
              }
            """,
        }

        try:
            time.sleep(random.uniform(1, 4))
            EventData = sess.post(
                "https://api.ticketswap.com/graphql/public",
                json=EventDataArgs,
                headers={"User-Agent": "Mozilla/5.0"},
            )
            EventData = json.loads(EventData.content)
        except:
            try:
                print("WAITED!")
                time.sleep(3600)
                EventData = sess.post(
                    "https://api.ticketswap.com/graphql/public",
                    json=EventDataArgs,
                    headers={"User-Agent": "Mozilla/5.0"},
                )
                EventData = json.loads(EventData.content)
            except:
                try:
                    print("WAITED 2!")
                    time.sleep(3600 * 2)
                    EventData = sess.post(
                        "https://api.ticketswap.com/graphql/public",
                        json=EventDataArgs,
                        headers={"User-Agent": "Mozilla/5.0"},
                    )
                    EventData = json.loads(EventData.content)
                except:
                    print("WAITED A DAY!")
                    time.sleep(3600 * 24)
                    EventData = sess.post(
                        "https://api.ticketswap.com/graphql/public",
                        json=EventDataArgs,
                        headers={"User-Agent": "Mozilla/5.0"},
                    )
                    EventData = json.loads(EventData.content)

        if GetStatus2(EventData) == "ACTIVE":

            for edge in EventData["data"]["node"]["types"]["edges"]:
                try:
                    NewChangingData = NewChangingData.append(
                        GetChangingData(EventData, edge, eventid),
                        ignore_index=True
                    )
                except:
                    print(eventid)

                try:
                    for edge2 in edge["node"]["availableListings"]["edges"]:
                        NewAvailableTickets = NewAvailableTickets.append(
                            GetTicketData(edge, edge2, eventid),
                            ignore_index=True
                        )
                except:
                    print(eventid)

                try:
                    for edge2 in edge["node"]["soldListings"]["edges"]:
                        NewSoldTickets = NewSoldTickets.append(
                            GetTicketData(edge, edge2, eventid),
                            ignore_index=True
                        )
                except:
                    print(eventid)

        else:
            PermanentData.loc[
                (PermanentData["EventID"] == eventid), "Status"
            ] = GetStatus2(EventData)

    PermanentData.to_feather("PermanentData.feather")

    ChangingData = (
        ChangingData.append(NewChangingData, ignore_index=True)
        .astype(str)
        .drop_duplicates(
            subset=[
                "EventID",
                "TypeID",
                "NumberOfAvailable",
                "NumberOfSold",
                "NumberOfAlerts",
                "Popular",
                "SoldOut",
            ],
            keep="first",
            ignore_index=True,
        )
    )
    ChangingData.to_feather("ChangingData.feather")

    AvailableTickets = (
        AvailableTickets.append(NewAvailableTickets, ignore_index=True)
        .astype(str)
        .drop_duplicates(
            subset=[
                "EventID",
                "TypeID",
                "ListingID",
                "SellerID",
                "TicketsInListing",
                "TicketsStillForSale",
                "OriginalPrice",
                "OriginalCurrency",
                "TotalPrice",
                "TotalCurrency",
                "SellerPrice",
                "SellerCurrency",
            ],
            keep="first",
            ignore_index=True,
        )
    )
    AvailableTickets.to_feather("AvailableTickets.feather")

    SoldTickets = (
        SoldTickets.append(NewSoldTickets, ignore_index=True)
        .astype(str)
        .drop_duplicates(
            subset=[
                "EventID",
                "TypeID",
                "ListingID",
                "SellerID",
                "TicketsInListing",
                "TicketsStillForSale",
                "OriginalPrice",
                "OriginalCurrency",
                "TotalPrice",
                "TotalCurrency",
                "SellerPrice",
                "SellerCurrency",
            ],
            keep="first",
            ignore_index=True,
        )
    )
    SoldTickets.to_feather("SoldTickets.feather")
```

The `ScrapeALot` function is used for running the whole webscraping process several times. It first calls the `GetNewEvents` function with the 'ANYTIME' setting (and the `CityID` and `IsHighlighted` arguments of our choice). Then it calls the `UpdateAll` function as well. Next, it stores the current date as `DateToday`. It then calls the `GetNewEvents` and `UpdateAll` functions again, changing the `Date` argument in `GetNewEvents` to the next day each time. The number of consequent days it should check (after the current one) can be specified using the `DaysToCheck` argument. Similarly, the `Turns` argument is for specifying how many times we want to repeat this whole process as described above.

```
def ScrapeALot(Turns, DaysToCheck, CityID, IsHighlighted, sess):
    for i in range(Turns):
        print("ANYTIME")
        GetNewEvents(CityID, "ANYTIME", IsHighlighted, sess)
        UpdateAll(sess)
        DateToday = datetime.datetime.today()
        for j in range(DaysToCheck):
            Date = (DateToday + timedelta(days=j)).strftime("%Y-%m-%d") + "T00:00:00Z"
            print(Date)
            GetNewEvents(CityID, Date, IsHighlighted, sess)
            UpdateAll(sess)
```

Finally, the webscraping is conducted simply buy calling the `ScrapeALot` function. We set both the `Turns` and `DaysToCheck` argument as 100, meaning that the programme will run the above process 100 times, each time checking the next 100 days after the current date. The chosen `CityID` argument is '`Q2l0eToz`', which is the ID of Amsterdam. We focus on events which are not highlighted (since highlighted events are very rare). The `sess` argument is simply a session of internet communication.

```
sess = requests.Session()
CityID = "Q2l0eToz"
IsHighlighted = False

ScrapeALot(100, 100, CityID, IsHighlighted, sess)
```

A.2 Tidying the TicketSwap dataset

In this appendix we include our codes for tidying the data scraped from TicketSwap. Using Python was more convenient for the webscraping process in appendix A.1, but from now on we switch to the R programming language (R Core Team, 2022). We first load the required R packages into our library.

```
library(arrow)
library(tidyverse)
library(data.table)
```

We open the four scraped datasets from the `.feather` files in our working directory. In all four cases, we use the `type_convert` function from the tidyverse package (Wickham et al., 2019) to automatically convert the types of variables. We also tell the function to interpret the `<NA>` and `None` strings as missing values (`NA`). In the `AvailableTickets` and `SoldTickets` datasets we create a new column (`Sold`), which simply indicates whether the current ticket listing is sold or not. In the case of `ChangingData`, we also create a new column (`TimeRecordedTicket`), which is the same as `TimeRecorded` for now. We then convert the `ChangingData` tibble to a data table. These modifications will be crucial for joining the four datasets into one.

```
PermanentData <-  read_feather("PermanentData.feather") %>%
  type_convert(na = c("<NA>", "None"), guess_integer = T)
ChangingData <-  read_feather("ChangingData.feather") %>%
  type_convert(na = c("<NA>", "None"), guess_integer = T) %>%
  mutate(TimeRecordedTicket = TimeRecorded) %>%
  setDT()
AvailableTickets <- read_feather("AvailableTickets.feather") %>%
  type_convert(na = c("<NA>", "None"), guess_integer = T) %>%
  mutate(Sold = F)
SoldTickets <- read_feather("SoldTickets.feather") %>%
  type_convert(na = c("<NA>", "None"), guess_integer = T) %>%
  mutate(Sold = T)
```

We bind the rows of `AvailableTickets` and `SoldTickets`, hence obtaining an integrated dataset of both available and sold ticket listings (the recently defined `Sold` variable tells us which is which). Then we left-join the `PermanentData` tibble to them (by the `EventID` and `TypeID` variables). This way, for each ticket listing we have the permanent data of the corresponding event in the same database. We convert this database to a data table and store it as `TicketSwapData`.

```
TicketSwapData <- bind_rows(AvailableTickets, SoldTickets) %>%
  left_join(PermanentData, by = c("EventID", "TypeID")) %>%
  setDT()
```

Joining `ChangingData` to `TicketSwapData` is a bit complicated. To each ticket listing in `TicketSwapData`, we need to join the most recent[47] row from `ChangingData` which contains data of the corresponding event. Such a conditional joining is not feasible in the `tidyverse` syntax, which is precisely why we have converted both datasets from tibbles to data tables. But even this way, we have to use the `setkey` function on both of them before joining.

[47]By 'most recent', we mean that the time we recorded the row in `ChangingData` must precede the time we recorded the ticket listing, but should also be as close to it as possible.

```
setkey(TicketSwapData, EventID, TypeID, TimeRecorded)
setkey(ChangingData, EventID, TypeID, TimeRecorded)
```

Once we have finished these modifications, we can finally join `ChangingData` to `TicketSwapData`. We then convert the result (which is still a data table) back to a tibble, and order it by the `EventID`, `TypeID`, `ListingID`, and `TimeRecordedTicket` variables. We redefine the `OriginalPrice`, `SellerPrice`, and `TotalPrice` variables (the TicketSwap API multiplied these prices by 100, so now we need to divide them by 100). We also create the `SellerPriceLimit` and `RealEnd` variables (see their description in table 3.1). The resulting dataset is stored under the same name (`TicketSwapData`).

```
TicketSwapData <-
  ChangingData[TicketSwapData,
               on = .(EventID,
                      TypeID,
                      TimeRecordedTicket <= TimeRecorded),
               mult = 'last'] %>%
  as_tibble() %>%
  arrange(EventID, TypeID, ListingID, TimeRecordedTicket) %>%
  mutate(
    OriginalPrice = OriginalPrice/100,
    SellerPrice = SellerPrice/100,
    TotalPrice = TotalPrice/100,
    SellerPriceLimit = ifelse(
      TotalCurrency == OriginalCurrency,
      ceiling(120*OriginalPrice)/100,
      NA
    ),
    RealEnd = ifelse(
      is.na(EndDate),
      StartDate,
      EndDate
    )
  )
```

When we defined the `RealEnd` variable, we used the `ifelse` function. As a consequence, its class was overwritten. However, we still want to treat these values as dates, so we need to convert them back.

```
class(TicketSwapData$RealEnd) <- class(TicketSwapData$TimeRecorded)
```

We started regularly scraping data from TicketSwap on 14 July 2022, 20:02. However, even before this date we had previous attempts of scraping some data. Back then, we were not running the code continuously, but we did run it from time to time. The observations from these previous attempts are also included in our datasets. However, it is crucial for our analysis that the data we use comes from a single (long) time window, and not from short separate ones. Therefore, we exclude the observations which were recorded before 14 July 2022, 20:02. To do this, we store the `TimeRecordedTicket` variable of the last such ticket listing as `border`.

```
border <- TicketSwapData %>%
  filter(ListingID == "TGlzdGluZzo3MDIxMjM3") %>%
  select(TimeRecordedTicket) %>%
  as.data.frame()
border <- border[1,1]
```

We now start filtering the `TicketSwapData` dataset. We first use `border` to exclude the listings recorded earlier than 14 July 2022, 20:02. Then we also filter out observations where the `SellerPrice` exceeds the 120% limit set by TicketSwap, as well as those which were made after the corresponding event had already ended.

```
TicketSwapData <- TicketSwapData %>%
  filter(TimeRecordedTicket > border) %>%
  filter(SellerPriceLimit >= SellerPrice) %>%
  filter(RealEnd >= TimeRecordedTicket)
```

We create the `firstseen` tibble. This tibble contains when each ticket listing was first recorded into our dataset.

```
firstseen <- TicketSwapData %>%
  group_by(ListingID) %>%
  summarise(FirstSeenAvailable = min(TimeRecordedTicket))
```

We left-join the `firstseen` tibble to the dataset (by `ListingID`), hence obtaining the `FirstSeenAvailable` column. We then create several other columns (see their description in table 3.1). We then order the remaining observations by the `TimeLeftNow` variable, group by `TypeID`, create the `RelativeTimeLeftNow` variable, and ungroup.

```
TicketSwapData <- TicketSwapData %>%
  left_join(firstseen,
            by = "ListingID") %>%
  mutate(
    NumberSoldNext = ifelse(
      lead(ListingID, default = "") == ListingID,
      ifelse(
        lead(TicketsInListing, default = 0) == TicketsInListing &
          lead(TicketsStillForSale, default = 0) < TicketsStillForSale,
        TicketsStillForSale - lead(TicketsStillForSale),
        0
      ),
      ifelse(
        lag(ListingID, default = "") != ListingID &
          Sold,
        TicketsInListing,
        TicketsStillForSale
      )
    ),
    PriceRatioNow = ifelse(
      TotalCurrency == OriginalCurrency,
      TotalPrice / OriginalPrice,
      NA
    ),
    NextPriceRatio = ifelse(
      lead(ListingID, default = "") == ListingID,
      ifelse(
        lead(TicketsInListing, default = 0) == TicketsInListing,
        lead(PriceRatioNow),
        NA
      ),
      PriceRatioNow
    ),
    NextSeen = ifelse(
      lead(ListingID, default = "") == ListingID,
      ifelse(
        lead(TicketsInListing, default = 0) == TicketsInListing,
        lead(TimeRecordedTicket),
        NA
      ),
      ifelse(
        Sold,
        TimeRecordedTicket,
        RealEnd
      )
    )
  )
```

Like in the case of RealEnd, we need to redefine the class of the NextSeen variable.

```
class(TicketSwapData$NextSeen) <- class(TicketSwapData$TimeRecorded)
```

We create the columns for the remaining variables in our dataset (see table 3.1).

```
TicketSwapData <- TicketSwapData %>%
  mutate(
    TimeLeftNow = as.numeric(RealEnd - TimeRecordedTicket),
    TimeLeftWhenAvailable = as.numeric(RealEnd - FirstSeenAvailable),
    TimeLeftNext = as.numeric(RealEnd - NextSeen),
    TimeAvailableFor = as.numeric(NextSeen - FirstSeenAvailable),
    EverSold = NextSeen != RealEnd)
```

We save the final TicketSwapData dataset as TidyTicketSwapData.RData.

```
save(TicketSwapData, file = "TidyTicketSwapData.RData")
```

A.3 Analysing the TicketSwap dataset

This appendix shows how we have conducted the analysis on the TicketSwap dataset (see section 3.3). We continue using the R programming language (R Core Team, 2022). We first load the required R packages and set the theme for our plots. Then we open the TidyTicketSwapData.RData file from the Webscraping folder of our working directory.

```
library(tidyverse)
library(stargazer)
library(tikzDevice)
library(ggrastr)
library(fixest)
library(lmtest)

theme_set(
  theme_bw()+
    theme(axis.title.y = element_text(face = "italic"),
          axis.title.x = element_text(face = "italic"),
          legend.title = element_text(face = "italic"),
          text = element_text(size = 8),
          panel.grid.major = element_blank(),
          panel.grid.minor = element_blank(),
          panel.background = element_blank(),
          plot.background = element_blank(),
          legend.background = element_blank(),
          legend.key = element_blank())
)

load("Webscraping/TidyTicketSwapData.RData")
```

We save some of the basic attributes (the number of observations, the number of events, etc.) of our dataset as `.tex` files into the `Paper` folder of our working directory (we will include these in our document later).

```
TicketSwapData %>%
  nrow() %>%
  cat(file = "Paper/Observations.tex")

TicketSwapData$EventID %>%
  unique() %>%
  length() %>%
  cat(file = "Paper/Events.tex")

TicketSwapData$TypeID %>%
  unique() %>%
  length() %>%
  cat(file = "Paper/EventTypes.tex")

TicketSwapData$SellerID %>%
  unique() %>%
  length() %>%
  cat(file = "Paper/Sellers.tex")

TicketSwapData$ListingID %>%
  unique() %>%
  length() %>%
  cat(file = "Paper/Listings.tex")

TicketSwapData$NumberSoldNext %>%
  sum() %>%
  cat(file = "Paper/Tickets.tex")

TicketSwapData$TimeRecordedTicket %>%
  min() %>%
  as.character('%d %B %Y, %H:%M') %>%
  gsub(pattern = "^0",
       replacement = "") %>%
  cat(file = "Paper/Firstobs.tex")

TicketSwapData$TimeRecordedTicket %>%
  max() %>%
  as.character('%d %B %Y, %H:%M') %>%
  gsub(pattern = "^0",
       replacement = "") %>%
  cat(file = "Paper/Lastobs.tex")
```

We create the unweighted TicketSwap dataset using the `uncount` function from the `tidyverse` package (Wickham et al., 2019).

```
UnweightedTicketSwapData <- TicketSwapData %>%
  uncount(weights = NumberSoldNext,
          .remove = F)
```

We store the names of our numeric variables (inside the `\textit{}` environment) as `varnames`. We then create table 3.2 about the summary statistics of the unweighted dataset and save it as `summarystats.tex` into the `Paper` folder of our working directory.

```
varnames <- UnweightedTicketSwapData %>%
  select_if(is.numeric) %>%
  colnames()

varnames <- paste("\\textit{", varnames, "}", sep = "")

UnweightedTicketSwapData %>%
  as.data.frame() %>%
  stargazer(style="aer",
            summary.logical = F,
            digit.separator = "$\\,$",
            digit.separate = 3,
            digits=2,
            float = F,
            omit.summary.stat = c("p25", "p75"),
            header=F,
            digits.extra = 2,
            font.size = "footnotesize",
            no.space = F,
            align=F,
            covariate.labels = varnames,
            median = T) %>%
  cat(file = "Paper/summarystats.tex")
```

We also save the number of tickets that never ended up being sold as `NeverSold.tex` into the `Paper` folder of our working directory.

```
UnweightedTicketSwapData %>%
  filter(!EverSold) %>%
  nrow() %>%
  cat(file = "Paper/NeverSold.tex")
```

We create figures 3.2, 3.3 and 3.4, and save them as `densityunweighted.tex`, `figure1.tex` and `figure2.tex` into the `Paper` folder of our working directory (respectively).

```
tikz('Paper/densityunweighted.tex', width = 6, height = 3)
densityunweighted <- UnweightedTicketSwapData %>%
  ggplot(aes(x = PriceRatioNow)) +
  geom_density() +
  theme(axis.title.y = element_blank())+
  scale_x_continuous(breaks = c(0, 0.5, 1, 1.0815, 1.0815/0.95, 1.2978),
                     minor_breaks = c(),
                     labels = function(x) round(x, digits = 3))+
  geom_vline(xintercept = c(1, 1.0815, 1.0815/0.95, 1.2978),
             linetype = "dotted")
print(densityunweighted)
dev.off()
```

```
tikz('Paper/figure1.tex', width = 6, height = 4)
figure1 <- UnweightedTicketSwapData %>%
  ggplot(aes(x = -TimeLeftNow,
             y = PriceRatioNow,
             col = EverSold)) +
  geom_point_rast(alpha = 0.2,
                  shape = ".",
                  dev = "ragg",
                  raster.dpi = 350) +
  guides(color = guide_legend(override.aes = list(shape = 16,
                                                  alpha = 1)))
print(figure1)
dev.off()

tikz('Paper/figure2.tex', width = 6, height = 4)
figure2 <- UnweightedTicketSwapData %>%
  ggplot(aes(x = -TimeLeftNow,
             y = PriceRatioNow,
             col = EverSold)) +
  geom_point_rast(alpha = 0.2,
                  shape = ".",
                  dev = "ragg",
                  raster.dpi = 350) +
  guides(color = guide_legend(override.aes = list(shape = 16,
                                                  alpha = 1))) +
  coord_cartesian(xlim = c(-5000000,0))
print(figure2)
dev.off()
```

We set the dictionary of the `fixest` package (Bergé, 2018), in order to make the table display the variable names in the italic form (inside the `\textit{}` environment). Then we create the `ReplaceThings` function, which will perform some additional modifications on the `.tex` file containing table 3.3. Next, we use the `feols` function from the `fixest` package (Bergé, 2018) to run the seven regression models in table 3.3[48]. Finally, we include these models in a LaTeX table and save it as `FEmodelsTicketSwap.tex` into the `Paper` folder of our working directory.

```
paste("\\textit{",
      colnames(UnweightedTicketSwapData),
      "}",
      sep = "") %>%
  setNames(colnames(UnweightedTicketSwapData)) %>%
  setFixest_dict()

ReplaceThings <- function(Latexoutput){
  Latexoutput %>%
    gsub(pattern = "\\\\times",
         replacement = "\\\\cdot") %>%
    gsub(pattern = "xxxx",
         replacement = "$\\\\times$") %>%
    gsub(pattern = "--",
         replacement = "$-$") %>%
    gsub(pattern = "& -",
         replacement = "& $-$") %>%
```

[48]After creating model (1), we perform the Breush-Pagan test (Breusch & Pagan, 1979) on it. But since the `fixest` package does not support Breusch-Pagan tests, we need to use the `lm` function from the `base` package (R Core Team, 2022) in this case. Nevertheless, the results are the same.

```r
    gsub(pattern = ",",
         replacement = "$\\\\,$") %>%
    gsub(pattern = "$\\\\,$ ",
         replacement = ", ") %>%
    gsub(pattern = " '",
         replacement = " `") %>%
    gsub(pattern = "\\\\multicolumn\\{6\\}\\{c\\}\\{Heteroskedasticity-robust\\}",
         replacement = "HC & HC & HC & HC & HC & HC")%>%
    gsub(pattern = "\\\\multicolumn\\{8\\}\\{l\\}",
         replacement = "\\\\multicolumn\\{8\\}\\{c\\}")
}

model1 <- feols(
  PriceRatioNow ~ TimeLeftNow,
  data = UnweightedTicketSwapData,
  vcov = "iid")

lm(PriceRatioNow ~ TimeLeftNow,
   data = UnweightedTicketSwapData) %>%
  bptest()

model2 <- feols(
  PriceRatioNow ~ TimeLeftNow,
  data = UnweightedTicketSwapData,
  vcov = "HC1")

model3 <- feols(
  PriceRatioNow ~ TimeLeftNow+
    NumberOfAvailable +
    NumberOfSold +
    NumberOfAlerts,
  data = UnweightedTicketSwapData,
  se = "HC1")

model4 <- feols(
  PriceRatioNow ~ TimeLeftNow+
    NumberOfAvailable +
    NumberOfSold +
    NumberOfAlerts | TypeID,
  data = UnweightedTicketSwapData,
  vcov = "HC1")

model5 <- feols(
  PriceRatioNow ~ TimeLeftNow+
    NumberOfAvailable +
    NumberOfAlerts | TypeID,
  data = UnweightedTicketSwapData,
  vcov = "HC1")

model6 <- feols(
  PriceRatioNow ~ TimeLeftNow+
    NumberOfAvailable +
    NumberOfSold +
    NumberOfAlerts +
    i(SoldOut)| TypeID,
  data = UnweightedTicketSwapData,
  vcov = "HC1")
```

```
model7 <- feols(
  PriceRatioNow ~ TimeLeftNow+
    NumberOfAvailable +
    NumberOfAlerts +
    i(SoldOut)| TypeID,
  data = UnweightedTicketSwapData,
  vcov = "HC1")

etable(model1, model2, model3, model4, model5, model6, model7,
       digits = "s1",
       digits.stats = "r4",
       tex=T,
       fitstat = c('n', "g", "r2", "ar2", "wr2", "f.p"),
       se.row=T,
       file = "Paper/FEmodelsTicketSwap.tex",
       replace = T,
       fixef_sizes = T,
       fixef_sizes.simplify = F,
       style.tex = style.tex(main = "aer",
                             line.top = "double",
                             yesNo = c("Yes","No"),
                             fixef.where = "var",
                             fontsize = "footnotesize",
                             interaction.combine = "xxxx",
                             fixef_sizes.prefix = "Number of ",
                             fixef_sizes.suffix = "s",
                             tablefoot = T,
                             tablefoot.value = "default"),
       poly_dict = c("", "$^2$", "$^3$"),
       postprocess.tex = ReplaceThings,
       coef.just =  ".")
```

A.4 Simulations

In this appendix we show how we run our simulations and create the two artificial datasets. Like in the previous sections, we use R (R Core Team, 2022). We first load the required R packages into our library.

```
library(tidyverse)
library(partitions)
library(matrixStats)
```

These codes are designed under the assumption that we already have some data saved in `.RData` files, and we want to append additional rows to them[49]. Therefore, we first need to open these datasets from our working directory.

```
load("ArtificialDataWithConstraints.RData")
load("ArtificialDataWithoutConstraints.RData")
```

The `ExpectedUtility1` function calculates a secondary seller's expected utility when she undercuts all other sellers on the platform. In our model we referred to this case as Case I, and dealt with it in subsection 4.2.1. The arguments of `ExpectedUtility1` are the chosen price of our seller (here we call it `MyPrice`, in our model it was denoted by p), the current number of previously subscribed buyers (`NumberOfAlerts` or

[49]This of course means that our first codes were slightly different.

n), the current smallest price (`MinPrice` or p_1), the time left until the event (`TimeLeft` or t), our seller's risk aversion parameter (`alpha` or α), the frequency of buyer arrivals (`lambda` or λ), and the share that a seller gets from the total price (`SellerShare` or $1-\tau$). Using equations (4.4) and (4.7) with the given arguments, the function simply returns the seller's expected utility.

```
ExpectedUtility1 <- function(MyPrice,
                             NumberOfAlerts,
                             MinPrice,
                             TimeLeft,
                             alpha,
                             lambda,
                             SellerShare) {
  (1 -
     exp(lambda * TimeLeft * (MyPrice - 1)) * (MyPrice / MinPrice) ^ NumberOfAlerts
  ) * (MyPrice * SellerShare) ^ alpha
}
```

The `ExpectedUtility2` function calculates a secondary seller's expected utility, should she choose a price above the current lowest one that is not equal to any other previously submitted price. The formula for the seller's expected utility in this case (referred to as Case II, see subsection 4.2.2) is given by equations (4.4) and (4.13). The `MyPrice`, `TimeLeft`, `alpha`, `lambda`, and `SellerShare` arguments in this function are defined in the same way as for `ExpectedUtility1`. The only 'new' argument is `SmallerPrices`, which is a vector containing the prices of tickets currently available for a cheaper price than our seller's. In our model this vector was $(p_1, p_2, ..., p_J)$.

The function first counts the elements in the `SmallerPrices` vector, thus obtaining the J parameter from our model (stored as `J`). Next, it uses the Poisson distribution to calculate the probability that less than `J` arrivals happen, and stores this value as `ProbOfNoSell`. When the number of arrivals (`k` or k) exceeds `J`, it is impossible to consider all possibilities because `k` has no upper limit: any number of arrivals may occur with a positive probability. We solve this issue by only considering values of `k` below the 99th percentile of the Poisson distribution (this results in a negligible numerical error). In the function we run a `for` loop on such values of `k`. If this `k` is over `J` (we have already dealt with values below `J`), we run another for `for` loop inside the previous one. In this loop we calculate the probability that a given number of transactions (`m` or m) occur, for each value of `m` below `J`. This is done in the way described in subsection 4.2.2, using the `compositions` function from the `partitions` package (Hankin, 2006). We store these probabilities as `mProbOfNoSell` and add them to `ProbOfNoSell`[50]. Finally, the function returns the expected utility.

```
ExpectedUtility2 <-function(MyPrice,
                            SmallerPrices,
                            TimeLeft,
                            alpha,
                            lambda,
                            SellerShare) {
  J <- length(SmallerPrices)
  ProbOfNoSell <- ppois(J, lambda = lambda * TimeLeft)
  for (k in 0:qpois(p = 0.99, lambda = lambda * TimeLeft)) {
    if (k > J) {
      for (m in 0:J) {
        SmallestPrices <- c(SmallerPrices, MyPrice) %>% head(n = m + 1)
        mProbOfNoSell <-
          (SmallestPrices ^ compositions(n = k - m,
```

[50]This way, the probability from equation (4.13) can be easily obtained by subtracting `ProbOfNoSell` from 1.

```
                                              m = m + 1,
                                              include.zero = T)) %>%
          colProds() %>%
          sum()
        mProbOfNoSell <- mProbOfNoSell *
          prod(1 - SmallestPrices[-(m+1)]) *
          dpois(x = k, lambda = lambda * TimeLeft)
        ProbOfNoSell <- ProbOfNoSell + mProbOfNoSell
      }
    }
  }
  return((1 - ProbOfNoSell) * (MyPrice * SellerShare) ^ alpha)
}
```

The ExpectedUtility3 function calculates a secondary seller's expected utility when she sets a price that is exactly equal to one (or more) of the previously submitted prices. In our model this is called Case III (see subsection 4.2.3), and the formula for the expected utility is given by equations (4.4) and (4.16). The arguments of the ExpectedUtility3 function are the same as the ExpectedUtility2 function's arguments, the only exception is NumberOfIdenticalPrices. This argument shows how many other tickets have the same price as our seller's (in our model this value was denoted by d).

The ExpectedUtility3 function uses the previously defined ExpectedUtility2 function to make the algorithm shorter. To calculate the probability that the ticket remains unsold (conditional on it being on a certain place on the preference list of buyers, and 0 new sellers arriving), it calls the ExpectedUtility2 function with the same arguments. Then it does the inverse of the steps in ExpectedUtility2 to obtain the ProbOfNoSell variable. This is then divided by NumberOfIdenticalPrices+1, like in equation (4.16). This process is first conducted on the original SmallerPrices vector (which was given as an argument). Then the function starts adding MyPrice as an additional element to this vector in a for loop (NumberOfIdenticalPrices times), each time repeating the above process and adding the result to ProbOfNoSell. Finally, the function uses this ProbOfNoSell to return the expected utility of a seller.

```
ExpectedUtility3 <-function(MyPrice,
                            SmallerPrices,
                            TimeLeft,
                            alpha,
                            lambda,
                            SellerShare,
                            NumberOfIdenticalPrices) {
  ProbOfNoSell <-
    (1 - ExpectedUtility2(MyPrice,
                          SmallerPrices,
                          TimeLeft,
                          alpha,
                          lambda,
                          SellerShare) / ((MyPrice * SellerShare) ^ alpha)) /
    (NumberOfIdenticalPrices + 1)

  if (NumberOfIdenticalPrices > 0) {
    for (i in 1:NumberOfIdenticalPrices) {
      SmallerPrices <- c(SmallerPrices, MyPrice)

      ProbOfNoSell <- ProbOfNoSell +
        (1 - ExpectedUtility2(MyPrice,
                              SmallerPrices,
                              TimeLeft,
                              alpha,
                              lambda,
                              SellerShare) / ((MyPrice * SellerShare) ^ alpha)) /
        (NumberOfIdenticalPrices + 1)
    }
  }
  return((1 - ProbOfNoSell) * (MyPrice * SellerShare) ^ alpha)
}
```

The `SimulateNewEvent` function is for simulating an event's ticket market. Its arguments are the event's ID (`EventID`), the frequency of buyer (`lambda` or λ) and seller (`mu` or μ) arrivals, the number of available tickets on the primary market at time zero (`NumberOfPrimaryTickets` or N_p), the price on the primary market (`OriginalPrice` or p_p), the lower price limit on the secondary market (`LowerLimit` or $\underline{p}$), the effective upper price limit (`UpperLimit` or $\min\{\overline{p}, 1\}$), the share that a secondary seller gets from the total price (`SellerShare` or $1 - \tau$), and two vectors containing the timing of buyer (`BuyerArrivals`) and seller arrivals (`SellerArrivals`).

In the function we first create a tibble (`NewEvent`) with the same columns as our artificial dataset. This tibble will contain the data of each primary ticket, buyer and secondary seller on this event's ticket market. Where it is possible, we already fill in the values (e.g. we randomise the reservation prices of buyers and the risk aversion parameters of sellers), but most of the columns remain empty for now. We drop the buyers and sellers who arrive only after the event (i.e. whose `TimeLeftNow` is negative), and order the remaining observations by `TimeLeftNow` (in descending order). Next, we go through the agents (buyers and secondary sellers alike) one by one, in the order of their arrivals.

If the current agent is a buyer, we select the currently available tickets which are cheaper than her reservation price. Out of these, we choose the cheapest one (if more tickets have the cheapest price, we pick one randomly). We store the price and ID of this ticket as `Match`. If `Match` is not empty, we have indeed found a match for the current buyer. Hence we fill the corresponding columns (`Matched`, `MatchedWith`, `TimeLeftWhenMatched`, and `Price`) for both the buyer and the agent she is matched to in the `NewEvent` tibble.

If the current agent is a seller, we first store her risk aversion parameter as `alpha`. Next, we select the buyers subscribed for alerts when she arrives, and store their data as `Alerts`. We also select the currently available tickets (on both the primary and secondary markets) and store them as `Available`. Similarly, in `AvailableSecondary` we store the data of currently available tickets, but this time only from the secondary market. If the `Available` tibble is not empty, we create a (sorted) vector containing the prices of these available tickets called `Prices`. The first value in this vector (which is hence the lowest price) is stored as well, in the `NextPrice` variable. But when the `Available` tibble is empty, the `Prices` vector is also left empty, and `NextPrice` is defined as being equal to `UpperLimit`.

Once we obtain `Prices` and `NextPrice`, we can start modelling the seller's pricing decision. First we numerically maximise[51] the `ExpectedUtility1` function in `MyPrice`, on the interval between `LowerLimit` and `NextPrice`. This way we find the optimal price in Case I (see subsection 4.2.1). We store this price and the corresponding value of the objective function in the `Optima` tibble. Next (if the `Prices` vector is not empty), we run a `for` loop, always reassigning `NextPrice` as the next unique value in `Prices`[52]. Each time, we create a vector (`SmallerPrices`) of the prices below `NextPrice`. If this vector is not empty, we numerically maximise the `ExpectedUtility2` function on the interval between `max(SmallerPrices)` and `NextPrice`. We record each optimal price and local maximum in the `Optima` tibble. This way we find all possible maxima in Case II (see subsection 4.2.2). After this, we run a `for` loop on the unique

[51] All numerical maximisations are carried out using the `optimise` function from the `stats` package (R Core Team, 2022).
[52] In the last turn, we assign the value of `UpperLimit` to `NextPrice`.

prices in `Prices` (and `UpperLimit`) again. Each time, we create the `SmallerPrices` vector like before, and also count the prices in `Prices` that are equal to the current price. This latter number is stored as `NumberOfIdenticalPrices`. Using these arguments (among others), we call the `ExpectedUtility3` function and store its result in `Optima`. This way we also cover Case III (see subsection 4.2.3). Finally, we just check which row has the largest objective function value in `Optima`. The corresponding price will be the current seller's choice. This allows us to fill the current seller's row in the `NewEvent` tibble. We also check whether there are any previously subscribed buyers who accept a ticket for this price. If there are, one of them (picked randomly) will be matched to our current seller.

Once it has gone through all agents and filled in their missing data, the `SimulateNewEvent` function simply returns the `NewEvent` tibble.

```
SimulateNewEvent <- function(EventID,
                             lambda,
                             mu,
                             NumberOfPrimaryTickets,
                             OriginalPrice,
                             LowerLimit,
                             UpperLimit,
                             SellerShare,
                             BuyerArrivals,
                             SellerArrivals){
  NewEvent <- tibble(
    Arrival = c(rep(0, NumberOfPrimaryTickets),
                BuyerArrivals,
                SellerArrivals),
    TimeLeftNow = 1 - c(rep(0, NumberOfPrimaryTickets),
                        cumsum(BuyerArrivals),
                        cumsum(SellerArrivals)),
    Buyer = c(rep(F, NumberOfPrimaryTickets),
              rep(T, length(BuyerArrivals)),
              rep(F, length(SellerArrivals))),
    ReservationPrice = c(rep(NA, NumberOfPrimaryTickets),
                         runif(length(BuyerArrivals)),
                         rep(NA, length(SellerArrivals))),
    alpha = c(rep(NA, NumberOfPrimaryTickets),
              rep(NA, length(BuyerArrivals)),
              runif(length(SellerArrivals))),
    Price = c(rep(OriginalPrice, NumberOfPrimaryTickets),
              rep(NA, length(BuyerArrivals)),
              rep(NA, length(SellerArrivals))),
    Objective = NA,
    Matched = F,
    MatchedWith = NA,
    TimeLeftWhenMatched = NA,
    lambda = lambda,
    mu = mu,
    NumberOfPrimaryTickets = NumberOfPrimaryTickets,
    OriginalPrice = OriginalPrice,
    LowerLimit = LowerLimit,
    UpperLimit = UpperLimit,
    EventID = EventID,
    NumberOfAvailable = NA,
    NumberOfMatched = NA,
    NumberOfAlerts = NA,
    AvgPrice = NA,
    MinPrice = NA,
```

```r
    NumberOfAvailableSecondary = NA,
    NumberOfMatchedSecondary = NA,
    AvgPriceSecondary = NA,
    MinPriceSecondary = NA
  ) %>%
    filter(TimeLeftNow >= 0) %>%
    arrange(-TimeLeftNow) %>%
    mutate(ID = row_number())

  if (nrow(NewEvent)>NumberOfPrimaryTickets) {
    for (Arrival in (NumberOfPrimaryTickets + 1):nrow(NewEvent)) {
      TimeLeft <- NewEvent$TimeLeftNow[Arrival]

      if (NewEvent$Buyer[Arrival]) {
        Match <- NewEvent %>%
          filter(!Matched &
                   !Buyer &
                   TimeLeftNow > TimeLeft) %>%
          select(ID, Price) %>%
          filter(Price < NewEvent$ReservationPrice[Arrival]) %>%
          slice_min(order_by = Price, n = 1) %>%
          slice_sample(n=1)

        if (nrow(Match) > 0){
          NewEvent$Matched[Arrival] <- T
          NewEvent$MatchedWith[Arrival] <- Match$ID
          NewEvent$TimeLeftWhenMatched[Arrival] <- NewEvent$TimeLeftNow[Arrival]
          NewEvent$Price[Arrival] <- Match$Price
          NewEvent$Matched[Match$ID] <- T
          NewEvent$MatchedWith[Match$ID] <- Arrival
          NewEvent$TimeLeftWhenMatched[Match$ID] <- NewEvent$TimeLeftNow[Arrival]
        }
      }
      else {
        alpha <- NewEvent$alpha[Arrival]

        Alerts <- NewEvent %>%
          filter(!Matched &
                   Buyer &
                   TimeLeftNow > TimeLeft)

        Available <- NewEvent %>%
          filter(!Matched &
                   !Buyer &
                   TimeLeftNow > TimeLeft)

        AvailableSecondary <- Available %>%
          filter(TimeLeftNow < 1)

        if(nrow(Available) > 0){
          Prices <- Available$Price %>% sort()
          NextPrice <- Prices[1]
        }
        else{
          Prices <- NULL
          NextPrice <- UpperLimit
        }
        Optimum <- optimise(ExpectedUtility1,
                            interval = c(LowerLimit,
                                         NextPrice),
                            tol = 0.001,
                            maximum = T,
                            NumberOfAlerts = nrow(Alerts),
                            MinPrice = NextPrice,
                            TimeLeft = TimeLeft,
                            alpha = alpha,
                            lambda = lambda,
                            SellerShare = SellerShare)

        Optima <- tibble(Price = Optimum$maximum,
                         Objective = Optimum$objective)

        if (length(Prices)>0){
          for (NextPrice in unique(c(Prices,UpperLimit))[-1]) {
            SmallerPrices <- Prices[Prices < NextPrice]

            if (length(SmallerPrices)>0){
              Optimum <- optimise(ExpectedUtility2,
                                  interval = c(max(SmallerPrices),
                                               NextPrice),
                                  tol = 0.001,
                                  maximum = T,
                                  SmallerPrices = SmallerPrices,
                                  TimeLeft = TimeLeft,
                                  alpha = alpha,
                                  lambda = lambda,
                                  SellerShare = SellerShare)

              Optima <- bind_rows(
                Optima,
                tibble(Price = Optimum$maximum,
                       Objective = Optimum$objective)
              )
            }
          }

          for (NextPrice in unique(c(Prices,UpperLimit))) {
            SmallerPrices <- Prices[Prices < NextPrice]
            NumberOfIdenticalPrices <- Prices[Prices == NextPrice] %>%
              length()

            Optima <- bind_rows(
              Optima,
              tibble(Price = NextPrice,
                     Objective =
                       ExpectedUtility3(NextPrice,
                                        SmallerPrices,
                                        TimeLeft,
                                        alpha,
                                        lambda,
                                        SellerShare,
                                        NumberOfIdenticalPrices))
            )
          }
        }
```

```r
        Optimum <- Optima %>%
          slice_max(order_by = Objective,
                    n = 1,
                    with_ties = F)

        NewEvent$Price[Arrival] <- Optimum$Price
        NewEvent$Objective[Arrival] <- Optimum$Objective
        NewEvent$NumberOfAvailable[Arrival] <- nrow(Available)
        NewEvent$NumberOfMatched[Arrival] <- sum(NewEvent$Matched)/2
        NewEvent$NumberOfAlerts[Arrival] <- nrow(Alerts)
        NewEvent$AvgPrice[Arrival] <- mean(Prices)
        NewEvent$MinPrice[Arrival] <- min(Prices)
        NewEvent$NumberOfAvailableSecondary[Arrival] <- nrow(AvailableSecondary)
        NewEvent$NumberOfMatchedSecondary[Arrival] <- NewEvent %>%
          filter(TimeLeftNow < 1) %>%
          filter(Matched==T) %>%
          filter(!Buyer) %>%
          nrow()
        NewEvent$AvgPriceSecondary[Arrival] <- mean(AvailableSecondary$Price)
        NewEvent$MinPriceSecondary[Arrival] <- min(AvailableSecondary$Price)

        Match <- Alerts %>%
          filter(ReservationPrice > Optimum$Price) %>%
          slice_sample(n = 1)

        if (nrow(Match) > 0){
          NewEvent$Matched[Arrival] <- T
          NewEvent$MatchedWith[Arrival] <- Match$ID
          NewEvent$TimeLeftWhenMatched[Arrival] <- NewEvent$TimeLeftNow[Arrival]
          NewEvent$Matched[Match$ID] <- T
          NewEvent$MatchedWith[Match$ID] <- Arrival
          NewEvent$TimeLeftWhenMatched[Match$ID] <- NewEvent$TimeLeftNow[Arrival]
          NewEvent$Price[Match$ID] <- Optimum$Price
        }
      }
    }
    return(NewEvent)
  }
}
```

Now that we have defined all the functions we need, we can finally start running the simulations. We simulate the ticket markets of many different events using a `for` loop. The IDs of these events are simply consecutive natural numbers. Since we already have some data stored in `ArtificialDataWithConstraints` and `ArtificialDataWithoutConstraints`, we cannot start numbering the events from 1, we need to start from the next number in the row. Hence we first store the maximal `EventID` as `Start`, and continue the `for` loop from there.

Inside the `for` loop, we start simulating a certain ticket market by randomising the values of `lambda` (λ), `mu` (μ), `OriginalPrice` (p_p), and `NumberOfPrimaryTickets` (N_p). These randomisations are carried out according to the distribution functions in equations (??), (??), (??), and (??). Next, we also randomise the two Poisson processes of buyer and seller arrivals. We make use of the fact that in a Poisson process, the amount of time between two consecutive arrivals follows an exponential distribution with the same parameter. This means that we just have to combine random exponential variables until their sum reaches 1, and these will be the arrival times of our buyers/sellers.

Once we are done with these randomisations, we use the `SimulateNewEvent` function to simulate the decisions of our agents. We first consider the constrained specification (see section 5.1) and assign the values of `SellerShare`, `LowerLimit` and `UpperLimit` accordingly. We then call the `SimulateNewEvent` function with these freshly defined arguments and bind its result to `ArtificialDataWithConstraints`. Then we turn to the unconstrained specification (see section 5.1) and reassign the values of `SellerShare`, `LowerLimit` and `UpperLimit`. We call the `SimulateNewEvent` function again, this time with the new arguments. We append the resulting tibble to `ArtificialDataWithoutConstraints`.

```r
Start <- max(ArtificialDataWithConstraints$EventID)

for (EventID in (Start+1):(Start+1000)) {
  print(EventID)

  lambda <- runif(1, min = 0, max = 20)
  mu <- runif(1, min = 0, max = 15)
  NumberOfPrimaryTickets <- sample(x=0:5, size = 1)
  OriginalPrice <- runif(1)

  BuyerArrivals <- NULL
  while (sum(BuyerArrivals) < 1) {
    BuyerArrivals <- c(BuyerArrivals,
                       rexp(1, rate = lambda))
  }

  SellerArrivals <- NULL
  while (sum(SellerArrivals) < 1) {
    SellerArrivals <- c(SellerArrivals,
                        rexp(1, rate = mu))
  }

  SellerShare <- 0.95 / 1.05 / 1.03
  LowerLimit <- runif(1, min = 0, max = OriginalPrice)
  UpperLimit <- min(OriginalPrice * 1.2 * 1.05 * 1.03, 1)

  ArtificialDataWithConstraints <-
    bind_rows(
      ArtificialDataWithConstraints,
      SimulateNewEvent(
        EventID,
        lambda,
        mu,
        NumberOfPrimaryTickets,
        OriginalPrice,
        LowerLimit,
        UpperLimit,
        SellerShare,
        BuyerArrivals,
        SellerArrivals
      )
    )

  SellerShare <- 1
  LowerLimit <- 0
  UpperLimit <- 1

  ArtificialDataWithoutConstraints <-
    bind_rows(
      ArtificialDataWithoutConstraints,
      SimulateNewEvent(
        EventID,
        lambda,
        mu,
        NumberOfPrimaryTickets,
        OriginalPrice,
        LowerLimit,
        UpperLimit,
        SellerShare,
        BuyerArrivals,
        SellerArrivals
      )
    )
}
```

Before saving the two datasets, we first need to make some slight modifications. In R, when the `mean` and `min` functions are called on an empty array, they return non-finite values. For convenience, we redefine these non-finite values as missing data (`NA`) where necessary. We also calculate the `PriceRatio`, `AvgPriceRatio`, `MinPriceRatio`, `AvgPriceRatioSecondary`, `MinPriceRatioSecondary`, and `SoldOut` variables in both datasets. Only after these modifications do we save the two datasets as `.RData` files in our working directory.

```r
ArtificialDataWithConstraints <- ArtificialDataWithConstraints %>%
  mutate(
    AvgPrice = ifelse(is.finite(AvgPrice),
                      AvgPrice,
                      NA),
    MinPrice = ifelse(is.finite(AvgPrice),
                      MinPrice,
                      NA),
    AvgPriceSecondary = ifelse(is.finite(AvgPriceSecondary),
                               AvgPrice,
                               NA),
    MinPriceSecondary = ifelse(is.finite(AvgPriceSecondary),
                               MinPrice,
                               NA),
    PriceRatio = Price / OriginalPrice,
    AvgPriceRatio = AvgPrice / OriginalPrice,
    MinPriceRatio = MinPrice / OriginalPrice,
    AvgPriceRatioSecondary = AvgPriceSecondary / OriginalPrice,
    MinPriceRatioSecondary = MinPriceSecondary / OriginalPrice,
    SoldOut = NumberOfAvailable == NumberOfAvailableSecondary
  )

save(ArtificialDataWithConstraints,
     file = "ArtificialDataWithConstraints.RData")

ArtificialDataWithoutConstraints <- ArtificialDataWithoutConstraints %>%
  mutate(
    AvgPrice = ifelse(is.finite(AvgPrice),
                      AvgPrice,
                      NA),
    MinPrice = ifelse(is.finite(AvgPrice),
                      MinPrice,
                      NA),
    AvgPriceSecondary = ifelse(is.finite(AvgPriceSecondary),
                                AvgPrice,
                                NA),
    MinPriceSecondary = ifelse(is.finite(AvgPriceSecondary),
                                MinPrice,
                                NA),
    PriceRatio = Price / OriginalPrice,
    AvgPriceRatio = AvgPrice / OriginalPrice,
    MinPriceRatio = MinPrice / OriginalPrice,
    AvgPriceRatioSecondary = AvgPriceSecondary / OriginalPrice,
    MinPriceRatioSecondary = MinPriceSecondary / OriginalPrice,
    SoldOut = NumberOfAvailable == NumberOfAvailableSecondary
  )

save(ArtificialDataWithoutConstraints,
     file = "ArtificialDataWithoutConstraints.RData")
```

A.5 Analysing the artificial datasets

This appendix includes our R codes (R Core Team, 2022) for analysing the two artificial datasets resulting from our simulations. We first load the required R packages and set the theme for our plots. Then we open the `ArtificialDataWithConstraints.RData` and `ArtificialDataWithoutConstraints.RData` files from the `Simulations` folder of our working directory.

```
library(tidyverse)
library(stargazer)
library(tikzDevice)
library(ggrastr)
library(fixest)
library(lmtest)

theme_set(
  theme_bw()+
    theme(axis.title.y = element_text(face = "italic"),
          axis.title.x = element_text(face = "italic"),
          legend.title = element_text(face = "italic"),
          text = element_text(size = 8),
          panel.grid.major = element_blank(),
          panel.grid.minor = element_blank(),
          panel.background = element_blank(),
          plot.background = element_blank(),
          legend.background = element_blank(),
          legend.key = element_blank())
)

load("Simulations/ArtificialDataWithConstraints.RData")
load("Simulations/ArtificialDataWithoutConstraints.RData")
```

We save some of the basic attributes (the number of observations, the number of events, etc.) of our datasets as `.tex` files into the `Paper` folder of our working directory (we will include these in our document later).

```
ArtificialDataWithConstraints %>%
  nrow() %>%
  cat(file = "Paper/ArtObs.tex")
```

```r
ArtificialDataWithConstraints$Buyer %>%
  sum() %>%
  cat(file = "Paper/ArtBuyers.tex")

(ArtificialDataWithConstraints$TimeLeftNow == 1) %>%
  sum() %>%
  cat(file = "Paper/ArtPrimSellers.tex")

ArtificialDataWithConstraints$alpha %>%
  na.omit() %>%
  length() %>%
  cat(file = "Paper/ArtSecSellers.tex")

ArtificialDataWithConstraints$EventID %>%
  unique() %>%
  length() %>%
  cat(file = "Paper/ArtEvents.tex")
```

We filter the two datasets, keeping only the observations on secondary sellers. These filtered datasets are stored as `OnlySecSellersConstrained` and `OnlySecSellersUnconstrained`.

```r
OnlySecSellersConstrained <- ArtificialDataWithConstraints %>%
  filter(is.finite(alpha))

OnlySecSellersUnconstrained <- ArtificialDataWithoutConstraints %>%
  filter(is.finite(alpha))
```

We store the names of our numeric variables (inside the \texttt{\textit{}} environment) as `varnnames`. We then create tables 5.2 and 5.3 about the summary statistics of the two filtered datasets and save them as `ArtWithsummarystats.tex` and `ArtWithoutsummarystats.tex` into the `Paper` folder of our working directory.

```r
varnames <- OnlySecSellersConstrained %>%
  select(!c(MatchedWith, EventID, ID, ReservationPrice)) %>%
  select_if(is.numeric) %>%
  colnames()

varnames <- paste("\\textit{", varnames, "}", sep = "")

OnlySecSellersConstrained %>%
  select(!c(MatchedWith, EventID, ID)) %>%
  as.data.frame() %>%
  stargazer(style="aer",
            summary.logical = F,
            digit.separator = "$\\,$",
            digit.separate = 3,
            digits=2,
            float = F,
            omit.summary.stat = c("p25", "p75"),
            header=F,
            digits.extra = 2,
            font.size = "footnotesize",
            no.space = F,
            align = F,
            covariate.labels = varnames,
            median = T) %>%
  cat(file = "Paper/ArtWithsummarystats.tex")

OnlySecSellersUnconstrained %>%
  select(!c(MatchedWith, EventID, ID)) %>%
  as.data.frame() %>%
  stargazer(style="aer",
            summary.logical = F,
            digit.separator = "$\\,$",
            digit.separate = 3,
            digits=2,
            float = F,
            omit.summary.stat = c("p25", "p75"),
            header=F,
            digits.extra = 2,
            font.size = "footnotesize",
            no.space = F,
            align = F,
            covariate.labels = varnames,
            median = T) %>%
  cat(file = "Paper/ArtWithoutsummarystats.tex")
```

We create figures 5.1, 5.2 and 5.3, and save them as `ArtDensity.tex`, `ArtWithfigure.tex` and `ArtWithoutfigure.tex` into the `Paper` folder of our working directory (respectively).

```
tikz('Paper/ArtDensity.tex', width = 6, height = 3)
ArtDensity <- ggplot() +
  geom_density(data = OnlySecSellersConstrained,
               mapping = aes(x = PriceRatio,
                             col = "Constrained"),
               trim = T) +
  geom_density(data = OnlySecSellersUnconstrained,
               mapping = aes(x = PriceRatio,
                             col = "Unconstrained"),
               trim = T,
               n = 2^20,
               bw = bw.nrd0(OnlySecSellersConstrained$PriceRatio)) +
  theme(axis.title.y = element_blank(),
        legend.title = element_text(face = "plain")) +
  scale_x_continuous(breaks = c(0, 0.5, 1, 1.0815, 1.0815/0.95, 1.2978, 1.5),
                     minor_breaks = c(),
                     labels = function(x) round(x, digits = 2))+
  geom_vline(xintercept = c(1, 1.0815, 1.0815/0.95, 1.2978),
             linetype = "dotted") +
  coord_cartesian(xlim = c(0,1.5)) +
  labs(col = "Dataset")
print(ArtDensity)
dev.off()

tikz('Paper/ArtWithfigure.tex', width = 6, height = 4)
ArtWithfigure <- OnlySecSellersConstrained %>%
  ggplot(aes(x = -TimeLeftNow,
             y = PriceRatio,
             col = Matched)) +
  geom_point_rast(alpha = 0.7,
                  shape = ".",
                  dev = "ragg",
                  raster.dpi = 350) +
  guides(color = guide_legend(override.aes = list(shape = 16,
                                                   alpha = 1)))
print(ArtWithfigure)
dev.off()

tikz('Paper/ArtWithoutfigure.tex', width = 6, height = 4)
ArtWithoutfigure <- OnlySecSellersUnconstrained %>%
  ggplot(aes(x = -TimeLeftNow,
             y = PriceRatio,
             col = Matched)) +
  geom_point_rast(alpha = 0.7,
                  shape = ".",
                  dev = "ragg",
                  raster.dpi = 350) +
  guides(color = guide_legend(override.aes = list(shape = 16,
                                                   alpha = 1)))+
  coord_cartesian(ylim = c(0,1.5))
print(ArtWithoutfigure)
dev.off()
```

We set the dictionary of the `fixest` package (Bergé, 2018), in order to make the table display the variable names in the italic form (inside the `\textit{}` environment). Then we create the `ReplaceThings` function, which will perform some additional modifications on the `.tex` files containing our regression tables. Next, we use the `feols` function from the `fixest` package (Bergé, 2018) to run the fourteen regression models in tables 5.4 and 5.5[53]. Finally, we include these models in two LaTeX tables and save them as `FEmodelsArtWith.tex` and `FEmodelsArtWithout.tex` into the `Paper` folder of our working directory.

```
paste("\\textit{",
      colnames(OnlySecSellersConstrained),
      "}",
      sep = "") %>%
  setNames(colnames(OnlySecSellersConstrained)) %>%
  setFixest_dict()

ReplaceThings <- function(Latexoutput){
  Latexoutput %>%
    gsub(pattern = "\\\\times",
       replacement = "\\\\cdot") %>%
    gsub(pattern = "xxxx",
         replacement = "$\\\\times$") %>%
    gsub(pattern = "--",
         replacement = "$-$") %>%
    gsub(pattern = "& -",
         replacement = "& $-$") %>%
    gsub(pattern = ",",
         replacement = "$\\\\,$") %>%
    gsub(pattern = "$\\\\,$ ",
         replacement = ", ") %>%
    gsub(pattern = " '",
         replacement = " `") %>%
    gsub(pattern = "\\\\multicolumn\\{6\\}\\{c\\}\\{Heteroskedasticity-robust\\}",
         replacement = "HC & HC & HC & HC & HC & HC")%>%
    gsub(pattern = "\\\\multicolumn\\{8\\}\\{l\\}",
         replacement = "\\\\multicolumn\\{8\\}\\{c\\}")
```

[53]To run the Breush-Pagan tests (Breusch & Pagan, 1979), we need to use the `lm` function again.

```
}

model1with <- feols(
  PriceRatio ~ TimeLeftNow,
  data = OnlySecSellersConstrained,
  vcov = "iid")

lm(PriceRatio ~ TimeLeftNow,
   data = OnlySecSellersConstrained) %>%
  bptest()

model2with <- feols(
  PriceRatio ~ TimeLeftNow,
  data = OnlySecSellersConstrained,
  vcov = "HC1")

model3with <- feols(
  PriceRatio ~ TimeLeftNow+
    NumberOfAvailableSecondary +
    NumberOfMatchedSecondary +
    NumberOfAlerts,
  data = OnlySecSellersConstrained,
  se = "HC1")

model4with <- feols(
  PriceRatio ~ TimeLeftNow+
    NumberOfAvailableSecondary +
    NumberOfMatchedSecondary +
    NumberOfAlerts | EventID,
  data = OnlySecSellersConstrained,
  vcov = "HC1")

model5with <- feols(
  PriceRatio ~ TimeLeftNow+
    NumberOfAvailableSecondary +
    NumberOfAlerts | EventID,
  data = OnlySecSellersConstrained,
  vcov = "HC1")

model6with <- feols(
  PriceRatio ~ TimeLeftNow+
    NumberOfAvailableSecondary +
    NumberOfMatchedSecondary +
    NumberOfAlerts +
    i(SoldOut)| EventID,
  data = OnlySecSellersConstrained,
  vcov = "HC1")

model7with <- feols(
  PriceRatio ~ TimeLeftNow+
    NumberOfAvailableSecondary +
    NumberOfAlerts +
    i(SoldOut)| EventID,
  data = OnlySecSellersConstrained,
  vcov = "HC1")

model1without <- feols(
  PriceRatio ~ TimeLeftNow,
  data = OnlySecSellersUnconstrained,
  vcov = "iid")

lm(PriceRatio ~ TimeLeftNow,
   data = OnlySecSellersUnconstrained) %>%
  bptest()

model2without <- feols(
  PriceRatio ~ TimeLeftNow,
  data = OnlySecSellersUnconstrained,
  vcov = "HC1")

model3without <- feols(
  PriceRatio ~ TimeLeftNow+
    NumberOfAvailableSecondary +
    NumberOfMatchedSecondary +
    NumberOfAlerts,
  data = OnlySecSellersUnconstrained,
  se = "HC1")

model4without <- feols(
  PriceRatio ~ TimeLeftNow+
    NumberOfAvailableSecondary +
    NumberOfMatchedSecondary +
    NumberOfAlerts | EventID,
  data = OnlySecSellersUnconstrained,
  vcov = "HC1")

model5without <- feols(
  PriceRatio ~ TimeLeftNow+
    NumberOfAvailableSecondary +
    NumberOfAlerts | EventID,
  data = OnlySecSellersUnconstrained,
  vcov = "HC1")

model6without <- feols(
  PriceRatio ~ TimeLeftNow+
    NumberOfAvailableSecondary +
    NumberOfMatchedSecondary +
    NumberOfAlerts +
    i(SoldOut)| EventID,
  data = OnlySecSellersUnconstrained,
  vcov = "HC1")

model7without <- feols(
  PriceRatio ~ TimeLeftNow+
    NumberOfAvailableSecondary +
    NumberOfAlerts +
    i(SoldOut)| EventID,
  data = OnlySecSellersUnconstrained,
  vcov = "HC1")

etable(model1with, model2with, model3with, model4with, model5with, model6with,
       model7with,
       digits = "s1",
       digits.stats = "r4",
       tex=T,
       fitstat = c('n', "g", "r2", "ar2", "wr2", "f.p"),
```

```r
       se.row=T,
       file = "Paper/FEmodelsArtWith.tex",
       replace = T,
       fixef_sizes = T,
       fixef_sizes.simplify = F,
       style.tex = style.tex(main = "aer",
                             line.top = "double",
                             yesNo = c("Yes","No"),
                             fixef.where = "var",
                             fontsize = "footnotesize",
                             interaction.combine = "xxxx",
                             fixef_sizes.prefix = "Number of ",
                             fixef_sizes.suffix = "s",
                             tablefoot = T,
                             tablefoot.value = "default"),
       poly_dict = c("", "$^2$", "$^3$"),
       postprocess.tex = ReplaceThings,
       coef.just =  ".")

etable(model1without, model2without, model3without, model4without, model5without,
       model6without, model7without,
       digits = "s1",
       digits.stats = "r4",
       tex=T,
       fitstat = c('n', "g", "r2", "ar2", "wr2", "f.p"),
       se.row=T,
       file = "Paper/FEmodelsArtWithout.tex",
       replace = T,
       fixef_sizes = T,
       fixef_sizes.simplify = F,
       style.tex = style.tex(main = "aer",
                             line.top = "double",
                             yesNo = c("Yes","No"),
                             fixef.where = "var",
                             fontsize = "footnotesize",
                             interaction.combine = "xxxx",
                             fixef_sizes.prefix = "Number of ",
                             fixef_sizes.suffix = "s",
                             tablefoot = T,
                             tablefoot.value = "default"),
       poly_dict = c("", "$^2$", "$^3$"),
       postprocess.tex = ReplaceThings,
       coef.just =  ".")
```

In both datasets, we create two new variables (`Utility` and `Group`). The latter simply indicates whether the current agent is a buyer, a primary seller or a secondary seller. The `Utility` variable contains the utility that the agent realises from the transaction she takes part in (if any). For buyers, this is their reservation price minus the price they paid for the ticket. For secondary sellers, this is the price multiplied by the share that they get (about 0.8784 in the constrained and 1 in the unconstrained dataset). For primary sellers, this is simply the price itself. An agent who does not take part in any transaction of course realises zero utility. By summing up these utilities by `Group`, we obtain the surpluses in both daatasets. We store these in a tibble as `Surpluses`.

```r
Surpluses <- bind_rows(
  ArtificialDataWithConstraints %>%
    mutate(
      Utility = ifelse(Matched,
                       ifelse(Buyer,
                              ReservationPrice - Price,
                              ifelse(is.finite(alpha),
                                     Price * 0.95 / 1.05 / 1.03,
                                     Price)),
                       0),
      Group = ifelse(Buyer,
                     "Buyers",
                     ifelse(is.finite(alpha),
                            "Secondary sellers",
                            "Primary sellers"))
    ) %>%
    group_by(Group) %>%
    summarise(Surplus = sum(Utility),
              Case = "Constrained"),
  ArtificialDataWithoutConstraints %>%
    mutate(
      Utility = ifelse(Matched,
                       ifelse(Buyer,
                              ReservationPrice - Price,
                              ifelse(is.finite(alpha),
                                     Price,
                                     Price)),
                       0),
      Group = ifelse(Buyer,
                     "Buyers",
                     ifelse(is.finite(alpha),
                            "Secondary sellers",
                            "Primary sellers"))
    ) %>%
    group_by(Group) %>%
    summarise(Surplus = sum(Utility),
              Case = "Unconstrained")
)
```

Next, we calculate the optimal social surplus. On each simulated event's ticket market, we count the buyers (`Demand`) and sellers (`Supply`). The minimum of these two values is the maximum amount of possible transactions on the market, which would be the equilibrium quantity under perfect competition (`EquilibriumQuantity`). The optimal social surplus on this market (`SocialSurplus`) is the sum of the highest `EquilibriumQuantity` reservation prices. The aggregate surplus is then obtained by simply summing up these optimal values on each market. We store this value as `OptimalSocialSurplus`.

```
OptimalSocialSurplus <- ArtificialDataWithConstraints %>%
  group_by(EventID) %>%
  summarise(
    Demand = sum(Buyer),
    Supply = n() - sum(Buyer),
    EquilibriumQuantity = min(Demand, Supply),
    SocialSurplus = sum(
      sort(ReservationPrice, decreasing = T)[1:EquilibriumQuantity]
      )
  ) %>%
  select(OptimalSocialSurplus) %>%
  sum(na.rm = T)
```

We append the platform's profit in the constrained case to the `Surpluses` tibble. This value is calculated simply by multiplying the surplus of secondary sellers by $\frac{\tau}{1-\tau} \approx 0.1384$. We also append the optimal social surplus to the tibble. We then redefine the `Case` and `Group` variables in the `Surpluses` tibble as factors.

```
Surpluses <- Surpluses %>%
  add_row(Group = "Platform",
          Surplus = as.numeric(Surpluses[3,2])*(1.05*1.03/0.95-1),
          Case = "Constrained") %>%
  add_row(Group = "Society",
          Surplus = OptimalSocialSurplus,
          Case = "Pareto-optimal")%>%
  mutate(Case = factor(Case,
                       levels = c("Constrained",
                                  "Unconstrained",
                                  "Pareto-optimal"),
                       ordered = T),
         Group = factor(Group,
                        levels = c("Buyers",
                                   "Platform",
                                   "Secondary sellers",
                                   "Primary sellers",
                                   "Society"),
                        ordered = T))
```

We create figure 6.1 and save it as `SurplusPlot.tex` into the `Paper` folder of our working directory.

```
tikz('Paper/SurplusPlot.tex', width = 4, height = 3.5)
SurplusPlot <- Surpluses  %>%
  ggplot(aes(x=Case,
             y=Surplus,
             fill=Group,
             linetype = Group)) +
  geom_col(position = "stack", col = "black") +
  geom_col(position = "stack", col = "black",
           data = Surpluses %>%
             group_by(Case) %>%
             summarise(Surplus = sum(Surplus)) %>%
             mutate(Group = "Society")) +
  scale_fill_manual(
    values=c("#F8766D", "grey", "turquoise3", "turquoise4", "#00000000")
    )+
  scale_linetype_manual(values=c("blank", "blank", "blank", "blank", "solid"))+
  theme(axis.title.y = element_text(face = "plain"),
        axis.title.x = element_text(face = "plain"),
        legend.title = element_text(face = "plain"))
print(SurplusPlot)
dev.off()
```