Predicting Ride-Hailing Demand: A Potential Solution For Decreasing the Income Inequality of Drivers

Bachelor's thesis in Informatics

Christian Skorski

Student ID Nr. 18-700-294

Completed at the Department of Informatics
of the University of Zurich
Prof. Dr. A. Hannák

Supervisor: Stefania Ionescu

Submission date: 29.03.2022

ABSTRACT

Ride-hailing services such as Uber and Lyft have become globally pervasive in the last decade, revolutionizing the taxi sector for both customers and drivers. This business model breaks many barriers of entry for new drivers and makes commuting by taxi cheaper and more convenient for customers. Nevertheless, it is also affected by drastic income inequalities and weak job security. The goal of this thesis is to investigate whether we can reduce income inequalities by using ML-based demand prediction to strategically dispatch drivers. As such, I first develop a machine learning model to predict customer demand using a real-world ride-hailing dataset collected by the city of Chicago. Secondly, I integrate the real-world data within an agent-based model to make an initial exploration of the potential of using the predictions for decreasing the income inequality of drivers. The results show that (a) the prediction model is able to fairly accurately predict demand, and (b) one of the three implemented rule-based naive dispatchers successfully used the predictions in order to increase the level of fairness.

ABSTRAKT

Ride-Hailing-Dienste wie Uber und Lyft haben sich in den letzten zehn Jahren weltweit durchgesetzt und das Taxigewerbe sowohl für Kunden als auch für Fahrer revolutioniert. Dieses Geschäftsmodell beseitigt viele Einstiegshürden für neue Fahrer und macht das Pendeln mit dem Taxi billiger und bequemer für die Kunden. Allerdings ist es auch von drastischen Einkommensunterschieden und geringer Arbeitsplatzsicherheit betroffen. Ziel dieser Arbeit ist es, zu untersuchen, ob wir die Einkommensungleichheit verringern können, indem wir ML-basierte Nachfragevorhersagen nutzen, um die Fahrer strategisch einzusetzen. Zu diesem Zweck entwickle ich zunächst ein Machine Learning Model zur Vorhersage der Kundennachfrage anhand eines realen Ride-Hailing-Datensatzes, der von der Stadt Chicago gesammelt wurde. Zweitens integriere ich die realen Daten in ein agentenbasiertes Modell, um das Potenzial der Vorhersagen zur Verringerung der Einkommensungleichheit von Fahrern zu untersuchen. Die Ergebnisse zeigen, dass (a) das Prognosemodell in der Lage ist, die Nachfrage ziemlich genau vorherzusagen, und (b) einer der drei implementierten regelbasierten naiven Dispatcher hat es erfolgreich geschafft, die Prognosen zu nutzen um eine Fairnessverbesserung zu erreichen.

ACKNOWLEDGEMENTS

Firstly, I would like to thank my professor, Dr. Prof. Anikó Hannák, for the opportunity to work on this project and build upon her research. Secondly I would like to thank my supervisor Stefania Ionescu for her amazing support. It is only thanks to our weekly, hour-long meetings and her invaluable inputs that I was able to tackle such an ambitious project. Finally, I would also like to thank Dr. Nicolò Pagan for his valuable insights regarding the inner workings of the agent-based taxi simulator I worked on.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Ride-hailing services such as Uber and Lyft have become pervasive in the last decade, revolutionizing the taxi sector for both customers and drivers. To give a perspective on their relevance, only in Chicago there were more than 67'000 active Uber and Lyft drivers before the Covid-19 pandemic, compared to only 6'999 licensed cabs (Channick). Ride-hailing companies define themselves as tech companies, providing a smartphone application to match people needing transportation with independent individuals willing to provide it. This business model makes on-demand transportation very convenient and cheap for customers, while also breaking many barriers of entry for new drivers and offering them complete flexibility over working hours.

Yet, ride-hailing services are also affected by important issues causing weak job security and scarce economic reliability. This is in part due to the employment status of the drivers which, according to ride-hailing companies, entails employing drivers as independent contractors instead of as formal employees (Scheiber [2021]). This means that ride-hailing drivers are not by default protected by the same laws that otherwise protect normal employees such as taxi drivers, barring exceptions where drivers successfully pursued legal action to be recognized as employees in their state (Conger and Scheiber [2019]). Especially problematic are the vast differences that can arise in the incomes of drivers that work for similar amounts of time, which are mainly due to algorithmic decisions of the ride-hailing companies' matching algorithms as analyzed by Bokányi and Hannák [2020]. Therefore, there exists a strong need to protect the interests of drivers which, apart from legislation, can be tackled from the perspective of algorithmic fairness.

Some previous literature tried to solve the latter issue for example by optimizing the driver-customer matching problem for the long term income equality (Sühr et al. [2019]), or by developing a matching algorithm prioritizing poor drivers in a certain radius instead of assigning customer to the nearest driver (Bokányi and Hannák [2020]). Nevertheless, the vast majority of literature, especially works studying machine learning approaches, focuses on optimizing the matching problem for the total utility instead (Li et al. [2019], Syed et al. [2019], Stein et al. [2020]). To my knowledge, there has been no attempt in the literature to optimize ride matching for income fairness.

1.2 Objectives

In my thesis, I aim to close the aforementioned gap in the literature by investigating whether machine learning-supported demand prediction can be used to create a fairness-focused driver dispatcher algorithm. The end goal is for the dispatcher to be able to increase income equality while maintaining the best possible total utility, in an attempt to make working for ride-hailing companies economically stabler and safer.

To this goal, after a data cleaning and preparation phase, many different regression models were trained and tested on the prepared training set. In the end, the system was implemented using a multi-output supervised machine learning algorithm, Random Forest regression, to predict real-time customer demand in each city division in the form of concrete counts of probable customers. The divisions of the city do not follow geopolitical boundaries, They are instead defined by an imaginary grid of squares, which is defined either by (approximated) square length in degrees or by the number of desired divisions on the latitude axis. This solution offers flexibility over the trade-off between the geographical precision of the predictions and the required regression accuracy. In contrast, Carson-Bell et al. [2021] compared different prediction models for customer demand on the same dataset, but their prediction was limited to indicate the most likely Chicago community area in which the next customer request will originate from.

In approaching this goal, my main contributions are:

- Developing a multi-output ride-hailing demand prediction model with customizable output size (prediction granularity), with concrete predicted counts of probable customers per city area as output;
- Integrating said model in an existing agent-based city model (or taxi simulator) developed by Bokányi and Hannák [2020], including adapting the simulator to use real-world data to generate demand;
- Developing three naive rule-based driver dispatchers making use of the pre-

diction model into the simulator, one of which achieved a promising fairness improvement of 8.35% (in Gini coefficient) over current fairness-focused algorithms.

Chapter 2

Literature Review

In the first of the next two sections, I give an overview of existing literature studying income fairness of ride-hailing drivers. In the section that follows, I look at some of the state of the art solutions for the ride-hailing matching optimization problem.

2.1 Ride-hailing income inequality

A lot of research in the last few years focused on studying income inequality of drivers and understanding its causes. For example, Zoepf et al. [2018] interviewed over 1'100 Uber and Lyft drivers and discovered that (a) the median profit from driving is \$3.37/hour before taxes, and (b) 74% of drivers earnings fall under the minimum wage in their state. Perhaps even more shockingly, 30% of drivers actually lose money by working, after including vehicle expenses.

As an example of the impact of algorithmic design in the context of ride-hailing, Chen et al. [2015] emulated 43 Uber drivers in downtown San Francisco and midtown Manhattan to understand the true impact of the *surge pricing* (noa [a]) algorithm on passengers and drivers, working around the fact that Uber does not provide any data about supply or demand. In my thesis I decided not to consider surge pricing, as its timescale is relatively small compared to a whole 42 hour work week (noa [b]).

While the before-mentioned literature focuses on investigating whether or not there are inequalities in the system, isolating the effects of each component of the system on the producing income inequality is a separate question. The difficulty of this question comes from the need of very detailed information and the means to investigate counterfactual scenarios. To overcome this difficulty, Bokányi and Hannák [2020] used agent-based modelling to analyze the effects of algorithm design decisions on wage inequality. For this purpose, they created a taxi and ride-hailing simulator in a simplified city model, which helped them conclude that even small changes to the system parameters can lead to large short term differences in the drivers wage distribution, which can then potentially escalate on the long term through algorithmic feedback loops. The goal of this study was to demonstrate (1) the large consequences that small changes in algorithmic design can cause in the context of ride-hailing, and (2) the urgency of better information transparency towards drivers of ride-hailing platforms, to enable them to make informed decisions while working. My work is highly akin to this study: the simulations performed by Bokányi and Hannák [2020] use synthetically generated data. I improve upon this technique by integrating real-world data in the simulation and validating the results of the study. I then use the simulator to test my demand prediction model and my three naive dispatchers in combination with the ride matching algorithms developed by the authors of the study.

Sühr et al. [2019] also observed significant income inequality by analyzing job assignments of a major taxi company. They proposed that income fairness should be measured over a longer time period, arguing that by removing the requirement that every matching should be as fair as possible individually, better overall benefit can be achieved for both drivers and passengers. To achieve this *two-sided fairness*, they experimented with various optimization problems. By balancing equality of both drivers and customers they were able to increase equality while maintaining the total utility volumes. They also showed that although letting worst-off drivers choose their customers first could increase equality, it could also cause a worst average and median total income. This work is also important to my thesis: I incorporated the authors' idea of measuring fairness over time by simulating driver-customer matches over a week at a time, while at the same time diverging from their solution to increase fairness by branching out into machine learning.

2.2 Ride matching optimization

There has been a lot more academic effort towards optimizing ride-hailing matching efficiency, and thus total company profits. Due to the high complexity and the dynamic nature of the ride-hailing matching problem, state of the art solutions often implement reinforcement learning algorithms. For example, Li et al. [2019] argued that classic rule-based solutions are designed on a simplified vision of the problem, and require sophisticated fine tuning to work. To effectively model the peer-to-peer interactions between multiple drivers and customers, they proposed to use multi-agent reinforcement learning (MARL) combined with mean field approximation to simplify local interactions. Apart from closely matching the problem, an advantage of such a system is the ability to be deployed as a fully distributed system, with the associated resilience benefits. The authors were able to prove that their system performs substantially better than three simple rule based systems, measuring a higher total daily income and order response rate. It also outperformed the total daily income of a combinatorial optimization method based on the Hungarian algorithm by Mills-Tettey et al. [2007].

On the other hand, Syed et al. [2019] approached the problem differently: They noted that it similar to the classic Dial a Ride Problem, which can be efficiently solved using the Adaptive Large Neighborhood Search Algorithm (ALNS), with the difference of the additional dynamic aspect (new requests are continuously incoming). Therefore, they evaluated the performance of a rolling horizon ALNS in an asynchronous real-time framework, i.e. with three separate processes to compute time and vehicles movements, requests and ride management, and route computation. Similarly to my work, they used real-world New York taxi data for testing. They were able to conclude that ALNS can be used for real-time applications and gives better solutions than the Nearest Neighbor algorithm normally used by ride-hailing platforms. They also remarked that the system could be further improved by combining it with customer demand prediction.

Although not specifically with ride-hailing in mind, Stein et al. [2020] analyzed the generic online (dynamic) resource allocation problem from the perspective of resilience to individuals' strategies. In fact, they argued that many works implementing reinforcement learning solutions neglect this aspect of human nature. Therefore, they developed a novel reinforcement learning-based mechanism in which truthful reporting and participation are incentivized. Their algorithm was able to stay within 90% of the optimal social welfare obtained by backwards induction and dynamic programming, and outperformed all benchmark algorithms such as first-come-first-served.

While all these papers focus on optimizing ride matching for the total utility (welfare), they do not analyze the effects on fairness. Moreover, reinforcement learning is very resource intensive and not yet extensively used in the industry. This thesis diverges from these studies by developing a supervised prediction model and pairing it with a dispatcher to optimize ride-matching for fairness, while maintaining the same or the highest possible total utility.

Chapter 3

Methodology

The next sections present the methods used for this thesis, including the work performed on the original data, the selection and training of the machine learning model, the integration of the data and the model into an agent-based taxi simulator, and the parameter configurations used for the simulations.

3.1 Data sourcing, cleaning and preparation

The data used for this work comes from the data portal of the city of Chicago, which makes all kinds of statistical records of the city available to the public. The data used in this thesis comes from a 60GB dataset with 240 millions anonymized trips reported by transportation network providers (ride-hailing companies). The data collection has started in November 2018, continuing at the time of writing this thesis (March 2022). Out of the 21 columns of the dataset, I only use the timestamp and the pick-up and drop-off centroid location coordinates in degrees.

The dataset had missing values for the coordinates fields, so it had to be cleaned. There were entries with a missing coordinate field, amounting for about 5% of the dataset. Because this was a low percentage of such entries, I decided to drop them. According to the dataset description, the missing coordinates are due to the pickup or drop-off location being outside of Chicago. The data also had three invalid values in the coordinates, which I removed with a regex. Finally, I separated ca. 95% of the data for training and testing the prediction model, leaving 5% for the simulations.

Since the number and nature of request is dependent on the type of weather as found by Liu et al. [2021], I added columns to the ride-hailing dataset to provide this contextual data. More precisely, I sourced historical weather data from the National Centers for Environmental Information website (NCEI). I also tried adding a column stating if a day is a holiday even though I could not find conclusive studies supporting this idea, sourcing federal/state holidays data from officeholidays.com.

3.2 Machine learning prediction model

For the purpose of this work, I view this problem as a supervised machine learning problem, batch (offline) and model-based. On one hand, matching customers with drivers is a typical online resource allocation problem, and state-of-the-art solutions often approach it with reinforcement learning as mentioned in the literature review section. Yet, such methods require a lot of computational power and are not yet as commonly used in the industry as supervised learning (z_ai [2021]). Therefore, the scope of this thesis is to investigate if it is at all possible to decrease the income inequality using a relatively simpler, more business-proven supervised learning solution.

As I did not have the possibility of learning about supervised learning during my Bachelor's studies, I prepared myself for this part of the thesis with the great introductory course by Andrew Ng (Ng) from the University of Stanford. I have also used the machine learning book by Géron [2019] as reference during the development of the model and, as in the book, I implemented the model using Python and the Scikit-learn library by Pedregosa et al. [2011]. The resulting code is publicly available on GitHub [1].

3.2.1 Model features and labels

The goal of the ML-model is to predict demand in various zones in the city based on data readily accessible by the drivers app or by the servers of the ride-hailing service platform. To predict the total amount of customer (riders) requests for each defined city division at any given time step, I chose to use features based on date, time, weather and holidays.

I decided to use a multi-output regression model, because the dispatcher needs to know demand in every division of the city for each decision. Instead of predicting demand for each area of the city separately in a loop, multi-output regression returns an array with the predicted customer counts for each area of the city at once. Both methods are viable, but the latter seemed a better fit for this problem to me, and might intuitively cause less overhead to the simulation or real-world application than a big prediction loop.

The features set defines the inputs of the prediction model. As the main inputs of my model are date and time, I chose to separate the time stamp into day, month, week of the year, day of the week, hours, and minutes. The reasoning behind this

---

[1] GitHub link for the repository: https://github.com/ianskoo/ride-hailing-income-fairness.git

is that supervised machine learning algorithms look for correlations and patterns between the features (training input) and the labels (training output), but they are often not able to infer correlations from combinations of inputs without manually combining the features beforehand (e.g. by multiplying them together), or extract implicit information from features such as the day of the month from dates. Moreover, all features must be transformed and scaled to numerical values between 0 and 1 to be understood and correctly interpreted by the algorithm. Therefore, extracting for example the day of the month from a date string enables the algorithm to find monthly patterns, whereas if only the date was kept and transformed to a numerical value between 0 and 1, that useful implied information would have probably been lost.

Other than the date and time features, I used the weather and holiday data: these are pieces of information that are easily available to the drivers (or the application/platform), and can provide useful correlations; For instance, more people getting an Uber when it is raining, or a shift/reduction in high demand areas during holidays. The three weather features are the daily high and low forecasted temperatures in Fahrenheit, and inches of rain (all constant for a given day). Having weather information for every single hour would create more useful variance, but I was not able to find a dataset with such detailed information. For holidays I added a single binary feature that equals 1 if the current day is a holiday, and 0 otherwise. Surprisingly, the improvement achieved with the latter was marginal (+1% accuracy at best), probably due to the relative scarcity of holidays with respect to normal work days throughout the year.

To have defined areas to dispatch drivers to, decrease the size of the output space (labels) and ease prediction difficulty, I grouped locations by *city divisions*

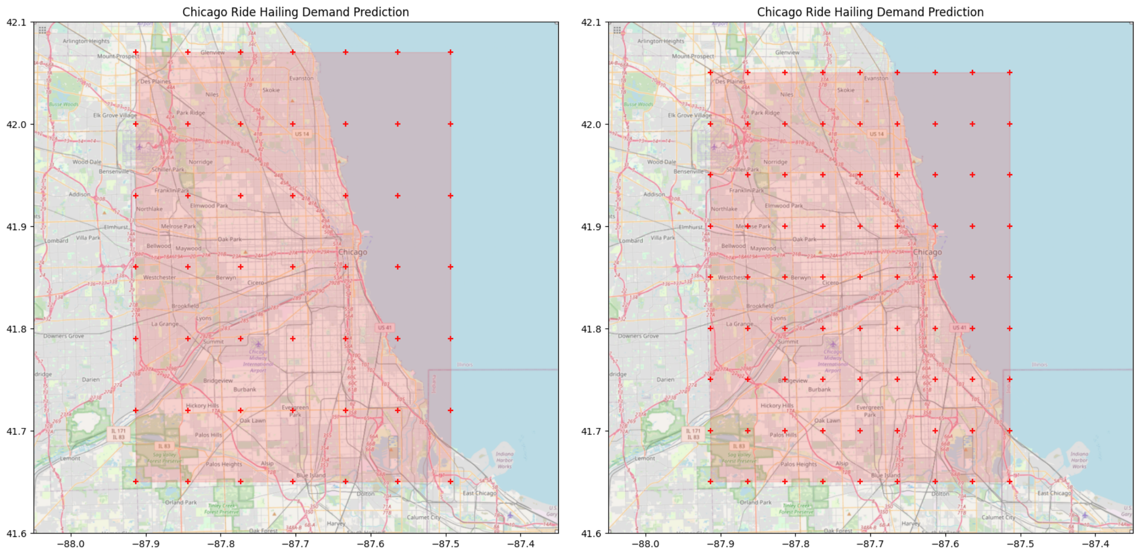
Chicago Ride Hailing Demand Prediction
Chicago Ride Hailing Demand Prediction
42.1
42.0
41.9
41.8
41.7
41.6
−88.0
−87.9
−87.8
−87.7
−87.6
−87.5
−87.4
Chicago

Figure 3.1: An example of the city divisions on a map of Chicago for different division sizes α and number of divisions. Left: $\alpha = 0.07°$, 36 divisions; Right: $\alpha = 0.05°$, 64 divisions

(i.e., rectangular areas in the city). Therefore, the resulting grouped dataset had for each timestep t and division d the number of requests at timestep t in division d. Each label (column) is named by the bottom-right corner coordinates of the city division square it represents.

The city divisions are rectangles determined by a parameter α, or alternatively a number of divisions of the city latitude range `lat_divisions`. The parameter α is an angle (in degrees) of either longitude or latitude of the Earth, which determines both the rectangle side lengths. At the latitude of Chicago (≈ 42), a change in latitude (in degrees) covers much more distance (in Km) than the same change in longitude, causing divisions defined by the same α to become rectangles. α can be chosen beforehand (e.g. 0.01°) or alternatively computed by the program by passing a divisor for the latitude range, i.e. the number of rows for the city grid. The smaller the size of α, the higher the number of city divisions, negatively impacting prediction accuracy but positively impacting the overall usefulness of the model and of the final dispatcher. Figure 3.1 shows two examples of how these divisions would look on a map.

3.2.2 Model selection

As usual in ML-training, the goal will be to maximize the accuracy of the model. However, please note that in this particular application, the model might still be useful with lower levels of accuracy. For example, a model that always predicts at least half of the actual number of requests is still useful for dispatching the poorest drivers to get half of the total number of requests, and leave the other drivers to satisfy the remaining half. I will expand on this later in the paper.

The model is selected by k-fold cross-validation (using the `cross_val_score()` function of Scikit learn). Cross-validation is a technique used to compare the performance of different machine learning algorithms on a given dataset, as well as for tuning the hyper-parameters of a model after choosing it. It works by dividing the training set into k folds, in my case five, training $k-1$ models of the same kind and parameters on the folds and using the last fold to test the predictions of the models. This technique avoids that the models under scrutiny overfit the dataset, i.e. exhibiting great training accuracy but sub-par generalization accuracy, meaning bad prediction accuracy when presented with new input data.

A problem with this approach is that the models probably ca not find yearly patterns, because the training dataset consists of less than two years of data which is again divided in 5 subsets for the cross-validation. The yearly patterns in question could e.g. include holidays, which most probably have an impact on the amount and location of pickup requests in the city. This is the reason why holidays were manually added in the training dataset.

model	polyn. degree	training time	NMSE	std NMSE
RandomForestRegressor(max_depth=10)	1	9.74	-17.21	0.78
RandomForestRegressor(max_depth=10)	3	256.24	-17.84	0.40
ExtraTreesRegressor(max_depth=10)	2	24.94	-18.51	1.04
RandomForestRegressor(max_depth=10)	2	56.06	-18.55	1.55
ExtraTreesRegressor(max_depth=10)	3	82.30	-18.61	0.98
ExtraTreesRegressor(max_depth=10)	1	7.36	-24.71	1.18
LinearRegression()	5	78.93	-84.72	3.48
Ridge()	6	131.26	-84.72	3.48
Ridge()	5	13.27	-93.18	3.35
LinearRegression()	4	5.77	-107.98	4.40
Ridge()	4	3.05	-107.98	4.40
Ridge()	3	0.63	-137.44	6.75
Ridge()	2	0.47	-182.85	9.52
SVR(kernel='poly')	1	295.02	-187.31	9.14
SVR(kernel='poly')	2	342.88	-190.25	9.42
ElasticNet(alpha=0.1)	4	535.82	-190.96	8.94
ElasticNet(alpha=0.1)	3	103.88	-200.88	9.72
ElasticNet(alpha=0.1)	2	8.23	-222.50	11.67
Ridge()	1	0.20	-244.72	12.10
ElasticNet(alpha=0.1)	1	1.86	-273.33	16.43
ElasticNet(alpha=0.1, max_iter=10000)	1	8.46	-273.33	16.43

Table 3.1: Performance comparison of different machine learning models by cross-validation on the training set, sorted by negative mean square error (NMSE). Training time is in seconds, polynomial degree refers to new features created by powers (of that degree) and combinations of the existing features.

I compared multiple regression models for the task. Some did not support multiple output or were not working well with it, which was a deal-breaker due to the requirement to predict demand in separate zones of the city at once mentioned before. The first shortlist of regression models therefore included: linear, Ridge (regularized regression), decision tree, Random Forest, Extremely Randomized Trees, Support Vector Machine, Ada Boost, and Elastic Net.

Table 3.1 shows a ranking of the shortlisted models in increasing order of negative mean square error (NMSE). Ensemble methods such as random tree proved to be among the best performing for this problem as also noted by Carson-Bell et al. [2021], even though I am using a processed version of the Chicago data. Random Forest regression is the clear winner, not even needing the features to be combined together with a polynomial degree like linear or Ridge regression do, and taking an impressively small amount of time to train compared to the latter two. Other options like support vector machine regression or Elastic Net have not performed as well for this particular problem. Extremely Randomized Trees, which are supposed to have lesser variance at the cost of more bias compared to Random Forest, surprisingly performed slightly worse than the latter. Based on these results, I decided to keep Random Forest regression and Ridge regression for the next phase of the selection, as the latter could still give interesting results with different parametrizations.

3.2.3 Hyperparameter tuning

The next step of the prediction model implementation is fine tuning the hyperparameters. Hyperparameters are the parameters of a machine learning model that control its learning process, such as the regularization factor of a regularized linear regression model. As with the model selection, k-fold cross validation is very useful in this part as well to avoid overfitting the training set and have poor generalization accuracy.

To find the best hyperparameters for each of my two final model candidates automatically over a range of combinations, I use a grid search algorithm implemented in `sklearn.model_selection.GridSearchCV()`. As the name implies, this function already includes cross validation built-in by default.

The choice of values to test out for the hyperparameters is usually a range of a few multiples of ≈ 3 under and over a baseline value. For example, if the standard hyperparameter used for a specific model is equal to 100, a good range of candidates could be 10, 30, 100, 300, and 1000. Of course, the bigger the ranges of each hyperparameter to try out, the longest it takes to check every combination of them. To solve this problem, there is another search algorithm based on randomly choosing combinations of hyperparameters, but I decided the simpler grid search method was enough for the scope of my thesis.

Random Forest regression

For Random Forest, the Scikit Learn documentation (noa [c]) recommends adjusting the number of trees (`n_estimators`) and the size of the random subset of features used to split a node of a tree (`max_features`).

Interestingly, it seems that also limiting the depth of the trees to a certain depth with `max_depth` can positively impact the cross-validated performance of the model. This could be intuitively attributed to the expected lower bias of such a limit.

The parameter with the strongest effect on accuracy was the number of estimators (trees), which capped its "return on investment" at around 300 trees, with 1000 trees taking much longer to train and giving close to no accuracy improvement.

The final hyperparameters chosen for Random Forest are therefore a max tree depth of 100, a max pool of features for splitting nodes of 8, and 300 estimators,

```
44.247130183005766 {’max_depth’: 30, ’max_features’: 7, ’n_estimators’: 300}
44.222346577150965 {’max_depth’: 30, ’max_features’: 8, ’n_estimators’: 300}
44.23452364395394 {’max_depth’: 30, ’max_features’: 9, ’n_estimators’: 300}
44.7236298360614 {’max_depth’: 30, ’max_features’: 10, ’n_estimators’: 300}
45.58037374301642 {’max_depth’: 30, ’max_features’: 11, ’n_estimators’: 300}
44.13458527211603 {’max_depth’: 100, ’max_features’: 7, ’n_estimators’: 300}
44.049430471330055 {’max_depth’: 100, ’max_features’: 8, ’n_estimators’: 300}
44.27055976644326 {’max_depth’: 100, ’max_features’: 9, ’n_estimators’: 300}
44.64257553673474 {’max_depth’: 100, ’max_features’: 10, ’n_estimators’: 300}
45.47461065777984 {’max_depth’: 100, ’max_features’: 11, ’n_estimators’: 300}
Best parameters: {’max_depth’: 100, ’max_features’: 8, ’n_estimators’: 300}
```

Figure 3.2: Sample output of the grid search function coupled with some custom print statements. The first value of each row is the mean squared error (MSE), which serves only as a comparison measure at this stage.

giving a mean error (not squared) of ≈ 3.80.

Ridge regression

The hyperparameters of the Ridge regression algorithm selected for tuning are the polynomial degree of features, and the regularization strength alpha (which is not to be confused with the city division size α). As mentioned before, the polynomial degree of features is not technically a parameter of the model itself, but a transformation of the training set features. It allows linear regression to approximate a polynomial function for fitting a non-linear prediction space, and it combines features together by multiplication to allow the model to find relationships between them.

Ridge regression additionally uses regularization, which is a way to constrain linear regression such that it does not overfit the data and allowing it to use higher polynomial degree features with minor bias increases. The higher alpha, the higher the regularization strength, flattening out the approximated function.

The learning rate of Ridge regression seemed to cap out at a mean error of around 8.0, which is more than double the best mean error of Random Forest. This was to be expected due to the higher complexity of an ensemble method, which combines different machine learning models to cancel out different kinds of errors each of them might make to reach a higher overall accuracy. Ridge regression is just one model, so it is impressive that it came this close to an ensemble method anyways.

3.2.4 Generalization error

Taking the winning model (Random Forest regression) to the next phase, it was now time to train it on the whole training set and measure the generalization error on the test set. Table 3.2 presents the generalization scores for the Random Forest regressor with the optimal hyperparameters found in the earlier step. Each row shows the performance of the model trained on a different dataset having the same features set but a different city division size, thus a differently sized labels set.

city division size	divisions count	MSE	mean error	accuracy (R^2)
0.394°	1	37018.19	192.40	0.990
0.09°	25	565.64	23.78	0.855
0.07°	36	354.04	18.82	0.777
0.06°	49	261.69	16.18	0.855
0.05°	64	190.51	13.80	0.798

Table 3.2: Generalization error and R^2 scores of Random Forest regression, trained on datasets with the same features set but differing multi-output granularity.

These results are better than those obtained through cross-validation, which is to be expected due to the model training on the whole training set as opposed to k = 5 separate folds (subsets). It is interesting that the model predicting demand for 49 divisions has counter-intuitively a much higher R^2 score (≈ 0.855) than the one predicting for 36 (≈ 0.777), which provides less prediction granularity. This phenomenon may be explained by a more fortunate disposition of the smaller divisions of the former model, better covering areas of high demand such as the Chicago downtown. Intuitively, this would increase variance between the features and the labels associated to those squares, allowing the model to better explain the data and have a lower error.

On the other hand looking at the next row, a prediction granularity of 64 divisions gives a score of 0.798, slightly lower than with 49 divisions. This may suggest the inverse of the earlier reasoning, that 36 divisions is just a highly sub-optimal grid for separating different demand zones in Chicago. Figure 3.1 could support this idea, as the grid with 36 divisions has a division encompassing a chunk of the highly populated downtown while being mostly on water, whereas the second grid has a square nicely covering the entire downtown.

The high R^2 scores may have alternative explanations linked to the limitations of my work. For example, the scores of more granular prediction models may be artificially boosted by a higher number of divisions completely overlapping the lake, which always have a count of 0 requests and are trivial for the model to predict. This may hold for very low demand areas as well, such as the outskirts or low income districts. I regrettably did not have much time to further validate these results. In addition, as mentioned before, I expected that even if the accuracy was actually lower, the models could still be useful to dispatch drivers to high demand areas. If a low-accuracy model gives some promising results, then a refined version with higher levels of accuracy could provide an additional increase in fairness. Therefore, I proceeded with the straightforward delimitation of rectangular city divisions, and left the development of models that use the city-level expertise for future work.

3.3 Integrating the model into a taxi simulator

In order to use the model to dispatch drivers, I chose the agent-based taxi simulator developed by Bokányi and Hannák [2020]. A simulation environment makes it possible to easily and quickly test different dispatching strategies, of which potential positive results could perhaps warrant a more expensive and time consuming real life experiment. The chosen simulator needed a number of adaptations to use real-life data from the Chicago dataset and the prediction model with it, which I will explain in the next sections. Most of the adaptations could be made without touching the original code in a separate class, the *Chicago class*, which I will sometimes mention in the explanations.

The simulator is an agent-based model of a simple rectangular city represented by a mxn grid. Customer requests are randomly generated and positioned in the city based on various included distribution functions. Drivers are initially either positioned on a predefined base, or randomly throughout the city. There are four driver-customer matching algorithms included: Random unlimited, random limited, nearest first, and poorest first. If a queued request can be assigned to a driver, the driver drives to the customer, picks them up and drives them to their destination. Afterwards, the driver can either stay there and wait for another match, or move to a predefined location. The simulator parameters and the matching algorithms are explained more in detail in the next sections.

3.3.1 Adapting the simulator to use Chicago data

The fundamental question to understand how to make a simulator use real data is: Which previously modifiable parameters of the simulation will now depend on the data, and how can they be adapted in the simulator to reflect that? The main dependent parameters are the customer demand (request rate in the simulator) and the position of the customer requests. On the other hand, the supply is still generated by the simulator and can be adjusted to see the effects of different driver densities and demand/supply ratios.

The first step was to map Chicago to the simulator city grid, which is a simple grid of $m \times n$ squares with internal length equal to 100m. In order to avoid cascading problems, I chose not to interfere with all the internal spatial and temporal parameters. Hence, in my simulation they remain fixed at the values chosen by Bokányi and Hannák [2020] based on real-life data. I therefore computed the distance in meters of the highest and lowest latitudes and longitudes in the simulation set, either of a trip pick-up or a drop-off, and divided these distances by the simulator distance unit $du = 100$m to get the grid size. The city size, depending on the chosen subset of the simulation dataset is around 32.0×41.2 Km, translating into a grid size of 320×413 blocks. Simulating only the space containing requests instead of the whole city using official measures helps avoiding potential index errors if some requests were outside of the official city limits, and lighten resource usage otherwise.

Then, I created a mapping for the time. In the simulator, a time unit tu is equal to 10 seconds. Since the time steps in the dataset are 15 minutes, we have: $15' = 900'' = 90 \cdot tu$. The conversion is implemented in a simple method of the Chicago class that takes the simulation time, computes the floored division by 90 and uses that result to access a time step from a list of all the distinct time steps in the current simulation Chicago subset.

Next I substituted the simulator request generation, originally based on random distributions, with a function taking requests from the Chicago dataset preassigned to each time unit *tu*. This was difficult because the simulator used a `request_rate` parameter to determine how many requests to generate per time unit *tu* by calling a single request generating function `request_rate` times. This was incompatible with the real-world data where the request rate should vary in time following the demand fluctuations in the Chicago dataset.

The simplest and least invasive solution was for the Chicago class to randomly assign all the requests of a Chicago time step (15') to the individual $90tu$ that cover that time step. This also avoids unnaturally inserting the requests in the city all at once at the beginning of the 15' time step. Since the simulator can only add one request at a time, method computes how many requests are assigned to each single *tu*, and feeds an approriate dynamic request rate to the simulation. To achieve all this, I added a column to the simulation dataset and populated it with random integers from 1 to 90 (0-89 in the code). Then, I slightly modified the simulator to use a method in the Chicago class to determine how many requests to pick from the dataset at a given simulation time unit *tu*. For example, if the simulation is at $tu' = 183$, we would get a Chicago time step $TS' =$ `unique_timesteps[floor(183 / 90)]`, which could e.g. be equal to "2020-1-6 10:15:00" if the simulation data starts on that day at 10:00:00. Then, all requests in the dataset with time step TS' and labeled with $index_{tu'} = 183 \bmod 90 = 3$ will have to be placed on the map. The simulation would thus receive a request rate equal to how many requests in the data are labeled with the modulo of 90 of the current simulation time *tu*, and ask the Chicago class for that many requests.

The following issue was determining the right amount of supply (drivers). In the original simulator, supply and demand were fixed for the entire simulation, but now we have a variable supply given by the Chicago class. For the sake of simplicity and to make comparisons between my results and those found by Bokányi and Hannák [2020] still possible, I decided to keep the number of drivers (supply) fixed and run the simulation during daytime only, for a total time equivalent to a 42-hour work week. To enable this, a new subset is taken out of the simulation dataset by choosing a start and an end date, and the desired working hours during the days between those dates. The demand to supply ratio is thus now given by the averaged demand of the simulation subset, divided by the fixed supply.

3.3.2 Fixing the initial placement of drivers

A limitation of the simulator is that it does not offer the possibility of placing the initial drivers using a normal distribution, similarly to how the customer requests are generated and placed in the city (to my knowledge). In reality, cities have differently populated areas with different average income classes. Intuitively, more densely populated areas with an average to low mean income would probably house a higher density of people driving for a ride-hailing company. This assumption is reinforced by the lower than average wages characterizing this job mentioned by Zoepf et al. [2018].

The reason why this is important is that if customer requests follow a normal distribution with one or more centers and the drivers are uniformly distributed over the whole city, there will be drivers that most probably will not get assigned to any request during the whole simulation. In fact, running a simulation using 5000 randomly and uniformly placed taxi drivers gives the following results:

- Number and percentage of missed customer requests: (19442, 19.78%)
- Number and percentage of completely idle drivers: (2942, 58.84%)

Whereas by increasing the drivers to 10000:

- Number and percentage of missed customer requests: (301, 0.31%)
- Number and percentage of completely idle drivers: (5626, 56.26%)

The percentage of unanswered requests went down to nearly zero, but the amount of taxi drivers without any completed trip is still more than half of all drivers, which is clearly really detrimental to income equality. My solution to this problem is: instead of placing the drivers randomly on the map at the start of the simulation, I can add a function to my Chicago class that samples a single random customer request from the data, returning its simulation grid coordinates. This way, the simulation can use this helper function to generate home coordinates for drivers following the same distribution of the customers. While alternative distributions could also make sense, I use this as a more realistic alternative to the uniform random placement. Most importantly, this new placement does not position drivers in uninhabited areas (e.g., on water). Below, I include the results with the new driver placement function:

- Number and percentage of missed customer requests: (713, 0.73%)
- Number and percentage of completely idle drivers: (26, 0.26%)

3.3.3 Using the ML-model to dispatch drivers

Besides integrating the Chicago data in the simulator, I also developed and implemented three basic dispatchers. A dispatcher is an algorithm which aims to use the demand prediction model to strategically direct less well-off drivers to high-demand areas of the city. It is important to note that positioning drivers efficiently is a non-trivial optimization problem on its own. The best solution depends on a variety of factors such as the current position of the drivers, the trust and accuracy of predictions, the driving time between the current position of the diver and possible requests, and the expected destinations of requests. In addition, depending on the size of the city, solving the full optimization problem might be computationally infeasible, so one could instead use approximations and simplified versions of the optimization problem. Moreover, the problem formulation might need to account for competing objectives, such as: (a) income equality, (b) revenue, (c) time and fuel efficiency. However, due to lack of time, I did not explore all the complexities of this allocation problem. Instead, I only implemented three naive solutions in order to investigate the potential of using a ML-based dispatcher to improve income fairness. If successful, this could then serve as a proof of concept and as an invite for improvements in future work.

`ml_poorest`

The dispatching algorithms come into play as soon as a driver has driven a customer to their destination. The first dispatcher, `ml_poorest`, works as follow: (1) it gets the date, time, weather and holiday information needed for the prediction from the Chicago class, (2) it uses the ML-model to predict the demand in 30 minutes. I chose this time offset by qualitatively checking on online maps services how long it takes to traverse the radius of the city for a few different routes without traffic, such as from the business center to O'Hare Intl. Airport.

Next (3), the algorithm compares the driver's income to the average income of all drivers. It standardizes the driver's income by subtracting the mean and dividing by the standard deviation of the income of all drivers, and does the same for a copy of the list of predicted demand for each city division. This operation facilitates inverse mappings from low income drivers to high demand areas.

Finally (4), the algorithm dispatches drivers based on the level of their income so far. First, it considers very poor drivers as having a standardized income of -0.5 or less, and dispatches them to the city division with the highest predicted demand. Secondly, it considers drivers with higher but still negative normalized income (i.e., between 0 and -0.5), and dispatches them to a randomly chosen city division between all of those that have a positive normalized predicted demand. Every other driver, meaning those richer than the average, are not dispatched in any way. In my case, this means using the post-ride behavior `"stay"` as defined in the simulator.

`ml_neutral_weighted`

This is a simplified version of the dispatcher above. It positions each driver which have just finished a ride in the same way. More precisely, it randomly chooses a city division within a chance proportional to the predicted demand at that city division. As before, I also account for the time to travel to that specific location and take the predictions 30 minutes later than the current time.

`ml_poorest_weighted`

Finally, this is another variation of the first dispatcher which skips dispatching the very poorest drivers to the best area, instead dispatching all poor drivers with normalized income < 0 to a randomly chosen city division weighted by its predicted demand in 30'. This means that this dispatcher combines features of both `ml_poorest` and `ml_neutral_weighted`.

An issue concerning all three of the dispatchers is that unscaled predictions could have been used for a more fine-grained mapping of a certain number of less well-off drivers to certain high-demand city areas, taking into consideration how many drivers are in said areas already. The problem is that this feature would require major modifications to how the simulator works and I could not do it due to time constraints. More on this in the discussion chapter.

3.4 Simulation parameters

The simulations have some parameters I chose to keep fixed, some given by the Chicago data, and some manually adjusted to achieve results for different scenarios. An overview of the simulation parameters used can be seen in table 3.3.

For consistency, I ran simulations using the same data from the second week of 2020 (06.01.2020 - 13.01.2020), with the same working hours for all drivers between 08:00 - 12:00 and 17:00 - 21:00, for a total of 42 working hours per week. Every simulation uses the same prices and costs as those used by Bokányi and Hannák [2020], i.e. an income of 2\$/trip and 1\$/km, and costs amounting to 0.08\$/Km. Admittedly, prices of ride-hailing services change from city to city due to different incomes and taxes, so these values have a generic representative purpose. The maximum customer waiting time is fixed at 10 minutes, after which requests are canceled. Drivers always move at a speed of 36 Km/h.

The simulator includes four basic matching algorithms to match drivers with customer requests: random, random limited, nearest and poorest. *Random* matches a request with a random available driver in the whole city, which in my case did not make much sense to use as nearly all requests would be cancelled due to the waiting time. *Random limited* does the same, but limiting the choice of available drivers to a radius around the request. *Nearest* chooses the nearest driver in a radius around the request, and *poorest* is an income equality-focused algorithm designed by Bokányi and Hannák [2020] which matches a request with the poorest driver in a fixed radius around it.

The radius itself used in the random limited, nearest and poorest matching algorithms is internally called `hard_limit`, and it is the maximal radius from a request in *du* at which drivers are searched for to be matched with that request. I used substantially larger radius limits than Bokányi and Hannák [2020] used in their simulations (3.5Km and 5Km compared to 1Km). I decided to do so due to the much bigger city size in my simulations (around 10 times bigger), and because of an issue in the simulator tied to this radius found by Dr. Nicolò Pagan, which I will explain in the discussion chapter.

As explained in the previous section, customer demand is given by the Chicago data, while the supply can be changed to simulate different driver densities. According to an article by Molina [2021], there were 65'689 Uber drivers working in April 2019 in total. As a safe guess for the more limited work hours chosen for the simulation,

Parameters	**Values taken**
Adjusted	
”start_date”	”2020-1-6”, ”2020-1-13”, ”2020-1-20”, ”2020-1-27”
”end_date”	”2020-1-13”, ”2020-1-20”, ”2020-1-27”, ”2020-2-3”
”num_taxis”	10000, 20000, 25000
”matching”	”nearest”, ”poorest”, ”random_limited”
”behaviour”	”stay”, ”ml_dispatcher_v2”, ”ml_dispatcher_distributor”
”ml_weighted”	True, False
”hard_limit”	35, 50
Fixed	
”use_chicago”	True
”chicago_grid_size”	49
”ml_model_path”	”data/random_forest_8_300_100_squares_49.pkl”
”working_hrs_1”	[8, 10]
”working_hrs_2”	[17, 21]
”max_time”	15120
”batch_size”	15120
”initial_conditions”	”home”
”price_fixed”	2
”price_per_dist”	1
”cost_per_unit”	0.008
”log”	False
”show_map_labels”	False
”show_pending”	False
”show_plot”	False
”max_request_waiting_time”	60
”avg_request_lengths”	22.7
”request_rate”	1
”reset”	”false”
”geom”	0
”request_origin_distributions”	[”location”: [20, 20], ”strength”: 1, ”sigma”: 10]

Table 3.3: Adjusted and fixed simulation parameters

I ran simulations with 10'000 and 20'000 drivers mainly, for a driver density of 7.57 and 15.13 respectively (taxis/Km2). These densities are also in the range of those found by Bokányi and Hannák [2020] for big cities like Chicago. To validate the promising results of one of the dispatchers, I finally also ran some simulations with 25'000 drivers for a density of 18.92 taxis/Km2 and with data from the remaining weeks of January 2020 and the first week of March 2020. In the next chapter I will go over the results of these simulations.

Chapter 4

Results

This chapter presents the final results of the simulations ran with the parameters explained in the previous section. Figure 4.1 shows an overview of the income distributions with all tested combinations of matching algorithms and dispatchers, in a simulation with 10'000 taxis over a week. The slimmer the curve, the better the income equality, as more drivers are situated in a closer range of total earnings. In the next sections, I will take a closer look at the performance of each of my dispatchers.

4.1 Dispatchers

4.1.1 `ml_poorest`

`ml_poorest` is the first of the three dispatchers. Although it does not have a poor performance, it is not really able to prove any tangible improvement in equality or total utility (figures in table 4.1). Without the poorest matching algorithm, it

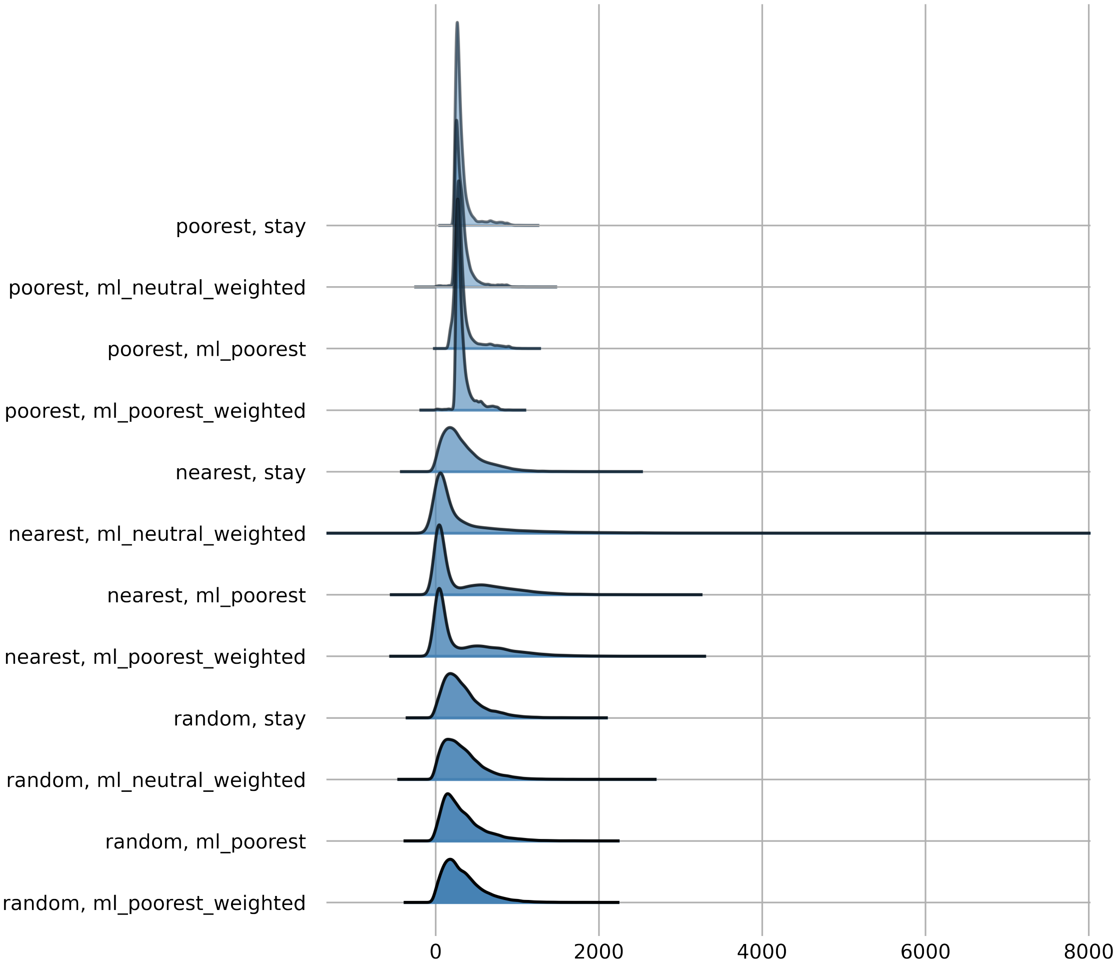

Figure 4.1: An overview of driver income distributions with all combinations of tested matching algorithms and dispatchers. Simulation with 20'000 taxis over a week. The first term of the labels is the matching strategy, the second the dispatcher or "stay" for no dispatching. X axis: driver income distribution

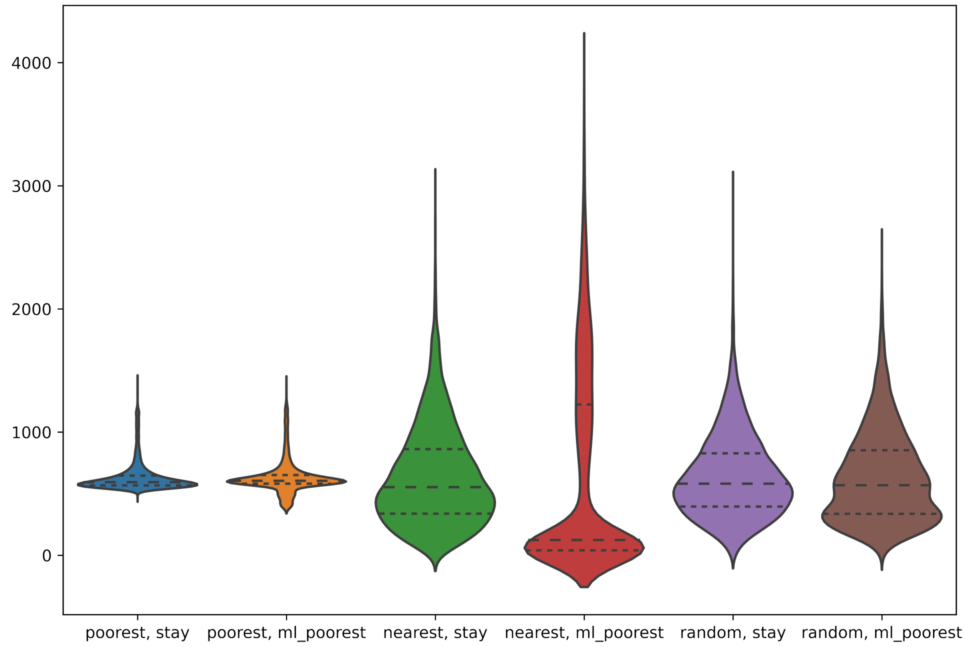

Figure 4.2: Income distributions for `ml_poorest` compared to dispatch-less runs, with all matching algorithms. A flatter distribution means better income equality, higher distributions (by volume) have higher total income. Simulation from 2020-1-6 to 2020-1-13, 7.57 taxis/Km2, search radius 50.

performs slightly worse than without using a dispatcher due to driving costs and the dispatched poorest drivers not being prioritized in the queue for matches.

Table 4.1 shows the data corresponding to the violin plot in figure 4.2. The poorest matching algorithm without dispatcher performs the best in terms of fairness (Gini coefficient 0.08), although its counterpart using `ml_poorest` performs only slightly worse (Gini 0.10). Notice that the latter has a fatter lower tail in 4.2, meaning that this dispatcher causes a group of drivers to earn a bit less than the vast majority. I

matching	dispatching	gini	atkinson	20/20	total utility
poorest	stay	0.08	0.01	1.26	6378326.86
poorest	`ml_poorest`	0.1	0.01	1.26	6343721.64
random	stay	0.29	0.07	1.57	6373486.47
random	`ml_poorest`	0.32	0.08	1.65	6330295.62
nearest	stay	0.34	0.1	1.69	6380972.12
nearest	`ml_poorest`	0.64	0.39	2.52	6368770.79

Table 4.1: Income inequality scores for `ml_poorest` compared to the scores without dispatching, sorted by Gini score best to worst. Corresponds to violin plot 4.2. Simulation from 2020-1-6 to 2020-1-13, 7.57 taxis/Km2, search radius 50.

suppose that this may be due to `ml_poorest` not giving a sufficient edge to poorer drivers, while also making them lose money through the driving costs of reaching the dispatch area, as opposed to better-off drivers remaining stationary after a ride. This tail is also the probable cause of the lower total income (total utility) of poorest matching + `ml_poorest`.

Moreover, `ml_poorest` flattens out when used with the nearest matching algorithm. This is also mirrored in the lower associated Gini of 0.64; as a compression, the run without dispatcher has a much lower Gini, namely 0.34. The total utility is lower as well. The relationships between these results are similar in other simulations, such as with double the amount of drivers (20'000). These additional results are available in the aggregated results table C.1.

4.1.2 `ml_neutral_weighted`

`ml_neutral_weighted` dispatches drivers in the same way, irrespective to their current level of income. Surprisingly, results in simulations are not better than those without

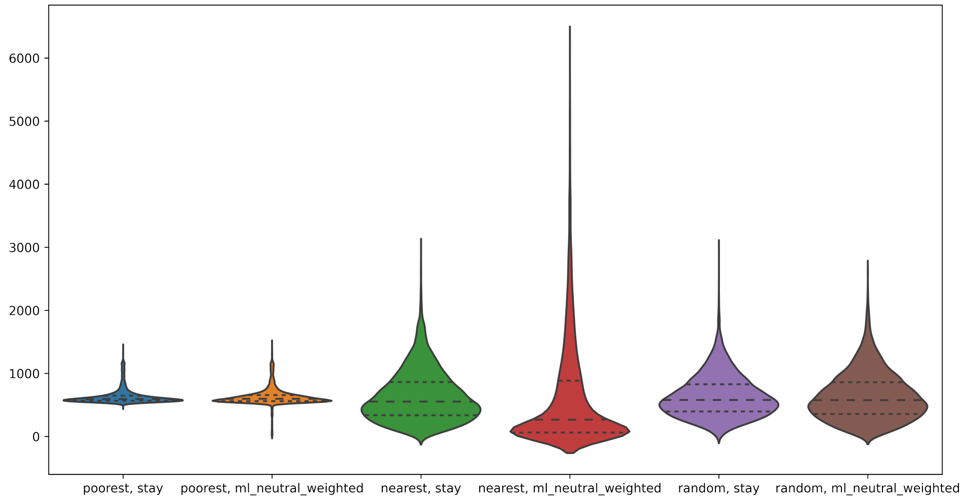

Figure 4.3: Income distributions for `ml_neutral_weighted` compared to dispatchless runs, with all matching algorithms. A flatter distribution means better income equality, higher distributions (by volume) have higher total income. Simulation from 2020-1-6 to 2020-1-13, 7.57 taxis/Km2, search radius 50.

it, behaving similarly to `ml_poorest`. Interestingly, both figure 4.3 and figure 4.1 show that `ml_neutral_weighted` has some extreme outliers with nearest matching compared to the other two dispatchers and runs without dispatchers. Apart from the fact that nearest matching consistently has much higher outliers than random limited and poorest, the additional boost by this dispatcher might be explained by some drivers already starting in a good spot with high demand, and luckily being dispatched in good spots again a few times, as the number of completed trips per driver are not very high in these simulations at a mere average of 4.9 trips per driver. The other two dispatchers never dispatch drivers with positive standardized income.

matching	dispatching	gini	atkinson	20/20	total utility
poorest	stay	0.08	0.01	1.26	6378326.86
poorest	`ml_neutral_weighted`	0.1	0.01	1.26	6339908.65
random	stay	0.29	0.07	1.57	6373486.47
random	`ml_neutral_weighted`	0.33	0.09	1.63	6377436.81
nearest	stay	0.34	0.1	1.69	6380972.12
nearest	`ml_neutral_weighted`	0.63	0.34	2.57	6220723.33

Table 4.2: Income inequality scores for `ml_neutral_weighted` compared to the scores without dispatching, sorted by Gini score best to worst. Total utility is the cumulative income of all drivers. Corresponds to violin plot 4.3. Simulation from 2020-1-6 to 2020-1-13, 7.57 taxis/Km2, search radius 50.

4.1.3 `ml_poorest_weighted`

In short, it appears that my first two ML-based naive dispatchers, namely `ml_poorest` and `ml_neutral_weighted`, are too simplistic to show any improvement over not using a dispatcher at all. However, this last variation, namely `ml_poorest_weighted`, was finally able to achieve some promising results.

As shown in 4.5, `ml_poorest_weighted` + poorest matching consistently beats poorest matching with no dispatching by a small margin, as long as the taxi density is high enough. With 10'000 taxis (7.57 taxis/Km2) they have basically the same Gini, the former having 0.086 and the latter 0.085, although the dispatcher had a 1.5% higher total utility. With 20'000 taxis (15.13 taxis/Km2), which is a closer estimate of the true number of ride-hailing drivers in Chicago that week, we start seeing some interesting and consistent results.

Figure 4.4 demonstrates the promising results of `ml_poorest_weighted` + poorest

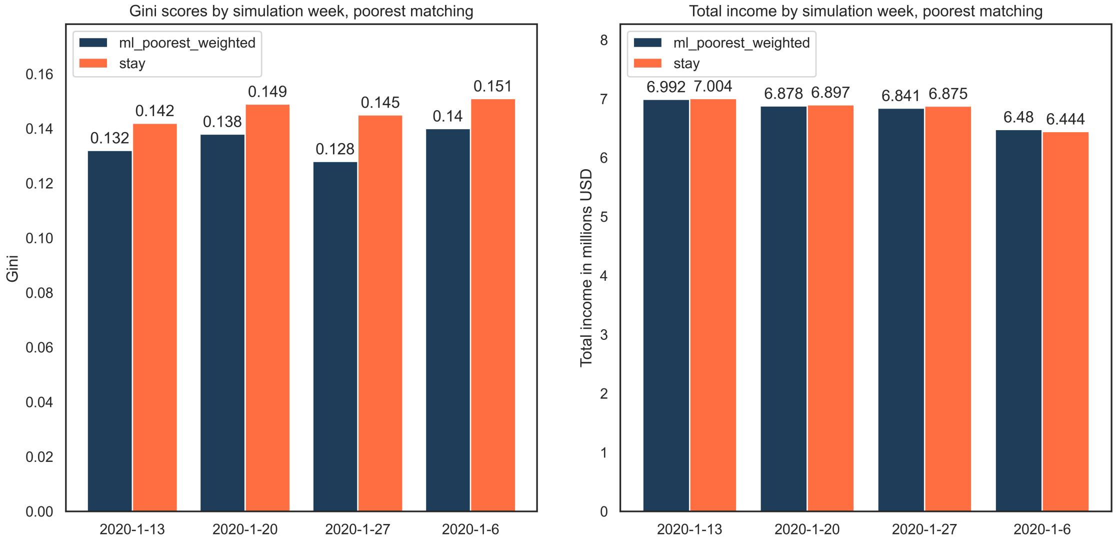

Figure 4.4: Scores of `ml_poorest_weighted` by simulation week, poorest matching, 15.13 taxis/Km2, search radius 50. The x axis is the starting day of the 7-day week. Left: Gini, right: total income.

start	dispatching	gini	atkinson	20/20	tot income $M	missed req %	idle %
2020-1-13	`ml_poorest_weighted`	0.132	0.017	1.376	6.992	0.23	0.0
2020-1-13	stay	0.142	0.02	1.44	7.004	0.36	0.0
2020-1-20	`ml_poorest_weighted`	0.138	0.018	1.397	6.878	0.17	0.0
2020-1-20	stay	0.149	0.022	1.463	6.897	0.24	0.0
2020-1-27	`ml_poorest_weighted`	0.128	0.016	1.362	6.841	0.16	0.0
2020-1-27	stay	0.145	0.021	1.443	6.875	0.18	0.0
2020-1-6	`ml_poorest_weighted`	0.14	0.02	1.407	6.48	0.14	0.0
2020-1-6	stay	0.151	0.023	1.461	6.444	0.45	0.0

Table 4.3: Scores of `ml_poorest_weighted` by simulation week, corresponding to the grouped bar plots in figure 4.4. Poorest matching, 15.13 taxis/Km2, search radius 50.

matching, consistent in simulations over different weeks in the dataset. The average income equality improvement over dispatch-less poorest matching is measured in a 8.35% decrease in the Gini coefficient, a 17.44% lower Atkinson, and a 4.56% lower 20/20 ratio.

Of course, these results hold only for the combination `ml_poorest_weighted` + poorest matching algorithm. Figure 4.5 shows that with random limited matching, `ml_poorest_weighted` causes a slightly worse Gini than dispatch-less, while with nearest matching the Gini is very high at ≈ 0.6. The figure also shows how the income equality gains achieved with the dispatcher increase with a higher taxi density, from 15.13 taxis/Km2 (left plot) to 18.92 taxis/km2 (right plot).

In conclusion, it seems that `ml_poorest_weighted` is able to exploit the decent basic idea behind `ml_poorest`, namely what in figure 4.2 seems like a better income equality, while also getting rid of the "fat tail" of poor drivers in 4.2 mentioned before. I speculate that the reason behind this higher economic efficiency is the dispatching of poorer drivers with the weighted probability technique introduced in `ml_neutral_weighted`, as poor drivers are much better distributed in

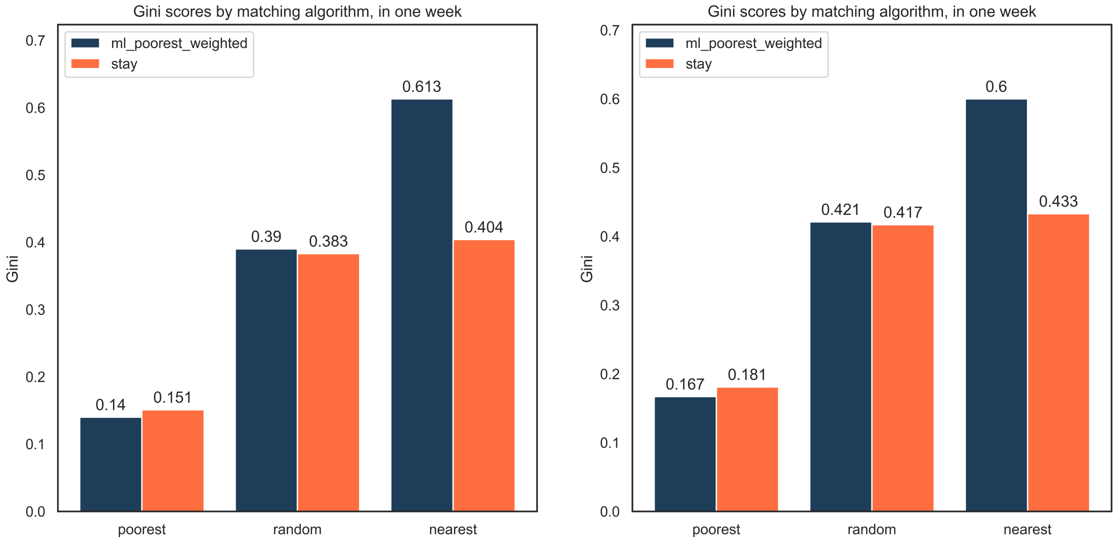

Figure 4.5: Gini coefficient with `ml_poorest_weighted` by matching algorithm, simulation from 2020-1-6 to 2020-1-13 with 15.13 taxis/Km2 (left) and 18.92 taxis/km2 (right)

the high-demand areas in relation to the predicted demands. At the same time, `ml_poorest_weighted` gains an edge in income equality over `ml_neutral_weighted` by only dispatching poor drivers instead of all of them equally, giving an advantage to the poor ones. Based on the consistent results in 4.3 and 4.4, this gained edge is sufficient for `ml_poorest_weighted` to noticeably overtake the dispatch-less *poorest matching* algorithm, which in my opinion shows great promise for the technique.

Chapter 5

Discussion and Limitations

The supervised learning prediction-based dispatchers `ml_poorest` and `ml_neutral_weighted` have failed to show any advantage in combination with all driver-customer matching algorithms (except random unlimited, which was not tested). I think that the main reason for this is that dispatching drivers to high-demand areas, while at the same time not relieving those areas of better-off drivers, has very slim margins of income fairness improvement that can be exploited without detracting from the total utility. As it is not possible to dispatch drivers to clearly less busy areas than those they find themselves in, due to them self-strategizing and not following the dispatching order, there is a limit to how many drivers can be dispatched to high demand areas before other areas run out of supply and economic inefficiencies arise.

Looking at the first overview distribution in 4.1, it is clear that the largest deviations in income fairness come from the choice of matching algorithm, and my simulations using real-world data can only attest to how good the *poorest* matching algorithm by Bokányi and Hannák [2020] is for income equality, while achieving to maintain virtually the same total utility.

On the other hand, my `ml_poorest_weighted` dispatcher has shown promising results when combined with the poorest matching algorithm, with an 8.35% better Gini coefficient on average than without it. This result suggests that it may in fact be possible to exploit that slim improvement margin, and use a business-proven and efficient supervised learning approach to increase driver income equality in ride-hailing platforms. Of course, the number of simulations with different parameters and data could have been much higher to better validate this result, but I was limited by the hardware and by time, as a single simulation over one week of data (42 hours) could take as long as 2-3 hours to complete.

As mentioned before in my thesis, there are some significant limitations in my work. Perhaps the most important, is the simplicity of the dispatcher. More precisely, in my work I only use dispatchers that are based on the *distributions* of requests rather than the actual counts. As a result, the ML-model provides more information than it is used by the naive dispatchers. This additional information could be key in making sure we only dispatch the right amount of drivers; using it could ensure that we do not create competition in the areas we dispatch drivers. In fact, this is a complex optimization problem on its own, especially considering that a dispatching decision also entails running costs for the driver and a lower total utility. Even more complexity is added by the online character of the matching problem, with requests continuously coming in and getting assigned to drivers. This complexity of the optimization problem explains why the recent literature often looks at reinforcement learning to find a solution, a technique that is particularly suited for these online allocation problems.

A second limitation is the behavior of non-dispatched drivers in the simulations. I think that the third ML dispatcher could have an even bigger equality or total utility advantage if it was tested in simulations where the non-dispatched drivers cruise randomly, or roughly towards important areas like the city center or the airport: in fact, in my simulations they just wait on the spot. In the former scenario every driver loses money due to driving costs, but the poorer dispatched drivers are more strategically dispatched to high-demand zones, with the poorest matching algorithm giving them priority for customers over non-dispatched drivers cruising around and burning fuel. The lack of such simulations is a limitation of this study, as I could not run another whole set of simulations. Moreover, although minor, continuous cruising is yet to be implemented in the simulator: The only current possibility is to dispatch drivers to a specific location in the city with no further planned movements afterwards, which is a bit unrealistic if the drivers cruise to high-demand, congested areas with no parking spots.

A third limitation is that dispatching and matching are still separated in my solution, creating a concern about whether the poorer dispatched drivers would trust that they would get assigned to a customer once reaching their dispatch area. On one hand, my results show that they would statistically be better off following the dispatcher suggestions, but on the other it is always hard to predict end-user behavior with respect to a software. And this is all assuming a ride-hailing company would be willing to take a risk and implement not only a different matching algorithm, but a dispatcher as well for no added monetary gain. By combining the dispatching with the matching, it would probably be slightly easier to convince both drivers and the ride-hailing companies to adopt the solution.

A fourth limitation is the single design of the ML model. It is possible that a classic single-output solution would have performed better, although probably at a cost of some performance in the dispatching phase of the solution. It is also possible that another regression model outside of those under scrutiny would have performed better than the Random Forest regression, as I logistically could not test every algorithm in existence.

There is one additional limitation tied to the simulator. This is a relatively hard to solve issue that was found by Nicolò Pagan: the search radius for the matching algorithms, defined as ”hard limit” in the simulator, makes it so that the nearest algorithm does not continue looking for a driver outside of the radius when no drivers are found within. Based on the definitions in the study by Bokányi and Hannák [2020], the hard limit was designed for the poorest matching algorithm to avoid looking for the poorest driver in the entire city. Yet, the radius is limiting also the nearest matching algorithm, which could cause problematic results with either a small driver density or a small radius.

A further important issue caused by dispatching in general is the ecological impact of drivers driving without passengers to reach their assigned zones, and the additional congestion caused to the city, which is already a big problem as stated by the Chicago mayor Lightfoot. A possible solution could be given by limiting the number of drivers to a fixed and sustainable quota. The slow switch to hybrid and electric cars can help as well. Finally, further incentives for actual ride-sharing can be of help too. From the perspective of the dispatcher, maybe one that could plan for tactical waiting times and traffic conditions could help offsetting pollution and congestion as well.

Chapter 6

Conclusion

To reiterate, the goal of this thesis was to attempt increasing income equality for drivers of ride-hailing services using a machine learning based driver dispatcher. In trying to achieve this, after comparing the performance of different ML algorithms for the problem I successfully implemented an accurate supervised learning model based on Random Forest regression to predict customer demand in customizable divisions of a city - Chicago in this case. I then adapted a taxi simulator developed by Bokányi and Hannák [2020] to use real-world anonymized ride-hailing (or taxi) trip data to generate customer demand, instead of using random distributions to do so. In the process, I was able to find and correct a bug in said simulator which entailed some coordinates being stored ”upside down” in a data structure, potentially causing a failure in simulations of non-squared cities. All the code developed and used for this thesis is publicly available online, making this work reproducible and enabling other interested researchers to implement variations of my work. [1]

---

[1] The code is available on GitHub at https://github.com/ianskoo/ride-hailing-income-fairness.git

Finally, I developed three naive rule-based dispatchers that could use the predictions of the ML model to redistribute drivers in the city. The first two, `ml_poorest` and `ml_neutral_weighted`, have failed to reach any meaningful result. The third, `ml_poorest_weighted`, has achieved an average improvement of 8.35% in the Gini coefficient when combined with the *poorest* matching algorithm developed by Bokányi and Hannák [2020], as compared said matching algorithm without dispatching. The total utility was also conserved. While testing the dispatchers against the three matching algorithms without dispatching, I could also validate the effectiveness of the poorest matching algorithm found by Bokányi and Hannák [2020] with real-life data by plotting virtually identical income distributions comparing nearest, random and poorest matching strategies as in their study. At a higher level, this confirms the usefulness of agent-based modelling to understand and anticipate real-world effects.

All this work would not have been possible without the diverse and thorough literature review. As motioned in the respective chapter, this branched over (a) fairness, (b) machine learning, (c) optimization, and (d) domain-specific ride-hailing work. Key to the direction of this thesis was the work of Sühr et al. [2019] who argued that fairness of matches should be measured over a longer time period as opposed to trying to make every match individually fair. I followed this idea by simulating driver-customer matches over a week, while at the same time diverging from their methods of finding a solution to increase fairness, branching out into machine learning.

Currently, a lot of work is being done to optimize ride matching with the overall utility as main goal. As the optimization problem is online, meaning that new requests come in continuously, many of these attempts use solutions based on reinforcement learning. Though I initially considered the idea of following this lead, reinforcement learning is very resource intensive and hardly used outside of research, at least for now. Coupled with the fact that I had already learned supervised learning from scratch for this work and learning reinforcement learning on top of that was logistically near-impossible, I decided to attempt reaching a more business-ready and lightweight solution with a supervised learning based solution.

The previous chapter also listed the main limitations of this work. The most important is the simplicity of the dispatcher, which is based on the *distributions* of requests rather than the actual counts. As a result, the dispatcher doesn't exploit all of the information provided by the ML model. Secondly, the simulations could span more data and more diverse non-dispatched driver behavior outside of waiting in place. Other valid options for the demand prediction could be explored as well.

In conclusion, as mentioned before I was able to show an improvement in driver income fairness. Nonetheless, although the demand prediction model seems solid, the `ml_poorest_weighted` dispatcher was just a naive and perhaps lucky attempt at reaching some results. Having proved that the potential exists, with some further small interventions to the simulator in which the dispatcher is integrated, and by taking into consideration the counts of drivers in each area of the city and the actual counts of predicted requests into an equation to optimize, I am sure that further improvements could be scraped towards income equality and total utility. Secondarily, the results reached with `ml_poorest_weighted` could be further validated with differently parametrized simulations, provided the time and more appropriate hardware resources.

Bibliography

Robert Channick. Too many Uber drivers? Chicago cabbies and ride-share workers join forces, urge cap on Uber and Lyft cars. URL https://www.chicagotribune.com/business/ct-biz-chicago-taxi-ride-share-drivers-limit-20181030-story.html. Section: , Business.

Noam Scheiber. Uber and Lyft Ramp Up Legislative Efforts to Shield Business Model. *The New York Times*, June 2021. ISSN 0362-4331. URL https://www.nytimes.com/2021/06/09/business/economy/uber-lyft-gig-workers-new-york.html.

Kate Conger and Noam Scheiber. California Bill Makes App-Based Companies Treat Workers as Employees. *The New York Times*, September 2019. ISSN 0362-4331. URL https://www.nytimes.com/2019/09/11/technology/california-gig-economy-bill.html.

Eszter Bokányi and Anikó Hannák. Understanding Inequalities in Ride-Hailing Services Through Simulations. *Scientific Reports*, 10(1):6500, April 2020. ISSN 2045-2322. doi: 10.1038/s41598-020-63171-9. URL https://www.nature.com/articles/s41598-020-63171-9. Bandiera_abtest: a Cc_license_type: cc_by Cg_type: Nature Research Journals Number: 1 Primary_atype: Research Publisher:

Nature Publishing Group Subject_term: Computational science;Socioeconomic scenarios Subject_term_id: computational-science;socioeconomic-scenarios.

Tom Sühr, Asia J. Biega, Meike Zehlike, Krishna P. Gummadi, and Abhijnan Chakraborty. Two-Sided Fairness for Repeated Matchings in Two-Sided Markets: A Case Study of a Ride-Hailing Platform. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3082–3092, Anchorage AK USA, July 2019. ACM. ISBN 978-1-4503-6201-6. doi: 10.1145/3292500.3330793. URL `https://dl.acm.org/doi/10.1145/3292500.3330793`.

Minne Li, Zhiwei Qin, Yan Jiao, Yaodong Yang, Jun Wang, Chenxi Wang, Guobin Wu, and Jieping Ye. Efficient Ridesharing Order Dispatching with Mean Field Multi-Agent Reinforcement Learning. In *The World Wide Web Conference*, WWW '19, pages 983–994, New York, NY, USA, May 2019. Association for Computing Machinery. ISBN 978-1-4503-6674-8. doi: 10.1145/3308558.3313433. URL `https://doi.org/10.1145/3308558.3313433`.

Arslan Ali Syed, Bernd Kaltenhaeuser, Irina Gaponova, and Klaus Bogenberger. Asynchronous Adaptive Large Neighborhood Search Algorithm for Dynamic Matching Problem in Ride Hailing Services. In *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, pages 3006–3012, October 2019. doi: 10.1109/ITSC.2019.8916943.

Sebastian Stein, Mateusz Ochal, Ioana-Adriana Moisoiu, Enrico Gerding, Raghu Ganti, Ting He, and Tom La Porta. Strategyproof reinforcement learning for online resource allocation. pages 1296–1304. University of Auckland, May 2020. doi: 10.5555/3398761.3398911. URL `https://eprints.soton.ac.uk/438382/`. Num Pages: 9.

Divine Carson-Bell, Mawutor Adadevoh-Beckley, and Kendra Kaitoo. Demand Prediction of Ride-Hailing Pick-Up Location Using Ensemble Learning Methods. *Journal of Transportation Technologies*, 11(02):250–264, 2021. ISSN 2160-0473, 2160-0481. doi: 10.4236/jtts.2021.112016. URL `https://www.scirp.org/journal/doi.aspx?doi=10.4236/jtts.2021.112016`.

Stephen M Zoepf, Stella Chen, Paa Adu, and Gonzalo Pozo. The economics of ride-hailing: Driver revenue, expenses and taxes. *CEEPR WP*, 5:1–38, 2018.

Le Chen, Alan Mislove, and Christo Wilson. Peeking Beneath the Hood of Uber. In *Proceedings of the 2015 Internet Measurement Conference*, IMC '15, pages 495–508, New York, NY, USA, October 2015. Association for Computing Machinery. ISBN 978-1-4503-3848-6. doi: 10.1145/2815675.2815681. URL `https://doi.org/10.1145/2815675.2815681`.

Uber, a. URL `https://www.uber.com/us/en/marketplace/pricing/surge-pricing/`.

How Uber surge pricing really works. *Washington Post*, b. ISSN 0190-8286. URL `https://www.washingtonpost.com/news/wonk/wp/2015/04/17/how-uber-surge-pricing-really-works/`.

G Ayorkor Mills-Tettey, Anthony Stentz, and M Bernardine Dias. The dynamic hungarian algorithm for the assignment problem with changing costs. *Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-07-27*, 2007.

City of Chicago. Transportation Network Providers - Trips | City of Chicago | Data Portal. URL `https://data.cityofchicago.org/Transportation/Transportation-Network-Providers-Trips/m6dm-c72p`.

Shan Liu, Hai Jiang, and Zhe Chen. Quantifying the impact of weather on ride-hailing ridership: Evidence from Haikou, China. *Travel Behaviour and Society*, 24:257–269, July 2021. ISSN 2214-367X. doi: 10.1016/j.tbs.2021.04.002. URL https://www.sciencedirect.com/science/article/pii/S2214367X21000302.

NCEI. Past Weather | National Centers for Environmental Information (NCEI). URL https://www.ncei.noaa.gov/access/past-weather/chicago.

officeholidays.com. Federal Holidays in Illinois in 2022. URL https://www.officeholidays.com/countries/usa/illinois/2022.

z_ai. The Good, the Bad, and the Ugly: Supervised, Unsupervised and Reinforcement Learning, March 2021. URL https://towardsdatascience.com/the-good-the-bad-and-the-ugly-supervised-unsupervised-and-reinforcement-learning-

Andrew Ng. Machine Learning. URL https://www.coursera.org/learn/machine-learning.

Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, Beijing China ; Sebastopol, CA, 2nd edition edition, October 2019. ISBN 978-1-4920-3264-9.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

1.11. Ensemble methods, c. URL https://scikit-learn/stable/modules/ensemble.html.

Nina Molina. Uber, Lyft riders are paying more and waiting longer, June 2021. URL https://chicago.suntimes.com/2021/6/22/22465111/uber-lyft-drivers-down-users-face-higher-prices-longer-wait-times. Section: Transportation.

Lori E Lightfoot. TRANSPORTATION NETWORK PROVIDERS AND CONGESTION IN THE CITY OF CHICAGO. page 20.

Appendix A

`ml_poorest` and `ml_poorest_weighted`

```
def ml_dispatcher_poorest(self, curr_taxi_id, weighted: bool):
    """Redistribute poor drivers to high demand city areas."""

    # Get predicted demand
    predictions = self.city.chicago.predict_demand(self.time)
    if predictions is None:
        return None
    else:
        predictions = predictions[0]

    # Get mean and std dev of all drivers incomes
    taxis_list_snapshot = self.taxis
    curr_incomes = [self.eval_taxi_income(taxi_id) for taxi_id in taxis_list_snapshot]
    mean_tot_income = np.mean(curr_incomes)
    std_dev_tot_income = np.std(curr_incomes)

    # Standardize current idle driver’s income
    curr_taxi_income = self.eval_taxi_income(curr_taxi_id)
    curr_stdized_taxi_income = (curr_taxi_income - mean_tot_income) / std_dev_tot_income

    # Compute mean and std dev of prediction vector, then standardize its values
    mean_pred_demand = np.mean(predictions)
    std_dev_pred_demand = np.std(predictions)
```

```
stdized_predictions = [(pred - mean_pred_demand) / std_dev_pred_demand for pred in predictions]

# Dispatch drivers
dispatch_district_idx = None

if curr_stdized_taxi_income >= 0:
    # Leave all drivers richer than the average where they are
    return None

if not weighted:
    if curr_stdized_taxi_income < -0.5:
        # Very poor, dispatch to district with most demand (best district)
        dispatch_district_idx = np.where(stdized_predictions == max(stdized_predictions))[0][0]

    elif curr_stdized_taxi_income < 0:
        # Poorer than average, randomly dispatch to better than mean districts
        next_best_demands = [demand for demand in stdized_predictions if demand > 0]
        chosen_demand = random.choice(next_best_demands)
        dispatch_district_idx = np.where(stdized_predictions == chosen_demand)[0][0]

else:
    if curr_stdized_taxi_income < 0:
        # Send poorer than average drivers randomly to areas weighted by their predicted demand
        predictions_sum = sum(predictions)
        dispatch_weights = [pred / predictions_sum for pred in predictions]
        dispatch_district_idx = np.where(
            predictions == np.random.choice(predictions, p=dispatch_weights))[0][0]

# Get coordinates and dispatch driver
long, lat = self.city.centered_districts_coords[dispatch_district_idx]
dispatch_coords = self.city.chicago.coord_to_grid(lat, long)
self.go_to_base(curr_taxi_id, dispatch_coords)
```

Appendix B

ml_neutral_weighted

```
def ml_dispatcher_weighted_distributor(self, curr_taxi_id: int):
    """Redistribute drivers to city areas based on predicted future demand, independently of
    their current income. Dispatch a driver having curr_taxi_id to an area with a probability
    based on predicted demand in 30’. """

    # Get predicted demand
    predictions = self.city.chicago.predict_demand(self.time)
    if predictions is None:
        return None
    else:
        predictions = predictions[0]

    # Compute mean and std dev of prediction array, then standardize its values
    predictions_sum = sum(predictions)
    dispatch_weights = [pred / predictions_sum for pred in predictions]

    # Choose one of the predictions based on their distribution
    # dispatch_district_idx = predictions.index(np.random.choice(predictions, p=dispatch_weights))
    dispatch_district_idx = np.where(
        predictions == np.random.choice(predictions, p=dispatch_weights))[0][0]

    # Get coordinates and dispatch driver
    long, lat = self.city.centered_districts_coords[dispatch_district_idx]
```

```
dispatch_coords = self.city.chicago.coord_to_grid(lat, long)
self.go_to_base(curr_taxi_id, dispatch_coords)
```

Appendix C

Results table of all simulations

Table C.1: Results of all simulations, ordered by starting date and Gini coefficient. Utility $M refers to the cumulative income in millions, missed % is the percentage of missed customer requests, and idle % is the percentage of completely idle drivers.

start	matching	dispatching	gini	atkinson	20/20	utility $M	missed %	idle %
2020-1-13	poorest	ml_poorest_weighted	0.132	0.017	1.376	6.992	0.23	0.0
2020-1-13	poorest	stay	0.142	0.02	1.44	7.004	0.36	0.0
2020-1-13	random	stay	0.371	0.111	1.789	7.007	0.33	1.03
2020-1-13	random	ml_poorest_weighted	0.391	0.123	1.81	7.014	0.24	1.08
2020-1-13	nearest	stay	0.397	0.125	1.853	7.044	0.13	1.2
2020-1-13	nearest	ml_poorest_weighted	0.624	0.349	2.515	7.066	0.04	0.15
2020-1-20	poorest	ml_poorest_weighted	0.138	0.018	1.397	6.878	0.17	0.0
2020-1-20	poorest	stay	0.149	0.022	1.463	6.897	0.24	0.0
2020-1-20	random	stay	0.383	0.117	1.813	6.878	0.3	1.26
2020-1-20	random	ml_poorest_weighted	0.397	0.127	1.815	6.899	0.13	1.41
2020-1-20	nearest	stay	0.398	0.126	1.857	6.921	0.17	1.25
2020-1-20	nearest	ml_poorest_weighted	0.626	0.353	2.51	6.95	0.02	0.17
2020-1-27	poorest	ml_poorest_weighted	0.128	0.016	1.362	6.841	0.16	0.0

start	matching	dispatching	gini	atkinson	20/20	utility $M	missed %	idle %
2020-1-27	poorest	stay	0.145	0.021	1.443	6.875	0.18	0.0
2020-1-27	random	stay	0.375	0.111	1.789	6.86	0.24	1.4
2020-1-27	random	ml_poorest_weighted	0.383	0.118	1.787	6.861	0.13	1.24
2020-1-27	nearest	stay	0.394	0.124	1.841	6.9	0.12	1.21
2020-1-27	nearest	ml_poorest_weighted	0.613	0.337	2.455	6.913	0.02	0.18
2020-1-6	poorest	stay	0.051	0.003	1.157	6.347	0.81	0.0
2020-1-6	poorest	stay	0.085	0.008	1.264	6.378	0.7	0.0
2020-1-6	poorest	ml_poorest_weighted	0.086	0.009	1.237	6.479	0.15	0.0
2020-1-6	poorest	stay	0.09	0.008	1.276	6.37	0.77	0.0
2020-1-6	poorest	stay	0.091	0.008	1.272	6.327	0.98	0.0
2020-1-6	poorest	ml_neutral_weighted	0.096	0.011	1.259	6.34	0.63	0.0
2020-1-6	poorest	ml_poorest_weighted	0.099	0.012	1.26	6.458	0.31	0.0
2020-1-6	poorest	ml_poorest	0.104	0.011	1.261	6.344	0.71	0.0
2020-1-6	poorest	ml_poorest_weighted	0.14	0.02	1.407	6.48	0.14	0.0
2020-1-6	poorest	stay	0.151	0.023	1.461	6.444	0.45	0.0
2020-1-6	poorest	ml_neutral_weighted	0.153	0.023	1.407	6.4	0.39	0.0
2020-1-6	poorest	ml_poorest_weighted	0.167	0.026	1.481	6.483	0.13	0.0
2020-1-6	poorest	ml_poorest	0.167	0.027	1.46	6.416	0.43	0.0
2020-1-6	poorest	stay	0.181	0.032	1.538	6.488	0.29	0.0
2020-1-6	random	stay	0.206	0.035	1.39	6.319	1.05	0.0
2020-1-6	nearest	stay	0.287	0.07	1.534	6.344	0.87	0.02
2020-1-6	random	stay	0.288	0.067	1.587	6.324	1.01	0.07
2020-1-6	random	stay	0.288	0.068	1.57	6.373	0.72	0.18
2020-1-6	random	stay	0.288	0.068	1.584	6.359	0.82	0.04
2020-1-6	random	ml_poorest_weighted	0.313	0.082	1.61	6.441	0.38	0.17
2020-1-6	random	ml_poorest	0.323	0.084	1.648	6.33	0.81	0.09
2020-1-6	random	ml_neutral_weighted	0.33	0.093	1.634	6.377	0.52	0.31
2020-1-6	nearest	stay	0.342	0.096	1.688	6.335	1.01	0.26
2020-1-6	nearest	stay	0.342	0.096	1.686	6.361	0.85	0.2
2020-1-6	nearest	stay	0.344	0.098	1.691	6.381	0.73	0.21
2020-1-6	random	ml_poorest_weighted	0.375	0.116	1.743	6.435	0.43	0.26
2020-1-6	random	stay	0.383	0.117	1.81	6.437	0.47	1.42
2020-1-6	random	ml_poorest_weighted	0.39	0.122	1.807	6.497	0.14	1.48
2020-1-6	random	ml_neutral_weighted	0.391	0.122	1.789	6.424	0.3	1.65
2020-1-6	random	ml_poorest	0.401	0.127	1.866	6.408	0.51	1.39
2020-1-6	nearest	stay	0.404	0.13	1.871	6.457	0.43	1.32
2020-1-6	random	stay	0.417	0.133	1.895	6.484	0.32	3.0
2020-1-6	random	ml_poorest_weighted	0.421	0.137	1.882	6.498	0.14	2.74
2020-1-6	nearest	stay	0.433	0.144	1.949	6.495	0.29	2.65

start	matching	dispatching	gini	atkinson	20/20	utility $M	missed %	idle %
2020-1-6	nearest	ml_poorest_weighted	0.6	0.311	2.435	6.542	0.03	0.58
2020-1-6	nearest	ml_poorest_weighted	0.613	0.334	2.47	6.499	0.24	0.26
2020-1-6	nearest	ml_poorest_weighted	0.623	0.371	2.425	6.388	0.7	0.02
2020-1-6	nearest	ml_poorest	0.624	0.346	2.524	6.441	0.46	0.2
2020-1-6	nearest	ml_poorest_weighted	0.625	0.372	2.437	6.401	0.63	0.0
2020-1-6	nearest	ml_neutral_weighted	0.626	0.34	2.574	6.221	1.95	0.0
2020-1-6	nearest	ml_poorest	0.642	0.394	2.522	6.369	0.75	0.0
2020-1-6	nearest	ml_neutral_weighted	0.668	0.371	2.867	6.366	0.71	0.0
2020-5-4	poorest	stay	0.816	0.197	1.492	0.249	0.15	66.14
2020-5-4	poorest	ml_poorest_weighted	0.816	0.193	1.463	0.248	0.09	66.78
2020-5-4	poorest	ml_neutral_weighted	0.818	0.198	1.468	0.245	0.2	66.67
2020-5-4	nearest	ml_neutral_weighted	0.822	0.202	1.457	0.246	0.0	66.66
2020-5-4	random	ml_neutral_weighted	0.84	0.193	1.355	0.245	0.2	69.92
2020-5-4	nearest	ml_poorest	0.843	0.205	1.364	0.249	0.0	70.14
2020-5-4	nearest	ml_poorest_weighted	0.843	0.206	1.376	0.249	0.0	70.21
2020-5-4	nearest	stay	0.844	0.203	1.372	0.25	0.0	70.68
2020-5-4	random	stay	0.848	0.199	1.342	0.249	0.23	71.35
2020-5-4	random	ml_poorest_weighted	0.85	0.202	1.323	0.248	0.26	71.33