OPTIMIZATION OF AMUSEMENT PARK QUEUING SYSTEMS

BENNETT COHEN
CHRISTOPHER LANDGREBE
ZEYNEP YELTEKIN
BAŞAK TUZCU

Simulation for Enterprise Scale Systems
Department of Industrial Engineering & Operations Research
University of California, Berkeley

Abstract

The main problem facing both amusement park customers and owners such as Disney World is customer satisfaction and efficiency, which are both negatively effected by high wait times. As a result, these parks have spent significant time and money to implement methods which reduce wait times to both increase customer satisfaction and efficiency. We explored the implementation of an reservation-dependent priority queuing system to devise how to best reduce average customer wait times for Expedition Everest, a popular ride at Disney World. Using third-party data, we first built constant-rate and time-dependent-rate queuing systems to model current behavior, followed by implementing an Express Queue into the system. We found a decrease in average wait time of 18.31% through simulating 30 days of typical customer behavior with the improvement strategy. Finally, we performed sensitivity analysis to optimize the parameters of our improvements, finding an ultimate optimal decrease in wait times of 44.42%.

Contents

Introduction

When visitors make the trip to Disney World, they spend a great deal of time in lines waiting to ride different attractions. It's unlikely that many think about the complex modeling that goes into how ride lines are designed, but queuing systems should be, and are, of great importance to Disney. This is because it is mutually beneficial to the customers, as well as Disney, to mitigate line wait times.

In our project, we focused on a single attraction at Disney World (*Expedition Everest, Animal Kingdom*). We first sought to use third-party data, which is detailed below, to simulate how the ride is currently operating. More specifically, the goal was to use rider throughput data, combined with known ride capacity and ride duration figures, to simulate many days worth of customers in the system.

The first model that we employed, *Model #1: The Naive Model*, was based on modeling customer arrivals as a Poisson Process with constant arrival rate. This model is naive because it simply uses the median hourly customer throughput for the entire day in order to simulate the system. This model provides a good baseline for understanding the system, but it fails to account for the time-dependent nature of arrivals to the line. Customers don't arrive at a constant rate throughout the day, and this is the shortcoming of Model #1 that leads to the more informed approach of our second model.

In our second model, *Model #2: A Time-Dependent Arrival Queuing Model*, we sought to ameliorate the naivete that our first model struggled from. Instead of a constant rate, the data, from both posted wait times and hourly throughput figures, show that after the ride opens at 8:00am, arrivals are slow during the beginning of the day. The arrival rate then increases quickly to a peak at about 12:00pm and slowly decreases, from there, until about 4:00pm, maintaining a relatively high rate of arrival during this period. In the time from about 4:00pm until the ride closes at 8:00pm, the customer arrival rate decreases more quickly, returning to a value similar to the arrival rate at the beginning of the day. In order to model this behavior, we utilize a simplified interpolation method to fit a 4th-order polynomial to the arrival rate data. We then use rejection sampling to simulate a time-dependent Poisson Process. Because the arrival rate is in-homogeneous throughout the day, this method leads to a more accurate and realistic simulation model.

After successfully simulating the current behavior of the ride's line, it's only natural to think of ways that the system could be improved. In our third model, *Model #3: A Queuing System with Express (Priority) Queue*, we introduced the idea of a priority line, or an "Express Queue" as we call it. This model, detailed below, sought to decrease the ride time of the customers by offering the customers the opportunity, upon arrival, to obtain an “Express Pass," which is valid for a specific time period in the future. The incentive to the customer is that upon arrival to the Express Queue, their wait time would be significantly lower because the Express Queue is always loaded onto the ride before the Standard Queue. The implementation of the Express Queue proved to be effective at lowering wait times because it distributes the volume of arrivals more uniformly throughout the day, resulting in lower average wait times for customers.

After implementing an improved model, we then performed sensitivity analysis on the assumptions and parameters to "tune" the model in *Model #4 A Tuned Improved Queuing System* to further decrease both the average wait time and the difference between the Standard and Express queues respective wait times. We first explored the effect of the assumptions of our model on these statistics, followed by an investigation into the impacts of changes in model parameters.

At its core, we explored solutions to solve a mutual issue between the customer and Disney, in hopes to drive customer satisfaction and efficiency. In the report that follows, we delve into the inner-workings of our process to design both the baseline and improved models in hopes to achieve a realistic system with significantly lower average wait times.

1.0.1 A Technical Note

For all simulation models outlined in this report, we used iPython Notebooks to simulate the systems. We used object-oriented programming to represent the customers and simulated queues. We then wrote functions that methodically simulated each customer arriving to the system at times defined by the assumptions made in each model. After arriving, customers were processed by the functions in a first-in first-out fashion. The functions recorded their wait times, then served them in amounts of time consistent with the service parameters defined below. In each of these functions, the system continued operating until each and every arrival had either been processed or abandoned the line. For more specifics on the code used to simulate the models, refer to the appendices at the end of the report.

Data Collection & Analysis

2.1 Data Collection

In order to build an accurate simulation model for a given ride, we need two key pieces of data, upon which the entire model depends: the inter-arrival time and service time. Historically, Disney World has not released arrival or service data, likely due to the implications on Disney stock prices. As a result, we were presented with two options: (1) Collect the data ourselves or (2) Find a 3rd party who already had collected the required data. Due to COVID-19, our travel is limited, so we contacted Len Testa, an accomplished computer scientist, co-author of *The Unofficial Guide to Walt Disney World*, and president of TouringPlans.com, a research team dedicated to analyzing theme park data and helping customers get the most value of their vacations.

Fortunately, through TouringPlans.com, we were able to access pre-existing data on the posted wait times at dozens of rides throughout the day from 2015 to 2019. As we demonstrate in greater detail below, there is a certain seasonality to amusement parks (mostly due to weather and common vacation times) such that we only analyzed the month that our other data is collected from, which is November.

	Date	Time	Posted Wait
0	2015-11-01	08:03:00	5.0
1	2015-11-01	08:10:00	5.0
2	2015-11-01	08:17:00	5.0
3	2015-11-01	08:24:00	5.0
4	2015-11-01	08:30:00	5.0
5	2015-11-01	08:30:00	5.0
6	2015-11-01	08:38:00	5.0
7	2015-11-01	08:45:00	5.0
8	2015-11-01	08:52:00	5.0
9	2015-11-01	08:59:00	5.0

Figure 2.1: First 10 entries of posted wait time data in November from 2015 to 2019 (14,518 total data points).

Further, using their own data collection and modeling methods, Mr. Testa's team was able to calculate approximate arrival rates for a few different intervals throughout the day: opening hour (8:00am to 9:00am), peak hour (11:00am to 12:00pm) and late afternoon (4:00pm to 5:00pm), with the park closing at 8:00pm sharp. The figure below shows the first 5 rows of a table with all 30 days of November 2019, along with calculated arrival rates.

	Date	8am - 9am	11am - 12pm	4pm - 5pm
0	2019-11-01	1517	3330	3090
1	2019-11-02	1540	2820	3870
2	2019-11-03	1517	3210	3090
3	2019-11-04	1657	3000	2820
4	2019-11-05	1517	3180	2640

Figure 2.2: First 5 entries of arrival rate data for Expedition Everest from November 2019, before the COVID-19 pandemic.

We were able to access service data approximations for Expedition Everest as well. While the ride itself is on a track run by computers with negligible variability for sake of modeling, the time to load and unload passengers onto the ride has definite stochastic traits. We'll explore these below.

Service Phase		Minutes
Loading Time	Minimum	0.5
	Maximum	1.0
Ride Time	Average	3.0
Unloading Time	Minimum	0.25
	Maximum	0.75

Figure 2.3: Service time data for the loading, unloading, and ride time phases of Expedition Everest.

Finally, though the capacity of the ride is generally dependent on crowds and defects such as broken seats, safety equipment, etc, it appears that Disney will add to their capacity as crowds increase throughout the day. The full capacity of the ride is 6 trains of 34 people each, however in order to be able to add a train (and sometimes two) in the middle of the day when the crowds are higher, the initial capacity must be 4 trains of 34, or 136 people total. For modeling sake, all of our models will fill trains to capacity whenever possible, unless there aren't enough people in the queue. We assume the average times of adding the first and second extra train to the ride as in the capacity table below.

Time	Capacity
8:00am to 11:00am	136 customers
11:00am to 1:00pm	170 customers
1:00pm to 8:00pm	204 customers

Figure 2.4: Capacity data for Expedition Everest for each interval throughout the day.

2.2 Exploratory Data Analysis

In order to build simulations that accurately model a queuing system for Expedition Everest, it's necessary to understand the input data. First, we analyze our arrival rate data set. The data collected doesn't give a specific timestamp for arrivals, but instead consists of the number of arrivals across the entire hour for every day in November, 2019. To visualize this clearly, for each point in the data set, we plot a point with an x-coordinate equal to a random uniform number between that hour's start and end (e.g. a random number between 8:00am and 9:00am), and a y-coordinate of it's arrival count.

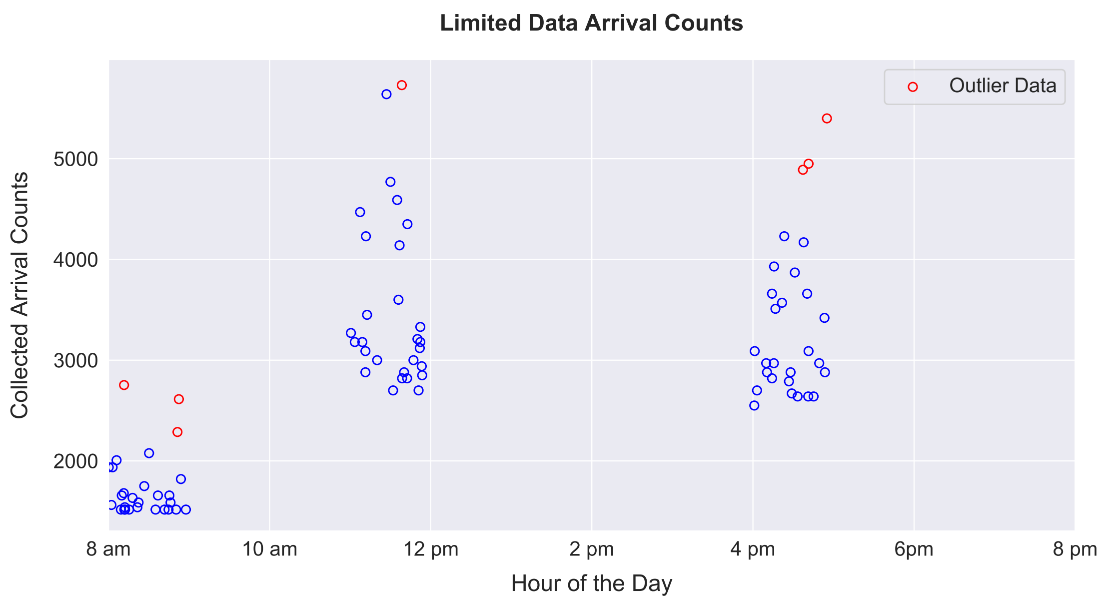


Figure 2.5: A scatter plot of the limited arrival count data set.

From this plot, two key trends are clear. First, there is very high variability in the arrival counts for a given hour, especially towards the middle of the day. With this trend—along with the sheer lack of data—in mind, we go through a modified-standard process of removing outlier data points using the Interquartile Range (IQR). In this method, we set a lower-bound and upper-bound of acceptable values as [Q_1 - 1.25*IQR, Q_3 + 1.25*IQR], where Q_1 is the lower quartile, and Q_3 is the upper quartile of the data. The standard method for removing outliers is setting the bounds of acceptable points at [Q_1 - 1.5*IQR, Q_3 + 1.5*IQR] (approximately $\pm 3\sigma$ when the distribution follows a Gaussian Distribution), however we see that the distributions of the arrivals don't follow a Gaussian Distribution well (they are heavily right skewed towards higher values), so it's acceptable to be more liberal in removing outliers.Using this process, we remove all missing and extreme data. The red points on the plot represent these outlier points which won't be used in future analysis and modeling. It's clear that our method partially removes the more extreme right-skewness of the data, as only upper end points were removed, while all lower end points were close enough to their respective 1st quantiles to be accepted.

One thing we must consider when doing our analysis and simulation is the seasonality of amusement parks. In the figure below, the monthly average wait time is displayed, which is directly positively correlated with higher attendance (as will be discussed). We see that November has an average wait time slightly higher than the average across the entire year, so that our findings in our models will likely generalize well to most times of year excluding peaks (such as Christmas time) or valleys (just after summer). This is quite important as our arrival rate data is only from November.

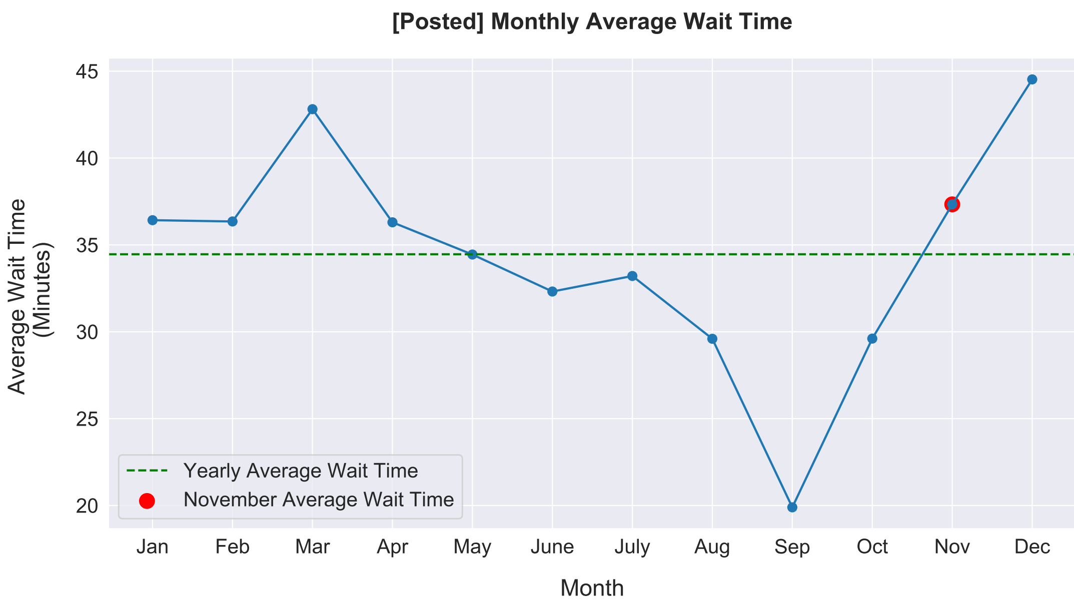


Figure 2.6: A plot of the posted monthly average wait times over the last five years.

The figure below plots the average posted wait time in November versus the time of day. It's clear from this that the wait times are not constant throughout the day, but increase very quickly in the morning, peaking around noon, and slowly decreasing as the day ends.

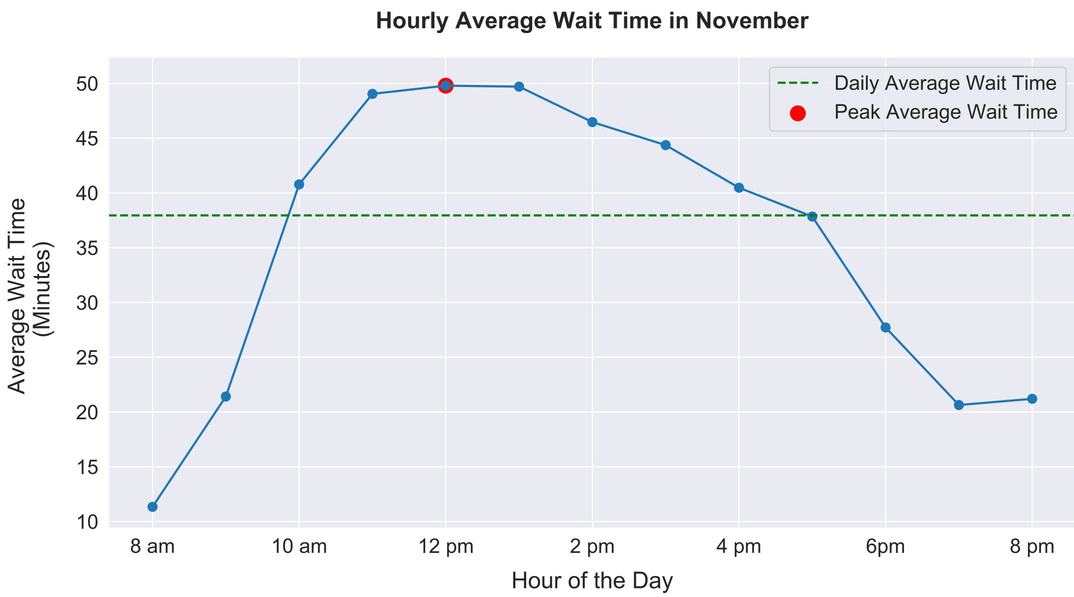


Figure 2.7: A plot of the posted average wait times versus hour of the day for November.

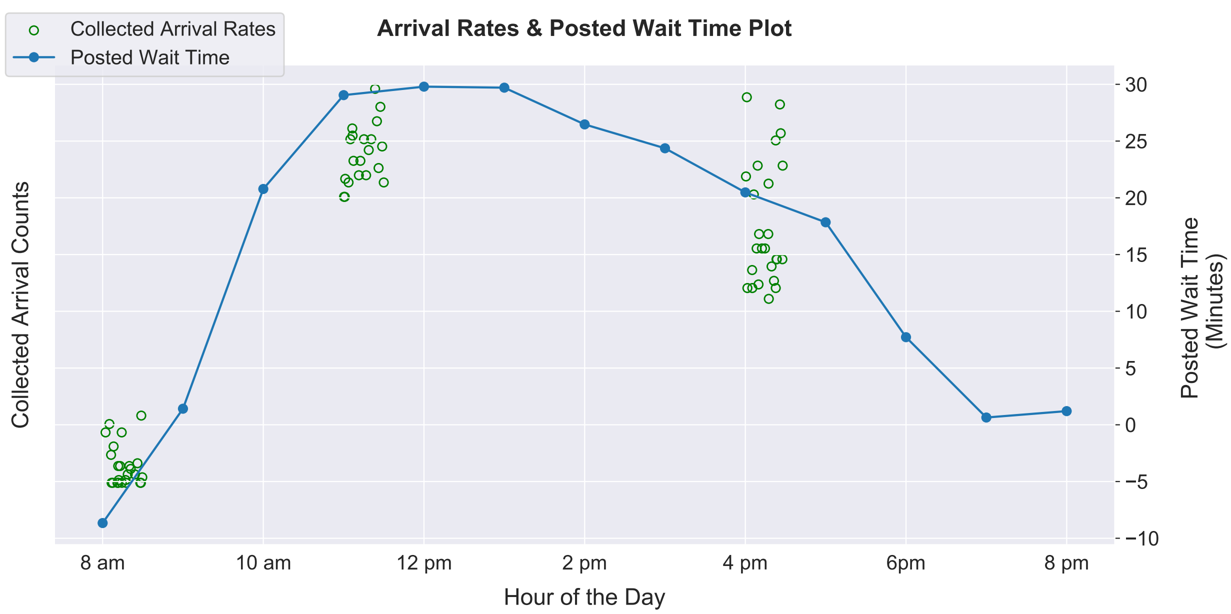


Figure 2.8: A superimposed plot of posted wait time data onto a scatter plot of arrival count data.

When we superimpose this line plot onto the arrival count data scatter plot Figure 2.5 above, we see that there is a clear correlation between arrival rates and wait times. It is impossible to calculate the extent to which these features are correlated as we don't have paired data points (i.e. a data point that has a time of day, along with arrival and wait times for that specific time), however this notion of correlation between arrivals and waits will drive our Model #2, the Informed Model. Now that the data sets have been analyzed and well comprehended, we move to build baseline models of the queuing system.

Base Models

Just as we need to understand the trends of our data sets before building a model, it's necessary to model the behavior of the current queue using the collected data so that our improved models have a baseline to which they can be compared. First, we create a naive queuing model, built upon a constant arrival and service rate. The second model maintains the constant service rate, but defines the time-inhomogeneous arrival rate to better reflect human traffic through the system at different times of the day, as described in the Exploratory Data Analysis section.

3.1 Model #1: The Naive Model

Our first, naive model is built upon the notion that the arrival rate of customers through the line of Expedition Everest is roughly constant throughout the day. We saw in the Exploratory Data Analysis section that the arrival rates to the system are not constant throughout the day, but instead has some time-dependent nature. However, for simplicity, we build a model using a constant arrival rate. We also note here the assumption that the service rates are affected by factors that don't apply to the baseline models, and thus will be kept as constant parameter distributions throughout the day. As such, we begin by defining the following distributions and corresponding parameters for the naive queuing model using our collected data.

The average hourly arrival rate is modeled by a Poisson Distribution with exponentially distributed inter-arrival times. From the arrival rate data analysis in the Exploratory Data Analysis section, we see that the distribution of collected rates is skewed right, indicating that higher-valued outliers would increase the mean arrival rate. Instead, we define the arrival rate as the median of all arrival rate data, which won't be impacted by outliers as much:

$$A \sim Poisson(\lambda_{med}), \quad \lambda_{med} = 2,880 \quad \frac{arrivals}{hour}$$

Given the service data provided in Figure 2.3, we are limited in the types of distributions we can use to model these processes. For instance, while an Exponential Distribution might be suitable for new or highly variable processes, and a Gaussian works well when the most likely values and standard deviation are known, these might not be the most suitable for our processes. Disney is known for their commitment to operating efficiency, and likely has done everything in their control (within reason) to make the loading and unloading processes as consistent as possible. Further, we don't know the average time or standard deviation of these distributions. Under this criteria, a Uniform Distribution is selected so that each time within the minimum and maximum loading and unloading values is equally likely to occur.

As a result, a Uniform distribution seems to apply well for the Loading and Unloading phases of the service times, while keeping the ride time a constant. For ease of computation, we define the service distribution as the sum of a constant and two Uniform Distributions, which is a shifted triangular distribution, as seen in the plot below. However, as we will explore further in later models, we will keep the service phases separate for now.

$$S \sim U(\alpha_{load}, \beta_{load}) + T_{ride} + U(\alpha_{unload}, \beta_{unload}))$$

$$S \sim U(0.5, 1) + 3 + U(0.25, 0.75)$$

The capacity of the ride refers to how many riders can be on the ride at a given time. From some historical data and expertise, we know that Expedition Everest does not always operate at full capacity. For this system, we model the capacity behavior as described in similar manner from Figure 2.4.

$$C_t = \begin{cases} 136 & \text{if } 0 \leq t \leq 3 \\ 170 & \text{if } 3 < t \leq 5 \\ 204 & \text{if } 5 < t \leq 8 \end{cases}$$

Finally, we address those who enter the line, but decide to leave at some point in the waiting process due to various factors, including the posted wait time seems too high, the actual line seems too long, or customers making other plans to ride other attractions. To accomplish this, we use the wait time of the most recently processed customer (the person in from of them in line) in order to approximate the wait time of a given customer. It's necessary to use this approximation, because customers are processed in a first-in first-out fashion, so the most recent wait time is the best predictor of successive wait time. In our model, we use this assumption such that the abandonment probability for each customer entering the line is dependent on the true wait time of the customer before them (which our model computes before determining abandonment). The table below defines the wait time-dependent abandonment probabilities for a given customer, $W \sim Bernoulli(p = p_i)$.

Previous Wait Time, T_{i-1} (Minutes)	Abandonment Probability, p_i
$0 \le T_{i-1} < 40$	0.005
$40 \le T_{i-1} < 50$	0.015
$50 \le T_{i-1} < 60$	0.03
$60 \le T_{i-1} < 75$	0.08
$75 \le T_{i-1} < 90$	0.1
$90 \le T_{i-1} < 180$	0.1
$T_{i-1} \ge 180$	0.15

Figure 3.1: A table of abandonment probabilities, dependent on the previous customer's wait time.

Now that the parameters of the naive model have been calculated from the data, we build the naive queuing system using Python. Figure 3.2 describes how customers flow through the system.

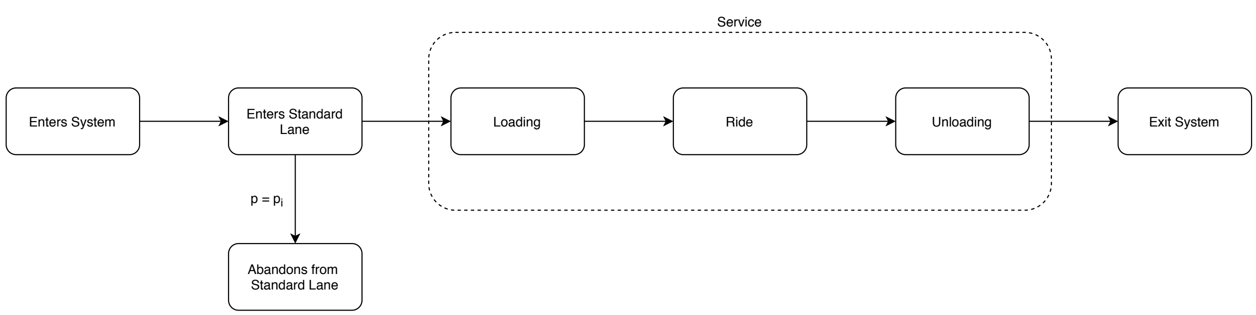


Figure 3.2: A flow diagram for customers in Model #1 using constant arrival and service parameters.

Finally, we run Model #1 and simulate 30 days of Expedition Everest in November. From the 991,590 individual customers serviced in this time, we obtain the following table of relevant outputs.

Statistic		Value
Wait Time	Minimum	0.00 mins
	Average	53.638 mins
	Maximum	78.562 mins
Daily Throughput	Minimum	32,739 customers
	Average	33,063 customers
	Maximum	33,370 customers
Abandonment Percentage	Minimum	3.59 %
	Average	4.44 %
	Maximum	5.29 %

Figure 3.3: A table of relevant outputs for the Naive Model.

From Figure 3.3, we see that the average time a customer spends waiting in the system is 53.652 minutes, with an average daily throughput of 33,053 customers. One thing to note is that while this value is larger than the posted wait time average (approximately 37 minutes), this isn't alarming for various reasons. Firstly, we assume a constant arrival rate, which clearly isn't the case, because with the true time-dependent arrival rate, it's possible that the system spends more time at a lower arrival rate than the median, which would "pull" the wait times down, assuming the service and capacity parameters were the same. Furthermore, we know that Disney implements some priority queue system, so their average wait time with this would be quicker than a single queue/single server system. Lastly, the posted wait times are merely approximations of the true wait time, and it's possible Disney is (1) incorrect in their predictions or (2) scales their predictions by some factor in order to affect abandonment rates, and by proxy, the length of the queue.

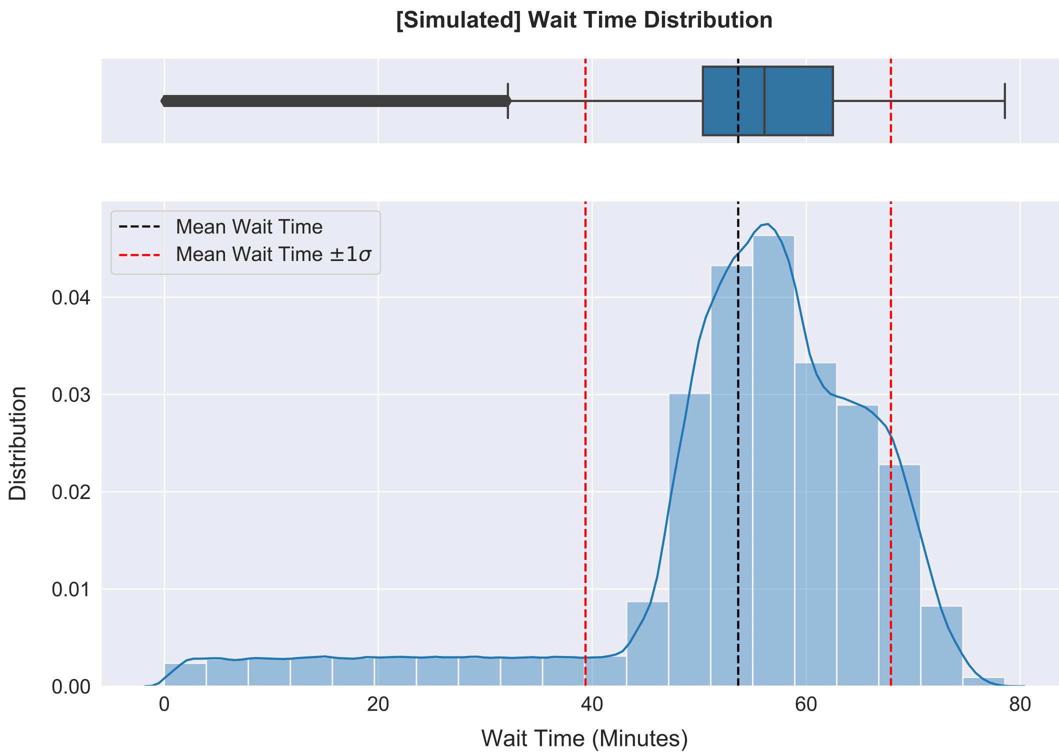


Figure 3.4: A histogram of all 991,590 simulated wait times over the 30 days of simulation.

The figure above is a histogram of all simulated wait times for the 30 days of simulation, along with axis lines for the mean wait time and mean $\pm 1\sigma$ wait times. The distribution is clearly very left skewed, with roughly 67 % of wait times below one hour, whereas the posted wait time data consists of 80 % of wait times below one hour. However, one interesting thing to note is that the standard deviation of the posted wait times is $\approx$ 25 minutes, whereas it is $\approx$ 14 minutes in our model. From a practical standpoint, this again is probably beneficial because resulting customer satisfaction and throughput would have more predictability (less variability).

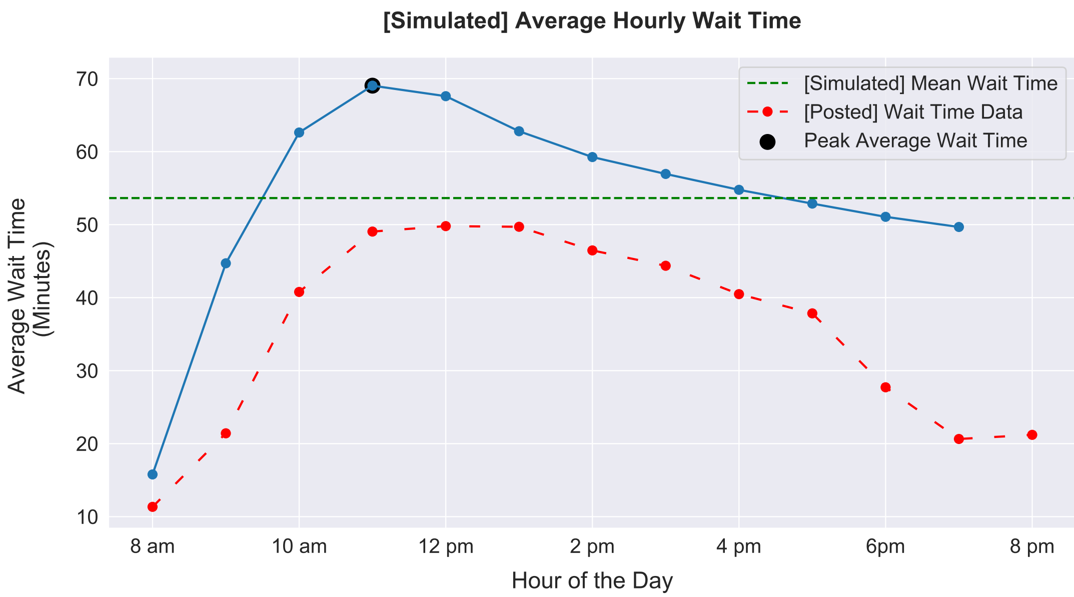


Figure 3.5: A line plot of the simulated wait times and posted wait time data versus the time of day.

The line plot above juxtaposes the posted wait time data from the earlier analysis with the simulated wait times versus the time of day. The key trends—and preceding causes of these trends—are likely the result of the capacity shifts throughout the day. From 8:00am to 11:00am, the capacity of the ride is 136 customers (4 trains of 34). Because we are modeling with the median arrival rate (which is likely higher than the true early morning arrival rates), the system can't serve customers quickly enough, resulting in the queue "backing up," and wait times increasing very quickly. At 11:00am, we add a fifth train, increasing the capacity to 170. This corresponds to the peak wait time on the plot, because after 11, even though the line backs up, the increased capacity slowly pulls the average wait time down.

3.2 Model #2: A Time-Dependent Arrival Queuing Model

To build upon the calculations from the naive model, we begin to explore the fact that arrival rates are not constant throughout the day. Again, though we don't have the data from Disney, we can reasonably assume that the arrival rate and average wait times are directly correlated and follow a similar distribution (Figure 2.8); as there are more (or fewer) arrivals in a given time period, the average wait time will increase (or decrease) as well, delayed by a small interval.

Using this notion, the goal is to define the arrival rate as a function of time of day, such that we can implement a time-inhomogeneous (or time-dependent) arrival parameter. One common solution to this problem is to introduce a step function, where for an entire hour the rate is constant, then instantaneously increases or decreases to a predetermined rate for the next hour. While this makes the calculations and modeling much easier, this solution is not feasible outside of simulation. The process of arrivals is continuous and cannot instantaneously change as it would in a step function. Therefore, we must create a continuous function. Another solution could be to define a piecewise rate function as a set of continuous functions. This remedies the instantaneous change issue of a step function, but the arrival data available only consists of arrival counts for three different hours (8:00am - 9:00am, 11:00am - 12:00pm, and 4:00pm - 5:00pm) throughout November. Without more data, it would be inaccurate modeling to arbitrarily pick arrival rates for the nine other data points needed. This would also be prone to overfitting to the posted wait time distribution, a concept that will be discussed below. The flow of customers through the system is effectively the same as the naive model, with the only change being different arrival rates.

We first use Python's Sci-Kit Learn library to perform polynomial regression on the posted wait time data. While ordinary linear regression is generally preferable to polynomial regression due to ease of implementation and simplicity, because there is only a single feature variable (the time of day), there is only one non-intercept term. This means the resulting output could only be of the form of a two-dimensional line. In this case, the line created by minimizing the L2-loss would be nearly horizontal with a constant value of ≈ 37 minutes, which is the average posted wait time throughout November for the 14,518 data points we have available. Our next consideration is what order function fits our data best without being prone to overfitting.

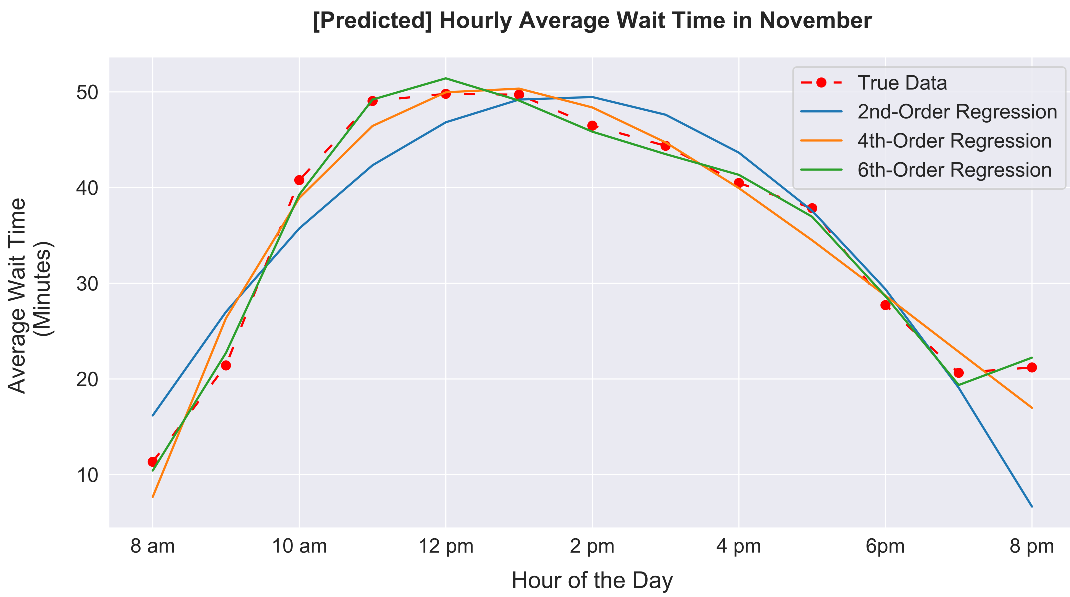


Figure 3.6: Plots of various polynomial regression models for predicted posted wait time from hour of the day.

Due to our lack of data on the actual arrival rates, we can't truly perform meaningful regression on it. This is because we don't have the exact time of day that the arrival rate was calculated (only the hour) and we only have these "general" data points from three different hours so it would be insufficient to predict the other hours from our limited data. However, our strategy is to find the minimum order function that fits the posted wait time data with marginal error benefits (as defined as the decrease in error by increasing the degree of our polynomial regression by 1). We then use our limited data and the assumption that the arrival rate follows a similar distribution to the posted wait time data from the Exploratory Data Analysis section to generate the arrival rate as a function of time. Expanding on Figure 3.6, we use K-Fold Cross Validation on polynomial regression with degrees from 1 to 10, and calculate the corresponding Root Mean Square Error (RMSE) and Standard Deviation.

Regression Type	Train RMSE	Cross-Val RMSE	Cross-Val Std	Test RMSE
Degree-1 Polynomial	25.007	25.634	11.970	25.219
Degree-2 Polynomial	22.505	24.389	10.712	23.706
Degree-3 Polynomial	22.232	23.121	10.854	22.456
Degree-4 Polynomial	22.227	23.127	10.845	22.448
Degree-5 Polynomial	22.225	23.130	10.838	22.455
Degree-6 Polynomial	22.112	23.045	10.817	22.371
Degree-7 Polynomial	22.102	23.045	10.818	22.367
Degree-8 Polynomial	22.100	23.045	10.809	22.360
Degree-9 Polynomial	22.101	23.047	10.812	22.363
Degree-10 Polynomial	22.104	23.050	10.813	22.368

Figure 3.7: Root Mean Square Error and standard deviation for varying-degree polynomial regression models with 10-Fold Cross-Validation.

From this, we see that the marginal error benefits are negligible, if at all, beyond a 3rd-order polynomial regression model. However, looking at the tail-end behavior of the graph, we see that wait times decrease at the beginning and end of the day. In a purely mathematical sense, $\lim_{x\to-\infty} = -\infty$ and $\lim_{x\to+\infty} = -\infty$. We need an even-ordered function achieve this behavior, so we decide to generate a 4th-order polynomial. To create a polynomial of degree-n, a minimum of n+1 points of the function are needed; as such, we need 5 points to do this for a quartic function. We are essentially given 3 arrival data points, which are the median arrival counts for the three hours from collected data. The remaining two points are easily extrapolated from the posted wait time data given. These points encapsulate a majority of the shape and variation of the plot, giving us the ability to create a 4th-order polynomial which models the arrival rate as a function of time with reasonable accuracy. Further, we make note of the fact that higher order functions do perform better on the training data prior to cross-validation, as seen in the 6th order plot nearly fitting our training data perfectly in Figure 3.6. However, this causes issues in regards to over fitting, in that we don't know the true distribution of the arrivals to the queue. From analysis and 3rd-party expertise, we have made the assumption that arrivals and wait time have a high correlation coefficient—which allows us to implement this interpolation method—but we don't know the extent of this. We would only want to fit the data perfectly if the correlation coefficient between arrivals and wait time were extremely high (≥ 0.9), but because this value is unknown, the most prudent method is to model the general trends and shape of the wait time data, rather than overfit to the noise.

By parameterizing our time of day as hours since the ride opened, such that 8:00am = 0, we have the three points $(0.5, 1587)$, $(3.5, 3180)$, and $(8.5, 2970)$ for the hours of 8:00am, 11:00am, and 4:00pm, respectively. Again, because we are only given counts across an entire hour, to interpolate it simply, the X coordinate of the three points is the midpoint between their respective hours. We then make the following extrapolations:

1) From our posted wait time data, we see the true peak is actually closer to the 12:00pm hour for Expedition Everest specifically, and that the 1:00pm wait time ≈ 11:00am wait time. Thus, we conclude $\lambda_{1pm} \approx \lambda_{11am} = 3180$, and our 4th point of $(5.5, 3180)$.

2) The average wait time at 7:00pm is roughly 58% of the total average wait time throughout November, however our wait time appears to increase as the park approaches closure time, which would appear to change the tail-end behavior, but upon deeper inspection, this might not be the case. Firstly, the wait time data we have are posted wait times—not the actual time a customer waited. Due to strict operating hours, it's very likely the posted times are inflated at the very end of the day in order to deter riders from joining the queue so late because they might not be able to ride before the 8pm closure. For instance, if a customer joins the queue at 7:45pm and the wait time is posted at 15 minutes, they might still stay in hopes of getting on the last available ride. However, if Disney inflates this 15 minutes to 25 minutes, the customers will see that they likely won't be able to ride it. This increases shutdown efficiency, and likely customer satisfaction as well, because customers won't be spending "wasted" time in line. Secondly, we don't have as many data points towards the end of the day compared with the peak hours, meaning the data itself might not be that reliable, and could be subject to greater noise and outliers. Following the general wait time patterns, it's reasonable to make the assumption that towards the end of the day, the arrival rate tends towards the beginning of the day rate as business slows down slightly. Therefore, We conclude $\lambda_{7pm} \approx \lambda_{8am} = 1587$, and our 5th point of $(11.5, 1587)$.

Before applying this strategy to the wait time data, we use the same simplified interpolation method of a 4th-order polynomial to the posted wait time data to confirm it captures most of the data trends. Below is the outputted plot of the true data, a proper 4th order regression model, and our simplified method. It can be seen that the simplified version does not have quite the goodness of fit as the 4th order polynomial regression, but with regard to our lack of available data, it serves the purpose of providing an accurate model without baselessly overfitting. It's clear that with further data collection, this function could be improved.

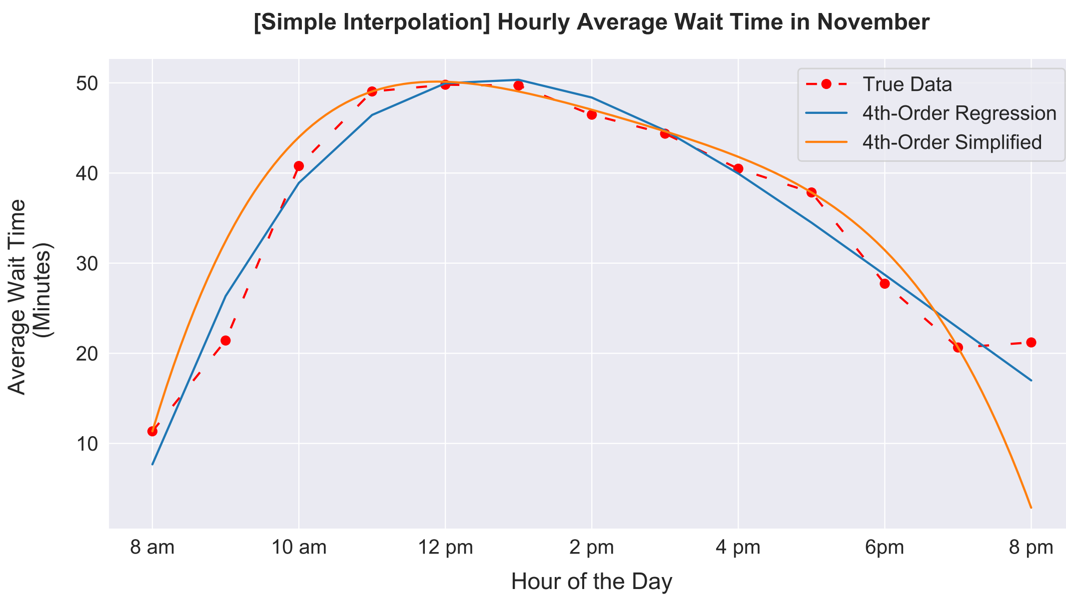


Figure 3.8: A simplified 4th-order interpolation of wait time data compared with a 4th-order polynomial regression

Finally, we interpolate our 4th-order function by creating a system of equations of the general form of a quartic function $f(x) = ax^4 + bx^3 + cx^2 + dx + e$.

$$E_1: \quad 1587 = a(0.5)^4 + b(0.5)^3 + c(0.5)^2 + d(0.5) + e$$

$$E_2: \quad 3180 = a(3.5)^4 + b(3.5)^3 + c(3.5)^2 + d(3.5) + e$$

$$E_3: \quad 3180) = a(5.5)^4 + b(5.5)^3 + c(5.5)^2 + d(5.5) + e$$

$$E_4: \quad 2970 = a(8.5)^4 + b(8.5)^3 + c(8.5)^2 + d(8.5) + e$$

$$E_5: \quad 1587 = a(11.5)^4 + b(11)^3 + c(11)^2 + d(11) + e$$

With four equations and four coefficient variables to solve for, the system has one unique solution. We obtain the rate function with variable t for the time after opening:

$$f(t) = -1.629t^4 + 40.849t^3 - 385.935t^2 + 1574.087t + 891.435, \quad t \in [0, 12]$$

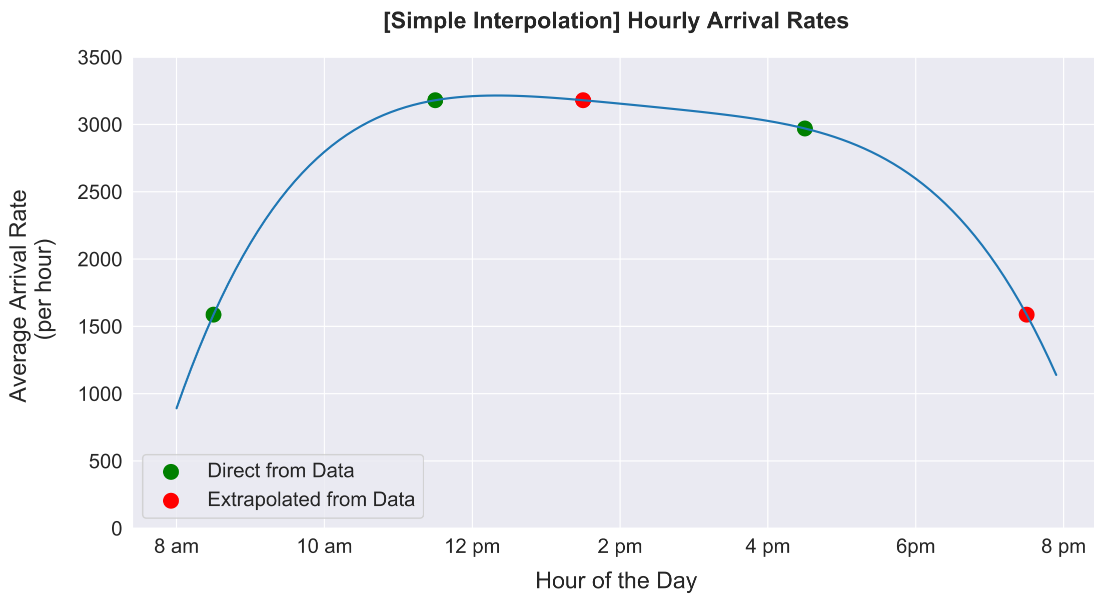


Figure 3.9: An interpolated arrival rate function plot using correlated wait time data.

The service time and capacity parameters are the same as defined in the Naive Model. The arrival distribution is defined below:

$$A \sim Poisson(\lambda_t)$$

In order to use this time-dependent arrival rate, we use what is called a Rejection Sampling Algorithm (or Thinning Method). We first simulate n arrivals $\tilde{A}_1, \tilde{A}_2, ..., \tilde{A}_n$ on the interval [0,12] from a Poisson distribution with a constant rate $\tilde{\lambda} = max_{t \geq 0} f(t)$. Using Python's Sci-Py library, we find this maximum arrival rate to occur just before 12:00pm, with coordinates (3.9, 1810). For each arrival $\tilde{A}_j$, we accept this arrival as a "true" arrival binomially with probability $= \frac{f(j)}{max_{t \geq 0} f(t)}$. The rejected arrivals are dropped from the arrival list, leaving us with a list of true arrivals falling under the density curve provided by our rate function. Lastly, we run Model #2 and simulate 30 days of Expedition Everest in November. From the 930,246 individual customers serviced in this time, we obtain the following table of relevant outputs.

Statistic		Value
Wait Time	Minimum	0.000 mins
	Average	48.839 mins
	Maximum	72.550 mins
Daily Throughput	Minimum	30,776 customers
	Average	31,008 customers
	Maximum	31,252 customers
Abandonment Percentage	Minimum	3.84 %
	Average	4.70 %
	Maximum	5.40 %

Figure 3.10: A table of relevant outputs for the Informed Model.

From Figure 3.10, we see that the average time a customer spends waiting in the system is 48.839 minutes, with an average daily throughput of 31,008 customers. In comparison to the naive model, these numbers are comparably lower, however this comes at no surprise.

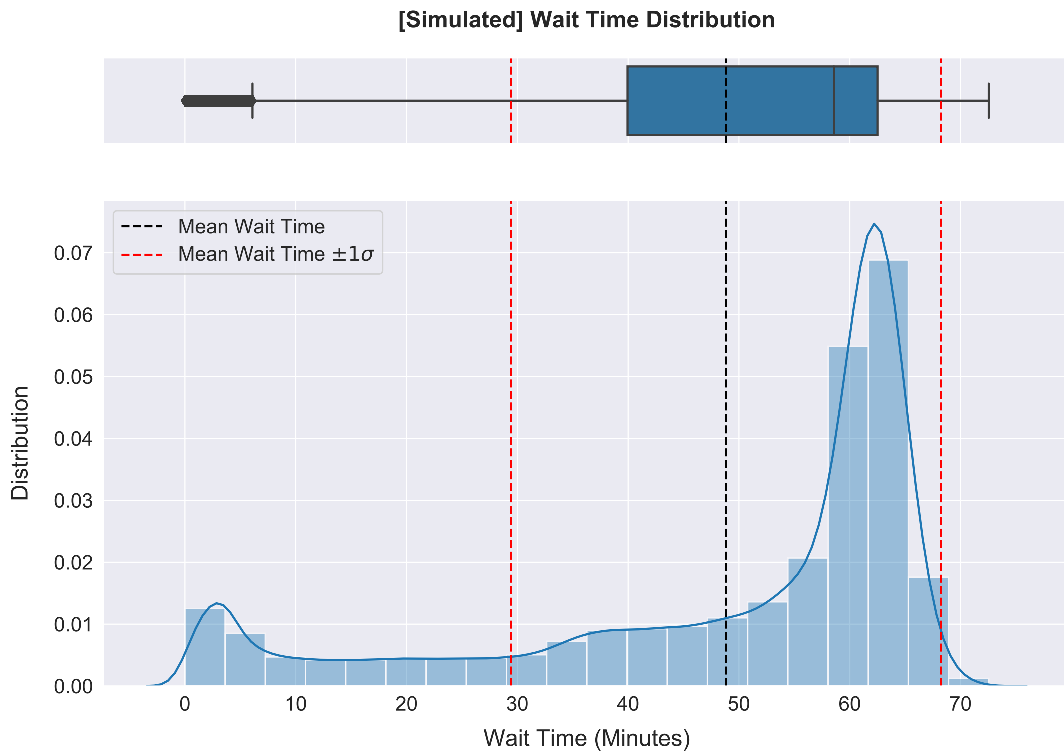


Figure 3.11: A histogram and box plot of all 930,246 simulated wait times over the 30 days of simulation.

Figure 3.11 contains a histogram and box plot for the simulated wait times. Though the shape is quite similar to Model #1, we see it is shifted left to reflect the decrease in average wait time. The left skewness is even more pronounced than in the naive model, with the new median being significantly larger than the mean now. The line plot in Figure 3.12 likely depicts the the true wait times versus time of day more accurately the the naive model. The peak in wait times is later than in the naive model, as the system takes time to "back up" before the highest wait times are realized. Further, the second half of the day has a concave shape, compared to the convex shape of Model #1. Similar to the beginning of the day, we see that the end of day wait times are better modeled with the more accurate arrival rate parameters. Now that we have built upon the naive system to simulate a current simple queuing system with a decent degree of accuracy, we can begin to devise improvements.

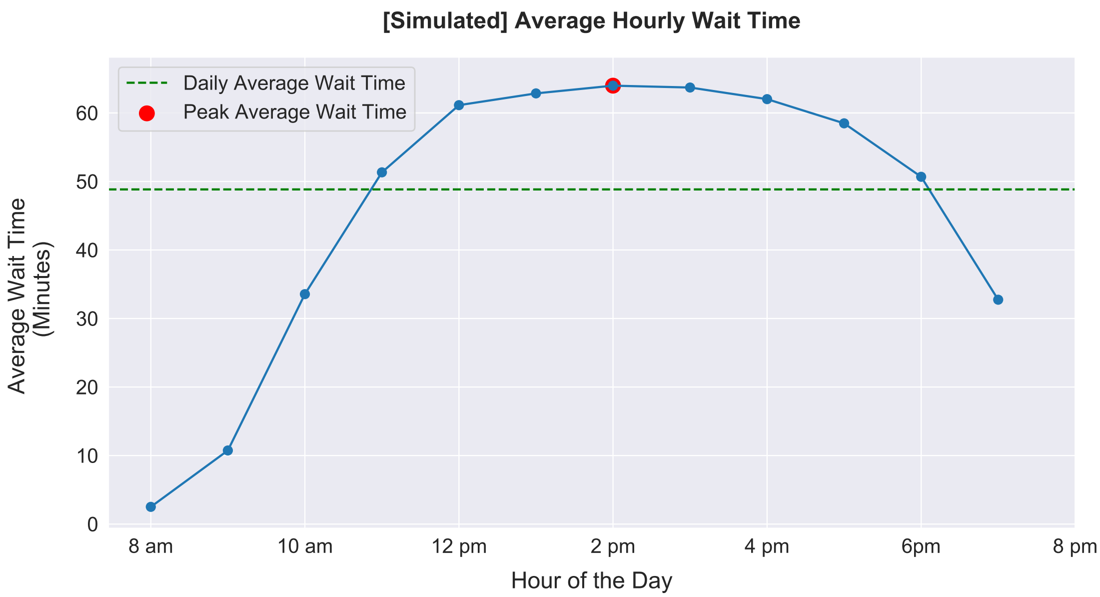


Figure 3.12: A line plot of the simulated wait times and posted wait time data versus the time of day.

Improved Models

First, we created a naive model with constant arrival and service parameters to give a baseline structure for our system. Then, using polynomial regression and a simplified interpolation method, we generated a continuous 4th-order function which models the arrival rate as a function of time. This function was used to more accurately model a queue at Expedition Everest via Rejection Sampling. Now, we implement an Express Queue (priority line) in addition to the Standard Queue to see the impact on wait time and throughput, in an attempt to minimize customer wait time. In order to enter the Express Queue, a customer must receive an Express Pass by making a reservation via a smartphone app. The following two improvement sections will demonstrate the implementation of the new queue, along with different methods of tuning the system to optimize average wait time and throughput.

4.1 Model #3: A Queuing System with Express (Priority) Queue

With this model, we diverge from the baseline queuing system, and attempt to devise a new system that will drive average wait times down in order to increase total daily throughput. To do so, we use a three-pronged strategy:

First, a Reservation-Driven Express Queue. When a customer arrives to the system, they are now presented with three options, whereas the baselines models had only two options. A customer now can (1) Abandon the System, (2) Join the Standard Queue, or (3) Obtain an Express Pass. The decision for each individual customer to abandon the system is defined the same way as in the baseline models, which is via a Bernoulli Distribution, $W \sim Bernoulli(p = p_i)$ in Figure 3.1. In order to most clearly analyze the impacts of introducing an Express Queue into the system, we set the proportion of customers who obtain an Express Pass statically as, $p_{express} = 0.15$ (we note here that this number is arbitrary and has no significant statistical basis. It will serve as a starting point from which we can tune the model in the future). The remaining customers (i.e. those who did not abandon the system or obtain an Express Pass) enter the Standard Queue. In order to obtain an Express Pass, a customer must be within a geo-fenced area of the ride (such that they need to have arrived at the system, but made the decision to get a pass instead of joining the Standard Queue or abandoning) and download a QR code via smartphone app run by the park. Further, before downloading, Express Riders will watch the standard safety video presentation that would be presented to Standard Riders during the loading phase of service. Obtaining an Express Pass via the smartphone app is the "reservation," which is required to enter the Express Queue. Upon receiving an Express Pass, the rider is permitted to enter the Express Queue, however it must be within 30 to 90 minutes of receiving the pass. Attempting to scan the Express Pass to enter the Express Queue will not work before 30 minutes, and the pass will expire after 90 minutes. If a customer does not enter the Express Queue within this time period, they will have to obtain another one. In this model, we again set a static probability of obtaining an Express Pass but missing the 30-90 minute interval (and effectively, an unused Express Pass) as $p_{unused} = 0.15$. There is no limit in this model to the number of Express Passes that the ride can give out, but a customer can only have one Express Pass at a time, and each Express Pass works for a single customer only. Due to the limitations on when a customer can come to the Express Queue, the system stops giving out Express Passes 10.5 hours into the day, at 6:30pm. Any customer who arrives after this point will either abandon the system or join the Standard Queue. A customer's path through the devised queuing system is shown below:

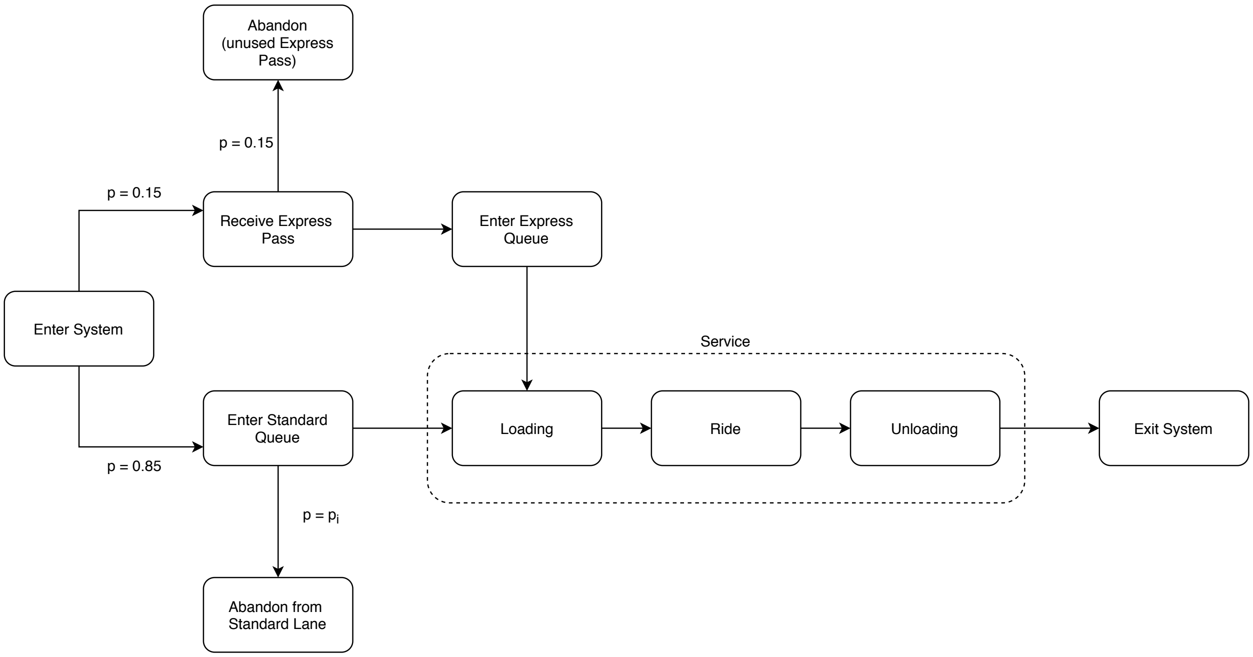


Figure 4.1: A flow diagram for the improved model.

Second, a Static Loading Strategy. With two queues leading to the same end point, we design the following strategy for determining who is serviced when: At the front of both the Standard and Express Queues, there is a pair turnstiles that operate under a Bernoulli Distribution with an opening probability, $X_{Priority} \sim Bernouli(p_{open})$, This means for each pair of customers at the front of their respective queues, the the Express turnstile opens with probability p_{open}, while the Standard turnstile opens with probability 1 - p_{open}. For this simple case, we set $p_{open} = 1$, so that Express Riders are always loaded first. If the Express Queue is empty (i.e. all Express Riders have been serviced), the Standard turnstile will open. As with the static probabilities defined above, this number serves as a starting point from which we can perform sensitivity analysis on in the future.

Third, a Service Speed Factor. Under the current system in both the Naive Model #1 and Informed Model #2, the service time distribution is the sum of the constant ride time and two uniform distributions, for the loading and unloading times, respectively. This was defined earlier as $S \sim U(0.5, 1) + 3 + U(.25, .75)$. However, in our newly designed system, we do two things such that it is actually quicker to load Express Riders than Standard Riders. As explained earlier, to obtain an Express Pass, customers must watch a safety video presentation. Standard Riders, on the other hand, watch the safety video presentation as part of their Loading Phase, but prior to getting on the train. Secondly, to account for this video presentation, we place the Express Queue physically closer to the actual train than the Standard Queue. Now, because all riders on a given train will have the same finish time, we cannot simply give Express Riders lower Loading Phase parameters. In other words, even if we load Express Riders faster than Standard Riders, the Express Riders would have to wait on the train until it filled to capacity with Standard Riders, meaning it wouldn't actually be faster in the end. Instead, we define a "speed factor," α, which is a function of the proportion of a train classified as Express Riders, $\theta_{express}$. When this proportion is 1—meaning the train is fully Express Riders—the servers can load all passengers 50% faster than normally. When the train is fully Standard Riders, the servers load customers at exactly the same as in previous models. This speed factor is defined as a linear function, resulting in the new service time distribution:

$$\alpha = -0.5 * \theta_{express} + 1, \quad \theta_{express} \in [0,1]$$

$$S_3 \sim \alpha * [U(0.5,1)] + 3 + U(.25,.75)$$

Upon implementing this more complex system, we output a table of relevant outputs. The table of statistics is separated into values for Standard Riders, Express Riders, and "Aggregate," which is data from all Riders, along with a column from these values from Model #2.

Statistic		Standard	Express	Aggregate Value	Baseline Value
Wait Time	Minimum	0.000 mins	0.000 mins	0.000 mins	0.000 mins
	Average	45.200 mins	2.106 mins	39.897 mins	48.839 mins
	Maximum	67.312 mins	4.658 mins	67.312 mins	72.550 mins
Daily Throughput	Minimum	26842	3658	30588	30,776
	Average	27107	3803	30911	31,008
	Maximum	27355	4021	31187	31,252
Abandonment Percentage	Minimum	—	—	4.06 %	3.84 %
	Average	—	—	5.03 %	3.70 %
	Maximum	—	—	6.02 %	5.40 %

Figure 4.2: A table of relevant outputs for the improved model.

The average wait time in our final baseline model using the exact same simulated arrival data as in this model was 48.839 minutes. In comparison, our new model has an aggregate average wait time of 39.897 minutes, amounting a decrease of 18.31 %. A histogram and box plot of the average wait time of 30 simulated days of the improved Model #3 is shown below, along with a box plot and kernel density estimate plot of Model #2 . We see that there are two modes for the distribution at around the average Express Queue wait time and average Standard Queue wait time, respectively. One possible issue which can be seen from this plot is how large the gap between the two peaks are. As we'll explore in future models, an intelligent customer would realize that it is *always* beneficial to get an Express Pass because, except at the beginning of the day or when the Standard Queue is extremely short, the Express Queue has lower wait times. This potentially could lead into issues of Express Capacity. If the system gives out too few passes, it will mimic more closely the baseline model, but if the system gives out too many passes (the critical amount has yet to be determined), the Express Queue will build up. With the current opening probability = 1, the wait time for both queues would increase greatly.

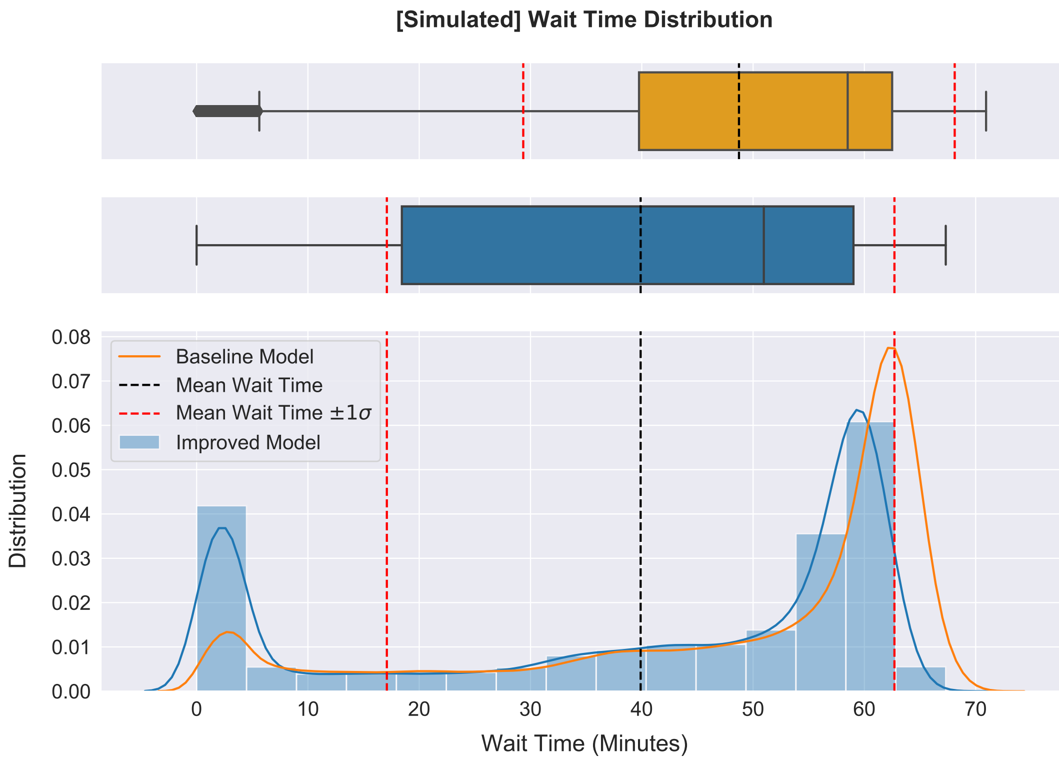


Figure 4.3: Histogram of average wait times for 30 simulated days of the improved system.

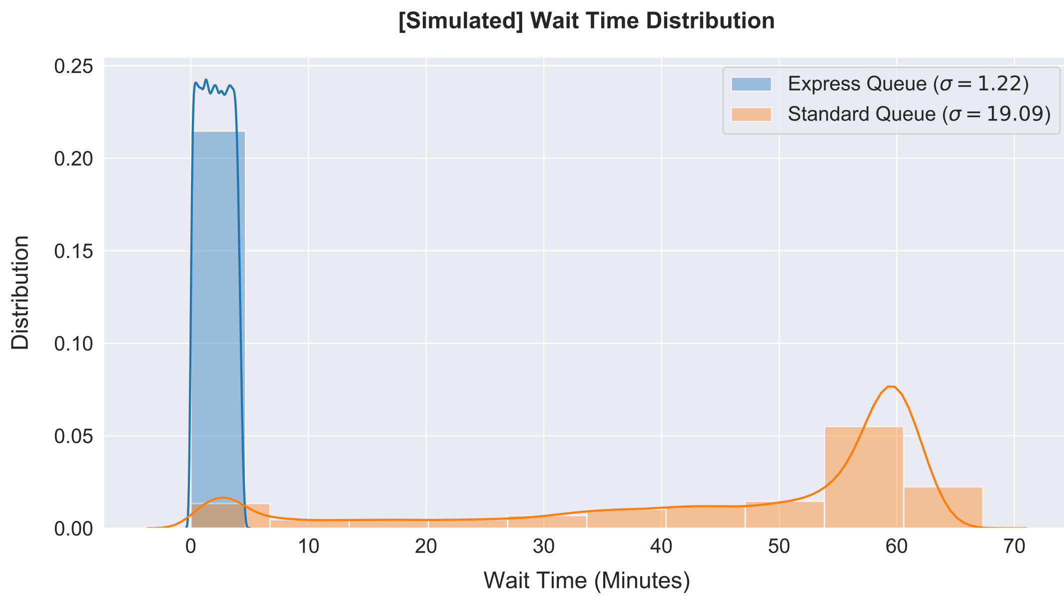


Figure 4.4: Histogram of average wait times for 30 simulated days for the Standard Queue and Express Queue.

Figure 4.5 contains a line plot for the Standard Queue, Express Queue, and Aggregate wait times throughout the day, along with the corresponding wait times in Model #2 . From the plot, we see the wait times clearly follow the same shape as the baseline models, highlighted by a quick increase in wait times as the arrival rate—and as a result, queue length—increases, followed by a gradual flattening as capacity increases throughout the day. At around 4:00pm, arrival rates begin to noticeably decrease while capacity stays high, so wait times decrease through the end of the day. It's in this plot that we can clearly see the impact of the Express Queue: because $p_{open} = 1$, the Express Queue never builds up throughout the day, leading to comparably constant wait times (Figure 4.4 depicts this even more clearly, showing the distributions and standard deviation values for the Standard Queue and Express Queue wait times). This extremely low and constant wait time for the Express Queue "pulls" the Aggregate wait time plot down, explaining why the Aggregate plots looks nearly identical to the Baseline plot, just shifted down $\approx$ 10 minutes. The Standard Queue is also slightly pulled down, though this is mostly due to the fact that the some of the customers that would have entered the Standard Queue went on to join the Express Queue later in the day (a concept that will be discussed at length in Model #4).

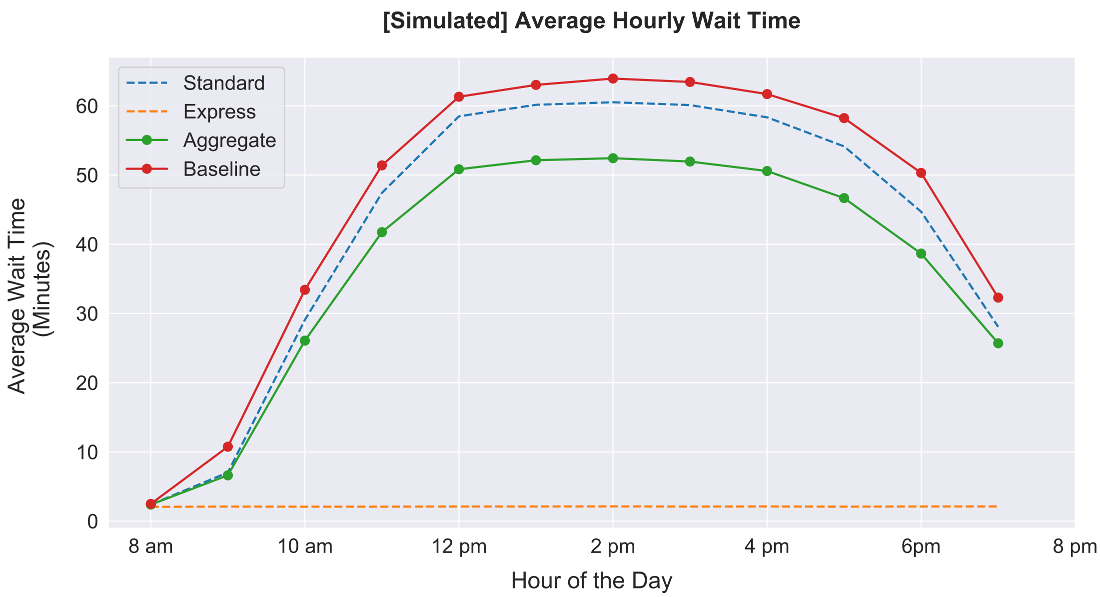


Figure 4.5: A line plot for the Standard Queue, Express Queue, and Aggregate wait times throughout the day.

4.2 Model #4: A Tuned, Improved Queuing System

For our final models, we will begin to "tune" the Improved Model by addressing its key shortcomings and assumptions to create a more optimal *and* realistic system. Although this exploration into improving the current system is just a model, the end goal is feasible implementation, so devising solutions and systems that have no real-world viability has little value.

4.2.1 Sensitivity Analysis of System Assumptions

First, we analyze our static probability assumptions from Model #3. In the improved model, we had no existing data on certain choices or behavior of Express Riders, so we set arbitrary, but realistic probabilities for these actions. Specifically, we set the probability of obtaining an Express Pass instead of joining the Standard Queue or abandoning the system as $p_{express} = 0.15$, and the probability a customer receives an Express Pass but doesn't use it as $p_{unused} = 0.15$. As such, we run the same model but with various combinations of $(p_{express}, p_{unused})$, where each value ranges from 0.05 to 0.5. We then plot the average wait time and difference in wait times between the Standard Queue and Express Queue to analyze the effects of these value changes (along with some random x-axis noise for visual clarity). We note here that in comparison with the tuning improvements below, these aren't changes we can implement in the model directly. Instead, this analysis is to see how our assumptions influenced the model's behavior.

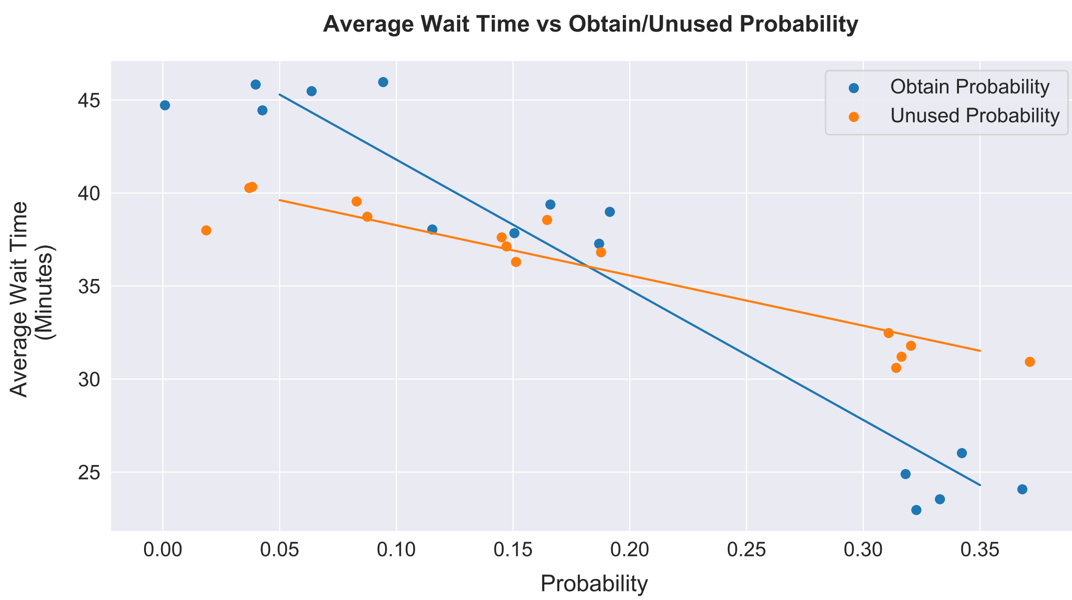


Figure 4.6: A scatter plot of the average wait time versus the probability values for obtaining an Express Pass & Unused Express Pass.

From the two plots above, we see rather similar trends. As the probability of a customer obtaining an Express Pass increases, the average aggregate wait time decreases rather quickly, as does the difference in wait times between the two queue types. This, of course, comes from the fact that with more Express Riders (who are always serviced first), there is a higher proportion of customers with low wait times to "pull" the aggregate mean wait time down. Similarly, the probability of an unused Express Pass is simply an abandonment probability, which decreases the amount of people in the Standard Queue. This in turn limits the number of comparably high wait times which would normally have pulled the aggregate up. This is less powerful of an impact than the increase in obtaining an Express Pass probability, which is why the magnitude slope of the best fit line is small. Again, these are factors we can't truly control in a model, but in conjunction with better data collection, can help to understand how the queuing system works on a macro level.

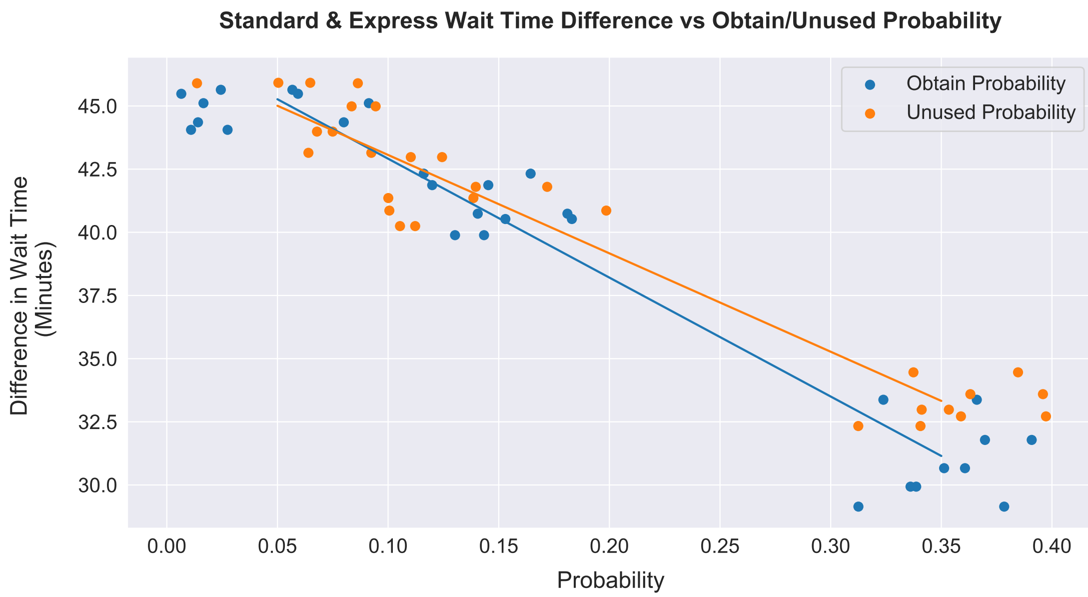


Figure 4.7: A scatter plot of the difference in wait times for Standard and Express queues versus the probability values for obtaining an Express Pass & Unused Express Pass.

4.2.2 Sensitivity Analysis of System Parameters

From here, we move to analyze the actual underlying parameters from which our calculations came. The following models will analyze (1) Opening Probabilities for the service turnstiles, p_{open} (2) Duration of the interval in which an Express Rider can join the Express Queue, $T_{duration}$ and (3) The time delay, T_{delay}, for how long Express Riders must wait before joining the Express Queue.

For the simple case of Model #3, we set the opening probability, $p_{open} = 1$, such that Express Riders are always served first. It became clear, through simulating with varying values of p_{open}, that there is no advantage to changing this value. In order to simplify the model as much as possible p_{open} is kept at 1 (i.e. Express Riders are always served first). See Appendix G for results of this simulation.

In Model #3, an Express Pass was only valid from 30 minutes after receiving to 90 minutes after receiving, meaning $T_{duration} = 1$ and $T_{delay} = 0.5$. The underlying strategy behind implementing a $T_{duration}$ is to effectively "spread out" the arrivals. In a trivial case, let us assume 66 customers arrive to the system in an hour, and 10 of the 66 receive an Express Pass, around 15%. In a given time frame, we have "removed" these 10 arrivals so that the system only needs to serve 56 customers now, and the other 10 customers can be served over a more spread out time period up to 1.5 hours later. This process is ongoing because arrival rates are continuous, so the demand traffic is spread out further and more evenly. To analyze the impact of changing the duration, we simulate the same model with the following change: once a customer receives an Express Pass, they can return and join the Express Queue from $[0.5, T_{duration}]$, hours after receiving, where $T_{duration} = 1, 1.5, 2, ...11.5$. If the park closes or $T_{duration}$ has elapsed, the Express Pass expires. A portion of the result table is shown below, along with relevant plots.

$T_{duration}$	Aggregate	Express	Standard
1.0	40.085 mins	2.113 mins	45.343 mins
2.0	38.522 mins	2.081 mins	43.649 mins
3.0	36.501 mins	2.098 mins	41.228 mins
4.0	35.879 mins	2.107 mins	40.522 mins
5.0	32.342 mins	2.090 mins	36.552 mins
6.0	31.660 mins	2.081 mins	35.701 mins
7.0	31.648 mins	2.079 mins	35.729 mins
8.0	31.138 mins	2.079 mins	35.041 mins
9.0	30.441 mins	2.118 mins	34.294 mins
10.0	30.070 mins	2.067 mins	33.878 mins
11.0	28.410 mins	2.091 mins	32.012 mins

Figure 4.8: A table of values for the Standard, Express, & Aggregate average wait times for various $T_{duration}$ values.

In figure 4.9, we see that increasing $T_{duration}$ consistently decreases the Aggregate and Standard Queue wait times, while keeping the Express Queue wait time relatively constant. It's possible that as $T_{duration}$ gets very large relative to the 12 hour day ($T_{duration} > 7$), there is too much of a limitation on when a customer can receive an Express Pass that the marginal benefit is minimal. For example, if $T_{duration} = 10$ but customers tend to obtain the Express Passes at peak hours (say 11:30am, 3.5 hours into the day), the closure of the park would make the true $T_{duration} < 10$.

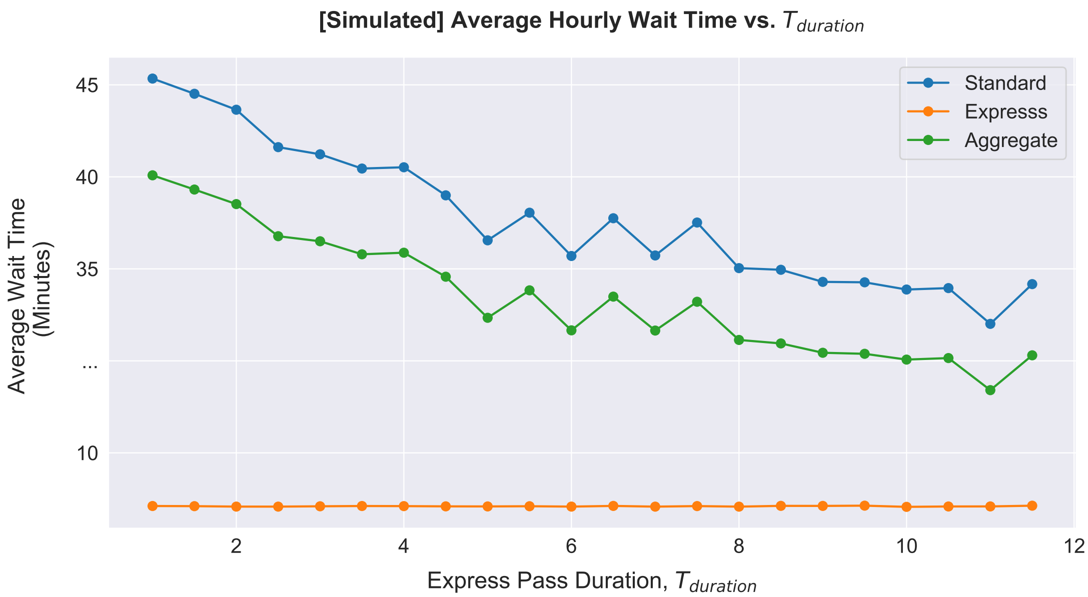


Figure 4.9: A plot of the average wait times versus duration of Express Pass, $T_{duration}$.

In our Exploratory Data Analysis and Model #2 sections, it became clear that arrival rates are not constant throughout the day. One key trend was that after the peak near 11:00am to 12:00pm, arrivals slowly decreased, with the rate of decrease growing after 4:00pm. We use this to our advantage by altering T_{delay} to determine the effects of increasing the delay past 30 minutes. Similar to the analysis on $T_{duration}$, by increasing the start time of the valid interval, we can "push" back customers to hours when the queue is less full. Using the same trivial example as before, let us assume that 66 customers arrive at 11:00am, and 10 of these customers obtain an Express Pass. Under the current regime, these customers could come back to the Express Queue between 11:30am and 12:30pm, both of which are still during the peak of the day with the highest arrival rates. As a result, the queues still build up. Instead, let us consider a more extreme T_{delay} = 5 (meaning the customers must wait 5 hours), with the same $T_{duration}$ = 1 as before. In the same example, the valid interval for these customers would be between 4:00pm and 5:00pm, which have considerably lower arrival rates than the peak hours. Thus, by "delaying" the customer's arrival to a less-crowded hour of the day, we can spread demand traffic throughout the day. This will result in lower average and peak wait times, along with higher predictability for staffing and other service factors in the real world. With this logic, we simulate Model #3 with the following changes: customers who obtain an Express Pass must wait a minimum of T_{delay} hours and maximum of $T_{duration}$ hours before returning to join the Express Queue, T_{delay} = 0.5, 1, 1.5, ... 3, $T_{duration}$ = 0.5, 1, 1.5, ...11. A portion of the result table is shown below, along with relevant plots.

T_{delay}	$T_{duration}$	Standard	Express	Aggregate
2.5	9.0	29.889 mins	2.115 mins	27.144 mins
2.0	10.0	30.799 mins	2.086 mins	27.804 mins
3.0	8.0	30.424 mins	2.119 mins	27.831 mins
3.0	9.0	30.570 mins	2.077 mins	27.894 mins
1.5	8.0	31.362 mins	2.052 mins	28.124 mins
2.5	6.0	31.139 mins	2.103 mins	28.277 mins
2.5	8.0	31.256 mins	2.105 mins	28.369 mins
3.0	6.0	31.300 mins	2.068 mins	28.602 mins
1.0	11.0	32.049 mins	2.092 mins	28.615 mins
1.5	10.0	31.864 mins	2.119 mins	28.629 mins

Figure 4.10: A table of values for the Standard, Express, & Aggregate average wait times for various combinations of $T_{duration}$ and T_{delay} values.

From the table above, we see that the lowest Aggregate wait time is achieved at T_{delay} = 2.5, $T_{duration}$ = 9.0, meaning upon receiving an Express Pass, a customer must wait a minimum of 2.5 hours to join the Express Queue, and a maximum of 9.0 hours until it expires. One interesting thing to note is how high $T_{duration}$ is. In a 12 hour day of operation, unless the arrival is early morning, the end of the day will occur before the Express pass expires.

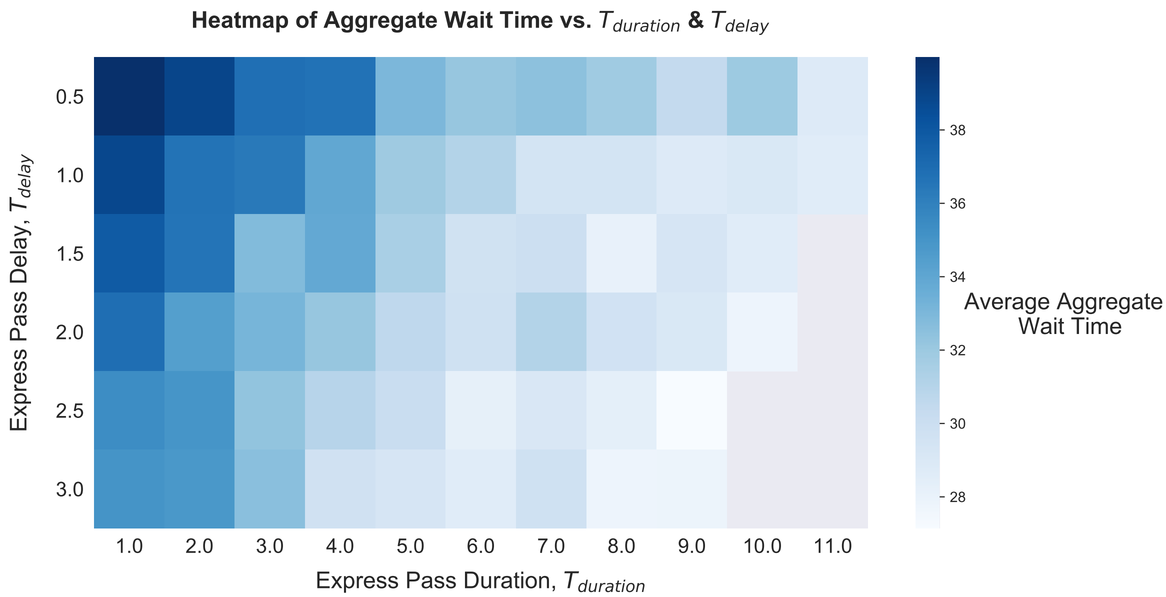


Figure 4.11: A plot of the average wait times versus duration of Express Pass, $T_{duration}$ and delay of Express Pass, T_{delay}.

In order to best visualize the correlation between T_{delay}, $T_{duration}$, and average wait time, a heat map is shown below. The lighter and more white a square is, the lower the wait time. There is obvious correlation between delay and duration, because delay impacts how long a duration can be (e.g. if the delay is 12 hours, then there can be now duration of a valid interval). Regardless, from the heat map, it's clear that increasing the both the duration and delay (with the constraint described earlier) of the valid interval does the best job of pushing back arrivals to even out demand and lower wait times.

Results, Discussion, & Conclusion

We began this exploration into amusement park queuing systems by simulating the Expedition Everest queuing system at its most simple form in what we call our "baseline models," using it to calculate outputs to which we could compare our optimization strategies. We then designed and built more complex queuing systems, which introduced an Reservation-Driven Express Queue, or priority line, in attempt to drive average customer wait time down. With this strategy, we again simulated 30 days of typical customer behavior and found an 18.31% decrease in average customer wait time. Finally, we took a step back to analyze both the assumptions and parameters of our "improved model" to see how each impacted our results. From our parameter sensitivity analysis and subsequent optimization, we were able to further decrease average wait times to achieve an overall decrease of 44.42% from the original model.

The improved models detailed above do a great job providing Disney with a strategy to decrease the amount of time customers spend waiting in lines, but there are areas in which more information is necessary, in order to obtain the most accurate results possible. First, having access to Disney's real data would clearly add a level of detail and accuracy that just can't be rivaled. Specifically, knowing the real load and unload distributions, along with the true hourly arrival rates would help us gauge our errors to build more accurate models. In addition to collecting more accurate data, having a statistical basis for our improved model assumptions of the probability a customer obtains an Express Pass, or true abandonment probabilities (both from the Standard Queue and from an unused pass) would help us model customer decisions better and potentially more dynamically. With these two key changes, we would be able to increase the accuracy of our model, such that the systems and strategies we designed would have real-world feasibility for Disney to implement.

Appendix

6.1 Appendix A: Exploratory Data Analysis

In this iPython notebook, we analyze the third-party data we collected. First, we plot the arrival count data, followed by analysis on the posted wait time data set. We also remove outliers and clean the data sets here.

eda_final

December 14, 2020

[2]:
```
# Importing Libraries
import numpy as np
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import math
from IPython.display import Markdown as md
import warnings
warnings.filterwarnings('ignore')
%matplotlib inline
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import train_test_split, cross_val_score, KFold
from scipy.optimize import minimize
```

0.1 Posted Wait Time Data EDA

[3]:
```
# Opening + Cleaning Posted Wait Time Data
df = pd.read_csv('Posted_Wait_Times.csv')
df['dt'] = pd.to_datetime(df['datetime'])
df.drop(['date', 'datetime', 'SACTMIN'], axis = 1, inplace = True)
df['date'] = df['dt'].dt.date
df['time'] = df['dt'].dt.time
df['hour'] = df['dt'].dt.hour
df['month'] = df['dt'].dt.month
df.drop(['date'], axis = 1, inplace = True)
df.rename(columns = {'SPOSTMIN': 'wait'}, inplace = True)
df.dropna(inplace = True)
df = df[df['wait'] > 0]
df = df[['dt', 'month', 'hour', 'time', 'wait']]
print('Posted Wait Time Data')
df.head(5)
```

```
Posted Wait Time Data
```

```
[3]:                      dt  month  hour      time  wait
     0 2015-01-01 07:47:00      1     7  07:47:00   5.0
     1 2015-01-01 07:54:00      1     7  07:54:00   5.0
     2 2015-01-01 08:05:00      1     8  08:05:00   5.0
     3 2015-01-01 08:12:00      1     8  08:12:00   5.0
     4 2015-01-01 08:19:00      1     8  08:19:00   5.0
```

```
[4]: # Creating Data Frame for November
     november = df[(df['month'] == 11) & (df['hour'] >= 8) & (df['hour'] <= 20)].
      ↪reset_index().drop('index', axis = 1)
     print('November Posted Wait Time Data')
     november.head(5)
```

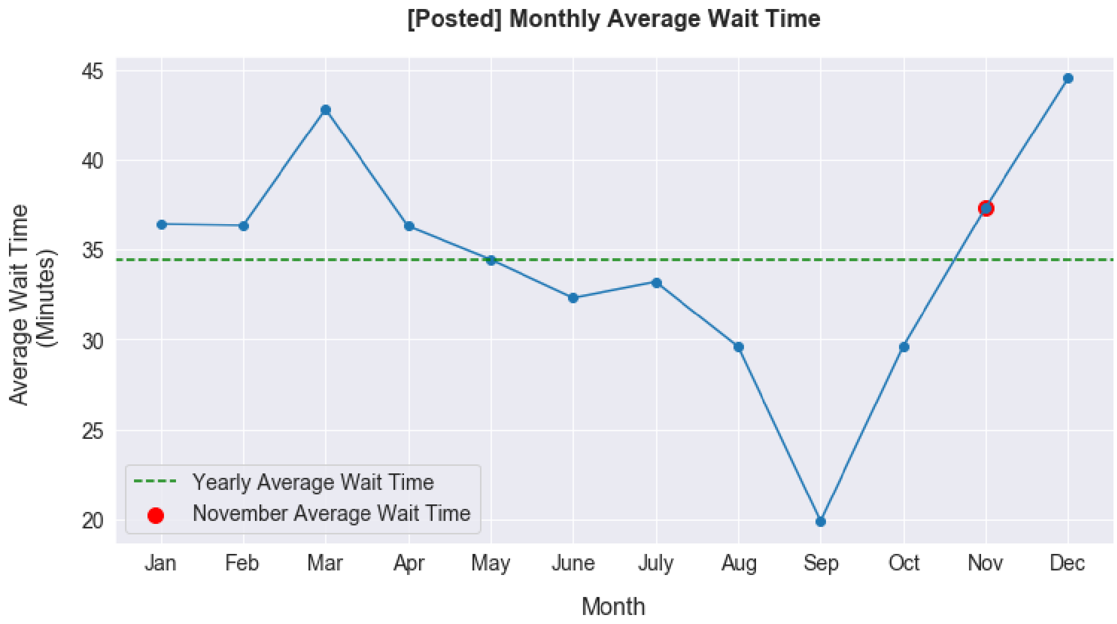
```
November Posted Wait Time Data
```

```
[4]:                      dt  month  hour      time  wait
     0 2015-11-01 08:03:00     11     8  08:03:00   5.0
     1 2015-11-01 08:10:00     11     8  08:10:00   5.0
     2 2015-11-01 08:17:00     11     8  08:17:00   5.0
     3 2015-11-01 08:24:00     11     8  08:24:00   5.0
     4 2015-11-01 08:30:00     11     8  08:30:00   5.0
```

0.2 Exploratory Data Analysis

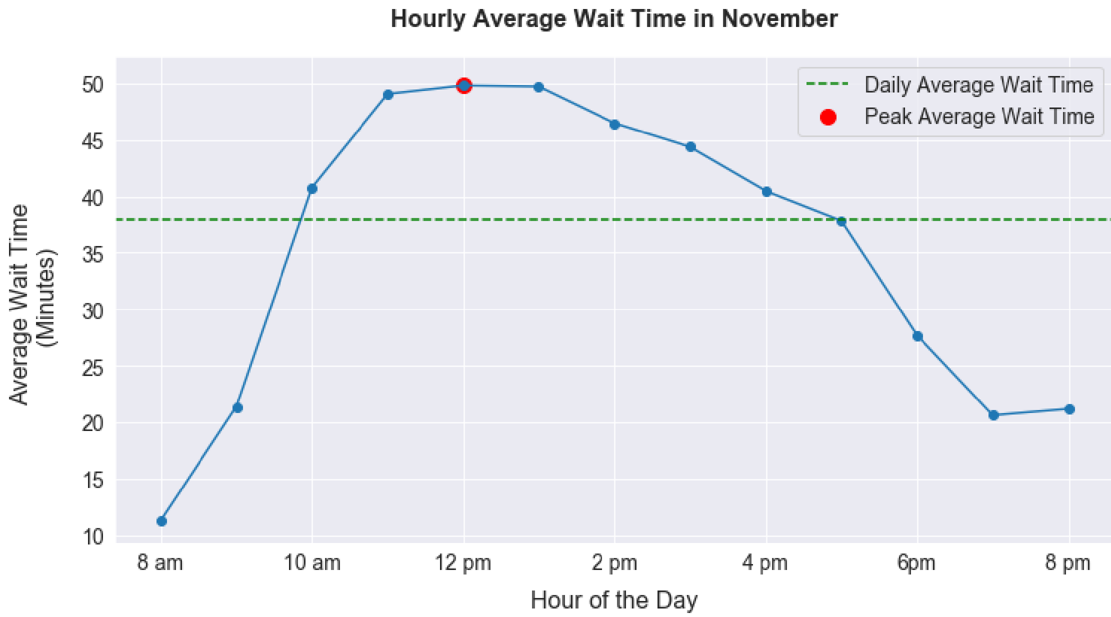
```
[5]: # Plotting Monthly Average Wait Time
     sns.set_style('darkgrid')
     fig0a, ax0a = plt.subplots(figsize = (12,6))


     plt.plot(df.groupby('month').mean()['wait'].index, df.groupby('month').
      ↪mean()['wait'].values, marker = 'o')
     ax0a.axhline(y = np.mean(df['wait']), c = 'g',linestyle = 'dashed', label =␣
      ↪'Yearly Average Wait Time')
     plt.scatter(11, np.mean(df[df['month'] == 11]['wait']), marker = 'o' , s = 100,␣
      ↪c = 'r', label = 'November Average Wait Time')


     plt.title('[Posted] Monthly Average Wait Time', fontsize = 16, pad = 20,␣
      ↪fontweight = 'semibold')
     plt.xlabel('Month', fontsize = 16, labelpad = 15)
     plt.ylabel('Average Wait Time \n (Minutes)', fontsize = 16, labelpad = 15)
     plt.xticks(df.groupby('month').mean()['wait'].index.to_list(), ['Jan', 'Feb',␣
      ↪'Mar', 'Apr', 'May', 'June',
                                                                   'July', 'Aug',␣
      ↪'Sep', 'Oct', 'Nov', 'Dec'])
     plt.yticks(fontsize = 14)
```

```
plt.xticks(fontsize = 14)
plt.legend(prop={'size': 14})
fig0a.savefig('fig0a.pdf')
```



[6]:
```
# Plotting Daily Average Wait Time vs. Hour
sns.set_style('darkgrid')
fig0b, ax0b = plt.subplots(figsize = (12, 6))

grouped_nov = november.groupby('hour').mean()[['wait']]

plt.plot(grouped_nov.index.to_list(), grouped_nov['wait'], marker = 'o')
ax0b.axhline(y = np.mean(november['wait']), c = 'g',linestyle = 'dashed', label␣
 ↪= 'Daily Average Wait Time')
plt.scatter(12, november.groupby('hour', as_index = True).mean()[['wait']].
 ↪max(),
            marker = 'o' , s = 100, c = 'r', label = 'Peak Average Wait Time')


plt.title('Hourly Average Wait Time in November', fontsize = 16, pad = 20,␣
 ↪fontweight = 'semibold')
plt.xlabel('Hour of the Day', fontsize = 16, labelpad = 10)
plt.ylabel('Average Wait Time \n (Minutes)', fontsize = 16, labelpad = 15)
plt.xticks([8,10,12,14,16,18,20], ['8 am', '10 am', '12 pm', '2 pm', '4 pm',␣
 ↪'6pm', '8 pm'], fontsize = 14)
plt.yticks(fontsize = 14)
plt.legend(prop={'size': 14})
```

```
fig0b.savefig('fig0b.pdf')
```



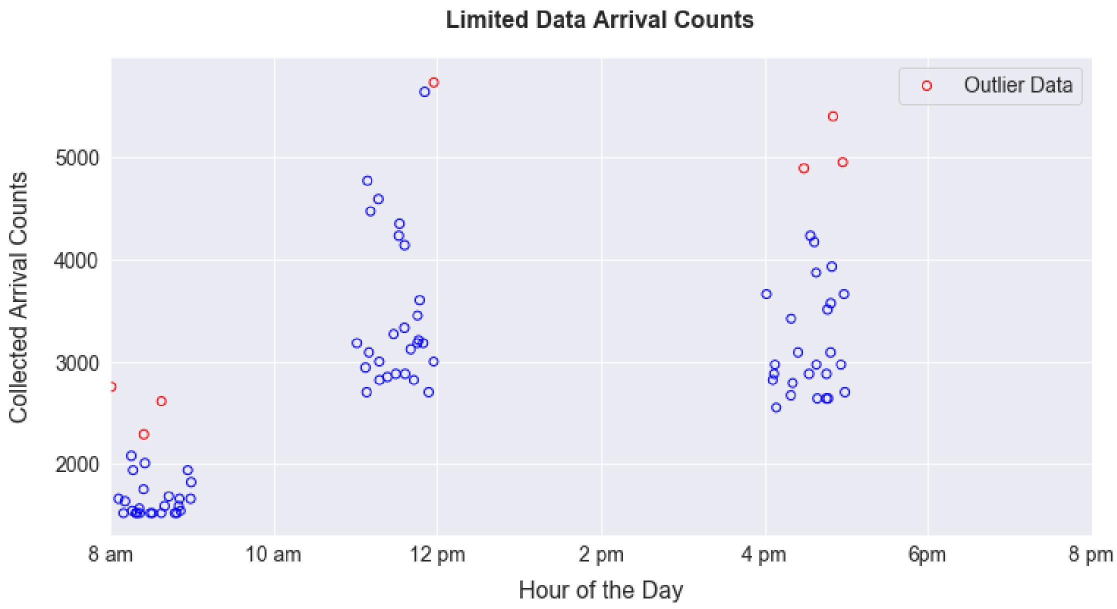
[7]:
```
## Loading in November Arrival Data ##
rates = pd.read_csv('november_arrival_data.csv')
rates['day'] = pd.to_datetime(rates['date'])
rates = rates.drop('date', axis = 1)
rates = rates[['day', 'open', 'peak', 'afternoon']]
rates = rates.rename( columns = {'day' : 'Date',
                                  'open' : '8',
                                  'peak' : '11',
                                  'afternoon' : '4', })
rates = rates.dropna()
rates.head(5)
```

[7]:
```
         Date     8    11       4
0  2019-11-01  1517  3330  3090.0
1  2019-11-02  1540  2820  3870.0
2  2019-11-03  1517  3210  3090.0
3  2019-11-04  1657  3000  2820.0
4  2019-11-05  1517  3180  2640.0
```

[8]:
```
# Removing Outliers
rates2 = rates.drop('Date', axis = 1)
q1 = rates2.quantile(0.25)
q3 = rates2.quantile(0.75)
IQR = q3 - q1
LB = q1 - 1.25*IQR
```

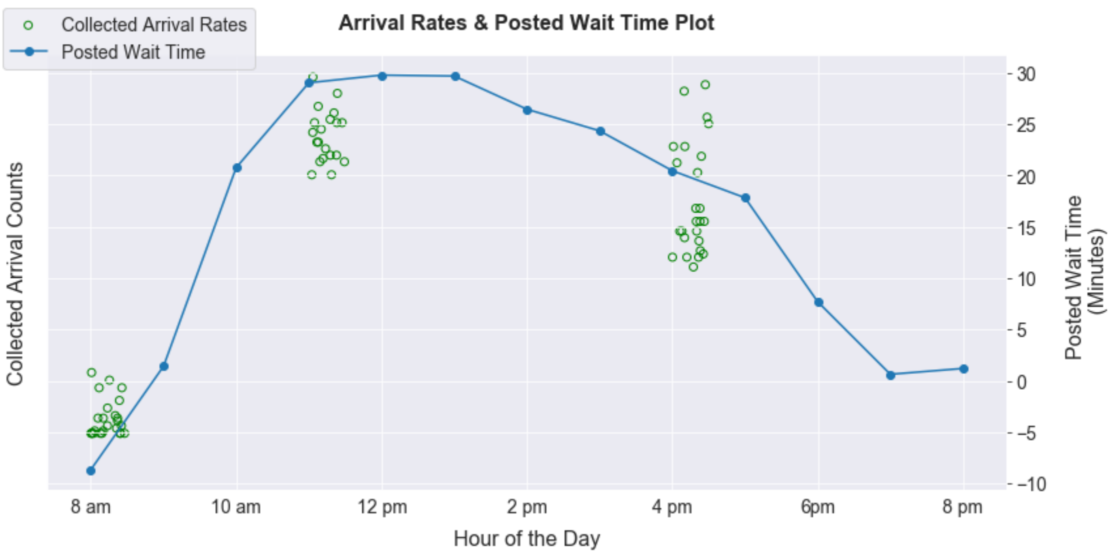
```
UB = q3 + 1.25*IQR

rates_in_8 = rates2[ (rates2['8'].between(LB[0], UB[0])) ]['8'].values
rates_out_8 = rates2[ ~(rates2['8'].between(LB[0], UB[0])) ]['8'].values

rates_in_11 = rates2[ (rates2['11'].between(LB[1], UB[1])) ]['11'].values
rates_out_11 = rates2[ ~(rates2['11'].between(LB[1], UB[1])) ]['11'].values

rates_in_4 = rates2[ (rates2['4'].between(LB[2], UB[2])) ]['4'].values
rates_out_4 = rates2[ ~(rates2['4'].between(LB[2], UB[2])) ]['4'].values
```

[9]:
```
# Scatterplot of Arrivals
sns.set_style('darkgrid')
fig0c, ax0c = plt.subplots(figsize = (12, 6))

plt.scatter(np.random.uniform(0,1, len(rates_in_8)), rates_in_8,␣
 ↪facecolors='none', edgecolors ='b')
plt.scatter(np.random.uniform(3,4, len(rates_in_11)), rates_in_11,␣
 ↪facecolors='none', edgecolors ='b')
plt.scatter(np.random.uniform(8,9, len(rates_in_4)), rates_in_4,␣
 ↪facecolors='none', edgecolors ='b')

plt.scatter(np.random.uniform(0,1, len(rates_out_8)), rates_out_8,␣
 ↪facecolors='none', edgecolors ='r', label = 'Outlier Data')
plt.scatter(np.random.uniform(3,4, len(rates_out_11)), rates_out_11,␣
 ↪facecolors='none', edgecolors ='r')
plt.scatter(np.random.uniform(8,9, len(rates_out_4)), rates_out_4,␣
 ↪facecolors='none', edgecolors ='r')

plt.title('Limited Data Arrival Counts', fontsize = 16, pad = 20, fontweight =␣
 ↪'semibold')
plt.xlabel('Hour of the Day', fontsize = 16, labelpad = 10)
plt.ylabel('Collected Arrival Counts', fontsize = 16, labelpad = 15)
plt.xticks([0,2,4,6,8,10,12], ['8 am', '10 am', '12 pm', '2 pm', '4 pm', '6pm',␣
 ↪'8 pm'], fontsize = 14)
plt.yticks(fontsize = 14)
plt.xlim(0, 12)
plt.legend(prop={'size': 14})
fig0c.savefig('fig0c.pdf')
```



[10]:
```
# Adding Horizontal Noise to Plot

random1, random2, random3 = np.random.uniform(8,8.5, len(rates_in_8)),np.random.
↪uniform(11,11.5, len(rates_in_11)),np.random.uniform(16,16.5,␣
↪len(rates_in_4))



fig, ax1 = plt.subplots(figsize = (12,6))

ax1.set_xlabel('Hour of the Day', fontsize = 16, labelpad = 10)
ax1.set_ylabel('Collected Arrival Counts', fontsize = 16, labelpad = 15)
plt.yticks([],fontsize = 14)

ax1.scatter(random1, rates_in_8-1000, facecolors='none', edgecolors ='g', label␣
↪= 'Collected Arrival Rates')
ax1.scatter(random2, rates_in_11+200, facecolors='none', edgecolors ='g')
ax1.scatter(random3, rates_in_4-500, facecolors='none', edgecolors ='g')
plt.ylim(0,4000)
plt.xticks([8,10,12,14,16,18,20], ['8 am', '10 am', '12 pm', '2 pm', '4 pm',␣
↪'6pm', '8 pm'], fontsize = 14)

ax2 = ax1.twinx()  # instantiate a second axes that shares the same x-axis

color = 'tab:blue'
ax2.set_ylabel('Posted Wait Time \n (Minutes)', fontsize = 16, labelpad = 15)
ax2.plot(grouped_nov.index.to_list(), grouped_nov['wait'] - 20, marker = 'o',␣
↪label = 'Posted Wait Time')
```

```
plt.yticks(fontsize = 14)
plt.xticks([8,10,12,14,16,18,20], ['8 am', '10 am', '12 pm', '2 pm', '4 pm',␣
↪'6pm', '8 pm'], fontsize = 14)
plt.title('Arrival Rates & Posted Wait Time Plot', fontsize = 16, pad = 20,␣
↪fontweight = 'semibold')
fig.tight_layout()
fig.legend(prop={'size': 14}, loc = 'upper left')
fig.savefig('fig0d.pdf')
```



[ ]:

```

```

6.2 Appendix B: Polynomial Regression Derivation

This notebook demonstrates the method for our regression-to-interpolation method. First, it runs polynomial regression using Python's Sci-Kit Learn library to determine the optimal function to fit the posted wait data. Then, it applies a simplified interpolation framework to the posted wait time data and arrival counts.

regression

December 14, 2020

[30]:
```
# Importing Libraries
import numpy as np
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import math
from IPython.display import Markdown as md
import warnings
warnings.filterwarnings('ignore')
%matplotlib inline
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import train_test_split, cross_val_score, KFold
from scipy.optimize import minimize
```

1 Polynomial Regression on Posted Wait Time Data

[31]:
```
# Opening + Cleaning Posted Wait Time Data
df = pd.read_csv('Posted_Wait_Times.csv')
df['dt'] = pd.to_datetime(df['datetime'])
df.drop(['date', 'datetime', 'SACTMIN'], axis = 1, inplace = True)
df['date'] = df['dt'].dt.date
df['time'] = df['dt'].dt.time
df['hour'] = df['dt'].dt.hour
df['month'] = df['dt'].dt.month
df.drop(['date'], axis = 1, inplace = True)
df.rename(columns = {'SPOSTMIN': 'wait'}, inplace = True)
df.dropna(inplace = True)
df = df[df['wait'] > 0]
df = df[['dt', 'month', 'hour', 'time', 'wait']]

# Creating Data Frame for November
november = df[(df['month'] == 11) & (df['hour'] >= 8) & (df['hour'] <= 20)].
↪reset_index().drop('index', axis = 1)
```

[32]:
```
# Creating Feature Matrix + Train/Test/Split
df2 = november[['hour', 'wait']]
X = df2[['hour']].values.reshape(-1,1)
y = df2[['wait']]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.33,␣
↪random_state = 42)
```

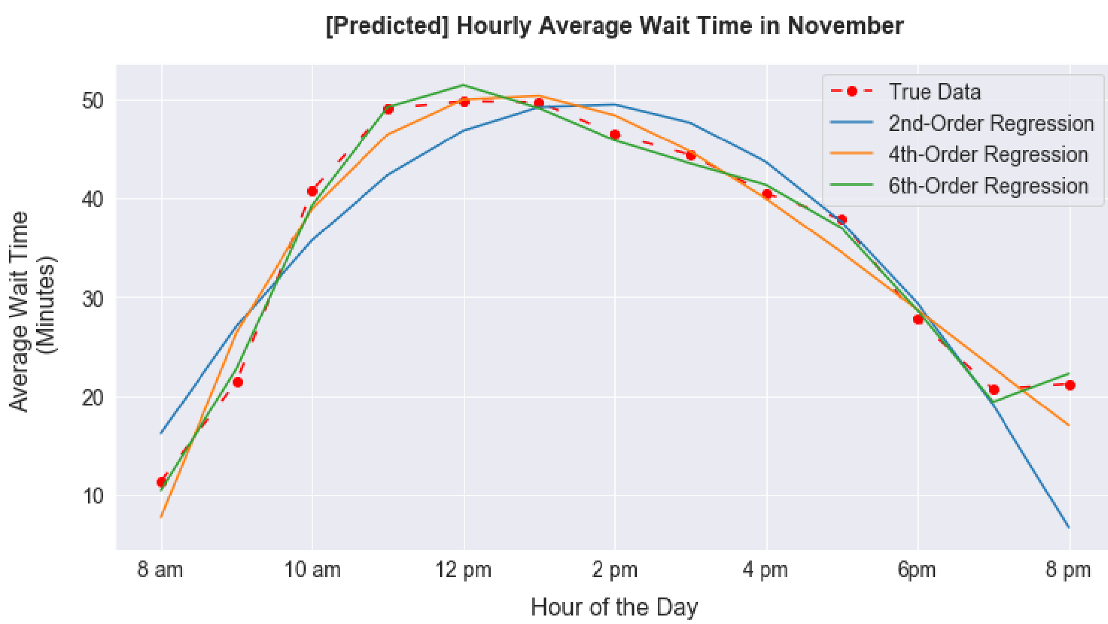
[33]:
```
# Running Polynomial Regression from Order 1 to Order 10

#Linear Regression
lm = LinearRegression()
model1 = lm.fit(X_train, y_train)
y_train_pred1 = model1.predict(X_train)
y_test_pred1 = model1.predict(X_test)
train_error1 = mean_squared_error(y_train, y_train_pred1)
test_error1 = mean_squared_error(y_test, y_test_pred1)

#2nd Order Regression
poly_features2 = PolynomialFeatures(degree = 2)
X_train_poly2 = poly_features2.fit_transform(X_train)
X_test_poly2 = poly_features2.fit_transform(X_test)
pm2 = LinearRegression()
model2 = pm2.fit(X_train_poly2, y_train)
y_train_pred2 = model2.predict(X_train_poly2)
y_test_pred2 = model2.predict(X_test_poly2)
train_error2 = mean_squared_error(y_train, y_train_pred2)
test_error2 = mean_squared_error(y_test, y_test_pred2)

#3rd Order Regression
poly_features3 = PolynomialFeatures(degree = 3)
X_train_poly3 = poly_features3.fit_transform(X_train)
X_test_poly3 = poly_features3.fit_transform(X_test)
pm3 = LinearRegression()
model3 = pm3.fit(X_train_poly3, y_train)
y_train_pred3 = model3.predict(X_train_poly3)
y_test_pred3 = model3.predict(X_test_poly3)
train_error3 = mean_squared_error(y_train, y_train_pred3)
test_error3 = mean_squared_error(y_test, y_test_pred3)

#4th Order Regression
poly_features4 = PolynomialFeatures(degree = 4)
X_train_poly4 = poly_features4.fit_transform(X_train)
X_test_poly4 = poly_features4.fit_transform(X_test)
pm4 = LinearRegression()
model4 = pm4.fit(X_train_poly4, y_train)
y_train_pred4 = model4.predict(X_train_poly4)
y_test_pred4 = model4.predict(X_test_poly4)
```

```
train_error4 = mean_squared_error(y_train, y_train_pred4)
test_error4 = mean_squared_error(y_test, y_test_pred4)

#6th Order Regression
poly_features6 = PolynomialFeatures(degree = 6)
X_train_poly6 = poly_features6.fit_transform(X_train)
X_test_poly6 = poly_features6.fit_transform(X_test)
pm6 = LinearRegression()
model6 = pm6.fit(X_train_poly6, y_train)
y_train_pred6 = model6.predict(X_train_poly6)
y_test_pred6 = model6.predict(X_test_poly6)
train_error6 = mean_squared_error(y_train, y_train_pred6)
test_error6 = mean_squared_error(y_test, y_test_pred6)
```

[35]:
```
# Creating Data Frames
frame2 = pd.DataFrame(data = {'hour' : X_train.flatten(),
                              'wait' : y_train_pred2.flatten()}).
 ↪groupby('hour').mean().sort_index()
frame4 = pd.DataFrame(data = {'hour' : X_train.flatten(),
                              'wait' : y_train_pred4.flatten()}).
 ↪groupby('hour').mean().sort_index()
frame6 = pd.DataFrame(data = {'hour' : X_train.flatten(),
                              'wait' : y_train_pred6.flatten()}).
 ↪groupby('hour').mean().sort_index()
november_grouped = november.groupby('hour').mean()
```

[36]:
```
# Plotting Predicted Values
frame2 = pd.DataFrame(data = {'hour' : X_train.flatten(),
                              'wait' : y_train_pred2.flatten()}).
 ↪groupby('hour').mean().sort_index()
frame4 = pd.DataFrame(data = {'hour' : X_train.flatten(),
                              'wait' : y_train_pred4.flatten()}).
 ↪groupby('hour').mean().sort_index()
frame6 = pd.DataFrame(data = {'hour' : X_train.flatten(),
                              'wait' : y_train_pred6.flatten()}).
 ↪groupby('hour').mean().sort_index()
november_grouped = november.groupby('hour').mean()

sns.set_style('darkgrid')
figRa, axRa = plt.subplots(figsize = (12, 6))

plt.plot(november_grouped.index, november_grouped['wait'], linestyle = (0,␣
 ↪(5,10)), marker = 'o', c = 'r', label = 'True Data')
plt.plot(frame2.index, frame2['wait'], label = '2nd-Order Regression')
plt.plot(frame4.index, frame4['wait'], label = '4th-Order Regression')
plt.plot(frame6.index, frame6['wait'], label = '6th-Order Regression')
```

```
plt.title('[Predicted] Hourly Average Wait Time in November', fontsize = 16,␣
↪pad = 20, fontweight = 'semibold')
plt.xlabel('Hour of the Day', fontsize = 16, labelpad = 10)
plt.ylabel('Average Wait Time \n (Minutes)', fontsize = 16, labelpad = 15)
plt.xticks([8,10,12,14,16,18,20], ['8 am', '10 am', '12 pm', '2 pm', '4 pm',␣
↪'6pm', '8 pm'], fontsize = 14)
plt.yticks(fontsize = 14)
plt.legend(prop={'size': 14})

figRa.savefig('figRa.pdf')
```



[9]:
```
#Creating Cross Validation + Errors Data Frame

test_errors = []
train_errors = []
for i in range(1,11):
    y_train_current = y_train
    y_test_current = y_test
    X_train_current = PolynomialFeatures(degree = i).fit_transform(X_train)
    X_test_current = PolynomialFeatures(degree = i).fit_transform(X_test)
    model = LinearRegression().fit(X_train_current, y_train_current)

    train_rmse = np.sqrt(mean_squared_error(y_train_current, model.
↪predict(X_train_current).flatten()))
    train_errors = np.append(train_errors, train_rmse)
```

```
    test_rmse = np.sqrt(mean_squared_error(y_test_current, model.
↪predict(X_test_current).flatten()))
    test_errors = np.append(test_errors, test_rmse)

regression_df1 = pd.DataFrame()
regression_df1['Regression Type'] = ['Degree-' + str(order) + ' Polynomial' for␣
↪order in range(1,11)]
regression_df1['Train Error'] = np.round(train_errors, 3)
regression_df1['Test Error'] = np.round(test_errors, 3)

crossvalidation = KFold(n_splits=5, random_state=1, shuffle=False)
total_score = []
for i in range(1,11):
    poly = PolynomialFeatures(degree=i)
    X_current = poly.fit_transform(X)
    model = lm.fit(X_current, y)
    scores = cross_val_score(model, X_current, y,␣
↪scoring="neg_mean_squared_error", cv=crossvalidation,
 n_jobs=1)
    total_score = np.append(total_score, scores)
regression_df = pd.DataFrame()
regression_df['Regression Type'] = ['Degree-' + str(order) + ' Polynomial' for␣
↪order in range(1,11)]
regression_df['RMSE'] = np.round(np.array([25.634305744880873, 23.
↪389330953566226, 23.120532713227643, 23.127462673757762, 23.12993466271858,␣
↪23.044671778090684, 23.04481202950483, 23.044859569459703,  23.
↪047102466776305, 23.050022635585513]), 3)
regression_df['STD'] = np.round(np.array([11.969822654542057, 10.7124043947728,␣
↪10.854200283971895, 10.844544549366471, 10.838287601973757, 10.
↪816940757152038,10.818146197513986,10.809363710112518, 10.81223137452443,10.
↪813004433120884]), 3)

totaldf = pd.merge(regression_df, regression_df1)
totaldf = totaldf.rename(columns = {'RMSE': 'Cross-Val RMSE',
                         'STD':'Cross-Val Std',
                        'Train Error': 'Train RMSE',
                        'Test Error' : 'Test RMSE'})
totaldf = totaldf[['Regression Type', 'Train RMSE', 'Cross-Val RMSE',␣
↪'Cross-Val Std', 'Test RMSE']]
totaldf
```

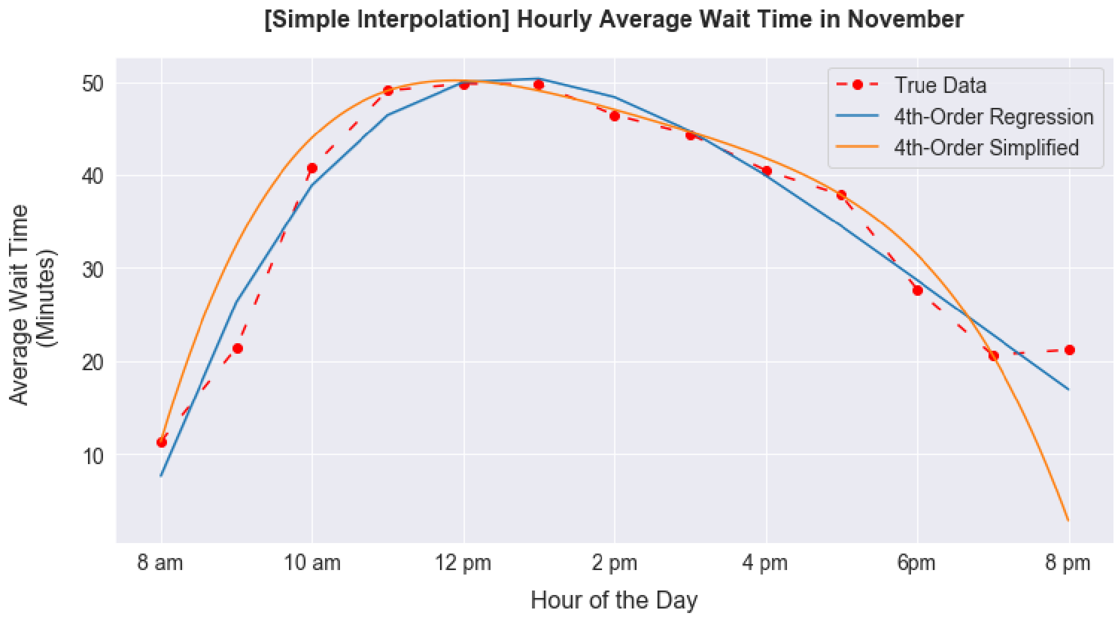
[9]:

	Regression Type	Train RMSE	Cross-Val RMSE	Cross-Val Std	Test RMSE
0	Degree-1 Polynomial	25.007	25.634	11.970	25.219
1	Degree-2 Polynomial	22.505	23.389	10.712	22.706
2	Degree-3 Polynomial	22.232	23.121	10.854	22.456

3	Degree-4 Polynomial	22.227	23.127	10.845	22.458
4	Degree-5 Polynomial	22.225	23.130	10.838	22.455
5	Degree-6 Polynomial	22.112	23.045	10.817	22.371
6	Degree-7 Polynomial	22.102	23.045	10.818	22.367
7	Degree-8 Polynomial	22.100	23.045	10.809	22.360
8	Degree-9 Polynomial	22.101	23.047	10.812	22.363
9	Degree-10 Polynomial	22.104	23.050	10.813	22.368

[38]:
```
# Plotting Simple Interp + 4th Order Regression
df5 = pd.DataFrame()
df5['hour1'] = X_train.flatten()
df5['ypred4'] = y_train_pred4.flatten()
df5['hour1'].replace({8:0, 9:1, 10:2, 11:3, 12:4, 13:5, 14:6, 15:7, 16:8, 17:9,␣
↪18:10, 19:11, 20:12}, inplace = True)
df5.head(5)

a = november_grouped.reset_index().reset_index().drop(['hour', 'month'], axis =␣
↪1).rename(columns = {'index' : 'hour'})
a

df5 = pd.DataFrame()
df5['hour1'] = X_train.flatten()
df5['ypred4'] = y_train_pred4.flatten()
df5['hour1'].replace({8:0, 9:1, 10:2, 11:3, 12:4, 13:5, 14:6, 15:7, 16:8, 17:9,␣
↪18:10, 19:11, 20:12}, inplace = True)
df5.head(5)

def f(t):
    return -.0263577*t**4 + .675513*t**3 + -6.62629*t**2 + 27.0795*t + 11.3421
xxx = np.arange(0, 12, .001)

sns.set_style('darkgrid')
figRb, axRb = plt.subplots(figsize = (12, 6))

plt.plot(a.index, a['wait'], 'o',linestyle = (0, (5,10)), c = 'r', label =␣
↪'True Data')
sns.lineplot(df5['hour1'], df5['ypred4'], label = '4th-Order Regression',␣
↪dashes = True)
plt.plot(xxx, f(xxx), label = '4th-Order Simplified')

plt.title('[Simple Interpolation] Hourly Average Wait Time in November',␣
↪fontsize = 16, pad = 20, fontweight = 'semibold')
plt.xlabel('Hour of the Day', fontsize = 16, labelpad = 10)
plt.xticks([0,2,4,6,8,10,12], ['8 am', '10 am', '12 pm', '2 pm', '4 pm', '6pm',␣
↪'8 pm'], fontsize = 14)
```

```
plt.ylabel('Average Wait Time \n (Minutes)', fontsize = 16, labelpad = 15)
plt.yticks(fontsize = 14)
plt.legend(prop={'size': 14})

figRb.savefig('figRb.pdf')
```



---

2 Applying the Framework to Arrival Data

[39]:
```
# Loading Rates Data + Removing Outliers
rates = pd.read_csv('november_arrival_data.csv')
rates['day'] = pd.to_datetime(rates['date'])
rates = rates.drop('date', axis = 1)
rates = rates[['day', 'open', 'peak', 'afternoon']]
rates = rates.rename( columns = {'day' : 'Date',
                                  'open' : '8',
                                  'peak' : '11',
                                  'afternoon' : '4', })
rates = rates.dropna()


# Removing Outliers
rates2 = rates.drop('Date', axis = 1)
q1 = rates2.quantile(0.25)
q3 = rates2.quantile(0.75)
```

```
IQR = q3 - q1
LB = q1 - 1.25*IQR
UB = q3 + 1.25*IQR

rates_in_8 = rates2[ (rates2['8'].between(LB[0], UB[0])) ][['8']]
rates_out_8 = rates2[ ~(rates2['8'].between(LB[0], UB[0])) ][['8']]

rates_in_11 = rates2[ (rates2['11'].between(LB[1], UB[1])) ][['11']]
rates_out_11 = rates2[ ~(rates2['11'].between(LB[1], UB[1])) ][['11']]

rates_in_4 = rates2[ (rates2['4'].between(LB[2], UB[2])) ][['4']]
rates_out_4 = rates2[ ~(rates2['4'].between(LB[2], UB[2])) ][['4']]

rates_in = pd.concat([rates_in_8,rates_in_11, rates_in_4], ignore_index=True,␣
 ↪axis=1)
rates_out = pd.concat([rates_out_8,rates_out_11, rates_out_4],␣
 ↪ignore_index=True, axis=1)
```

[40]:
```
# Means & Medians after Removing Outliers

old_means = [np.mean(rates2.iloc[:,i]) for i in np.arange(0,3)]
old_medians = [np.median(rates2.iloc[:,i]) for i in np.arange(0,3)]

new_means = [np.mean(rates_in[i].dropna()) for i in np.arange(0,3)]
new_medians = [np.median(rates_in[i].dropna()) for i in np.arange(0,3)]

print('Old Means: ', np.round(old_means, 2))
print('New Means: ', np.round(new_means,2))

print('Old Medians: ', np.round(old_medians, 2))
print('New Medians: ', np.round(new_medians,2))
```

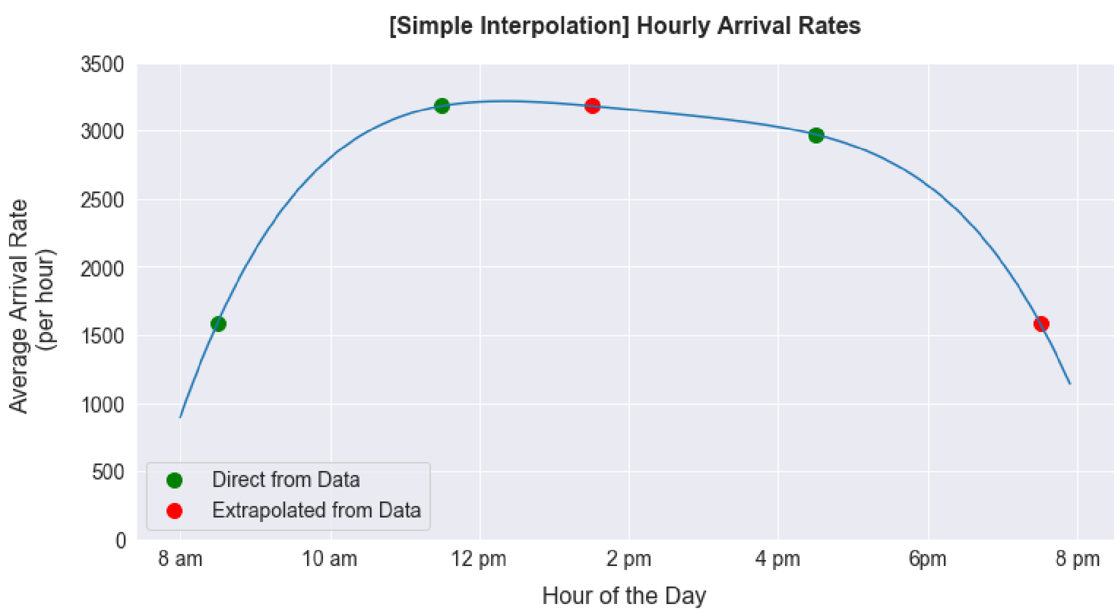
```
Old Means:  [1747.68 3540.   3372.86]
New Means:  [1651.28 3458.89 3168.  ]
Old Medians:  [1610. 3180. 3030.]
New Medians:  [1587. 3180. 2970.]
```

[41]:
```
## Find the Coefficients for our Fourth-Order Arrival Rate Function ##
p1 = [0.5, np.round(new_medians[0], 3)] #8:00 am
p2 = [3.5, np.round(new_medians[1], 3)] #11:00 am
p3 = [5.5, np.round(new_medians[1], 3)] #1:00 pm
p4 = [8.5, np.round(new_medians[2], 3)] #4:00 pm
p5 = [11.5,np.round(new_medians[0], 3)] #7:00 pm

A = [[p1[0]**i for i in np.arange(5)], [p2[0]**i for i in np.arange(5)],␣
 ↪[p3[0]**i for i in np.arange(5)],
     [p4[0]**i for i in np.arange(5)], [p5[0]**i for i in np.arange(5)]]
```

```
B = [p1[1], p2[1], p3[1], p4[1], p1[1]]
x = np.linalg.inv(A).dot(B)
```

[42]:
```
# Defining Arrival Rate Function
def f_arrival(t):
    return x[0] + x[1]*t + x[2]*t**2 + x[3]*t**3 + x[4]*t**4
```

[43]:
```
sns.set_style('darkgrid')
figRc, axRc = plt.subplots(figsize = (12, 6))

plt.plot(np.arange(0,12,0.1), f_arrival(np.arange(0,12,0.1)))
plt.scatter(p1[0], p1[1], s = 100, c = 'g', label = 'Direct from Data')
plt.scatter(p2[0], p2[1], s = 100, c = 'g')
plt.scatter(p3[0], p3[1], s = 100, c = 'r', label = 'Extrapolated from Data')
plt.scatter(p4[0], p4[1], s = 100, c = 'g')
plt.scatter(p5[0], p5[1], s = 100, c = 'r')


plt.title('[Simple Interpolation] Hourly Arrival Rates', fontsize = 16, pad =␣
 ↪20, fontweight = 'semibold')
plt.xlabel('Hour of the Day', fontsize = 16, labelpad = 10)
plt.ylabel('Average Arrival Rate \n (per hour)', fontsize = 16, labelpad = 15)
plt.xticks([0,2,4,6,8,10,12], ['8 am', '10 am', '12 pm', '2 pm', '4 pm', '6pm',␣
 ↪'8 pm'], fontsize = 14)
plt.yticks(fontsize = 14)
plt.ylim(0, 3500)
plt.legend(prop={'size': 14})

figRc.savefig('figRc.pdf')
```



58

```
[ ]:
```

6.3 Appendix C: Generating Arrival Data using Rejection Sampling

Building on Appendix B, this notebook implements a rejection sampling method on the arrival rate 4th-order function an exports a csv file for each simulated day (30 days in this case).

arrival_data_final

December 14, 2020

[1]:
```python
## Importing Libraries ##
import numpy as np
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import math
from IPython.display import Markdown as md
import warnings
warnings.filterwarnings('ignore')
from scipy.optimize import minimize
```

[2]:
```python
## Defining Classes and Stuff ##

##############################
class PoissonProcess():
    def __init__(self, lam, T):
        self.lam = lam
        self.T = T
        self.simulate()

    def simulate(self, method='inter_arrival_time'):
        if method == 'inter_arrival_time':
            N = int(self.lam * self.T * 1.3)
            inter_ls = np.random.exponential(1/self.lam, size=N)
            arrival_time_ls = np.cumsum(inter_ls)
            self.arrival_time_ls = arrival_time_ls[arrival_time_ls <= self.T]
        if method == 'uniformity_property':
            N = np.random.poisson(self.T * self.lam)
            arrival_time_ls = np.random.uniform(0, self.T, size=N)
            self.arrival_time_ls = np.sort(arrival_time_ls)

    def get_arrival_time(self):
        return self.arrival_time_ls

    def print_parameter(self):
        print('lambda = {}, T = {}'.format(self.lam, self.T))
```

```
    def N_t(self, t):
        assert t >= 0
        assert t <= self.T
        if t == 0:
            return 0
        else:
            return np.argmax(self.arrival_time_ls > t)

    def plot_N_t(self, color='r',alpha=1):
        positive_inf = max(self.arrival_time_ls) * 1.2
        negative_inf = - max(self.arrival_time_ls) * 0.1
        n_arrival = len(self.arrival_time_ls)
        x_ls = np.concatenate([[negative_inf, 0], np.repeat(self.
↪arrival_time_ls,2), [positive_inf]])
        y_ls = np.concatenate([[0], np.repeat(np.arange(n_arrival + 1),2)])
        plt.plot(x_ls, y_ls, c=color, alpha=alpha)

#############################
```

0.1 Implementing Rejection Sampling

[75]:
```
## Loading in November Arrival Data ##
rates = pd.read_csv('november_arrival_data.csv')
rates['day'] = pd.to_datetime(rates['date'])
rates = rates.drop('date', axis = 1)
rates = rates[['day', 'open', 'peak', 'afternoon']]
rates = rates.rename( columns = {'day' : 'Date',
                                 'open' : '8',
                                 'peak' : '11',
                                 'afternoon' : '4', })
rates = rates.dropna()
rates.head(5)
```

[75]:

	Date	8	11	4
0	2019-11-01	1517	3330	3090.0
1	2019-11-02	1540	2820	3870.0
2	2019-11-03	1517	3210	3090.0
3	2019-11-04	1657	3000	2820.0
4	2019-11-05	1517	3180	2640.0

[76]:
```
# Removing Outliers
rates2 = rates.drop('Date', axis = 1)
q1 = rates2.quantile(0.25)
q3 = rates2.quantile(0.75)
```

```
IQR = q3 - q1
LB = q1 - 1.25*IQR
UB = q3 + 1.25*IQR

rates_in_8 = rates2[ (rates2['8'].between(LB[0], UB[0])) ][['8']]
rates_out_8 = rates2[ ~(rates2['8'].between(LB[0], UB[0])) ][['8']]

rates_in_11 = rates2[ (rates2['11'].between(LB[1], UB[1])) ][['11']]
rates_out_11 = rates2[ ~(rates2['11'].between(LB[1], UB[1])) ][['11']]

rates_in_4 = rates2[ (rates2['4'].between(LB[2], UB[2])) ][['4']]
rates_out_4 = rates2[ ~(rates2['4'].between(LB[2], UB[2])) ][['4']]

rates_in = pd.concat([rates_in_8,rates_in_11, rates_in_4], ignore_index=True,␣
 ↪axis=1)
rates_out = pd.concat([rates_out_8,rates_out_11, rates_out_4],␣
 ↪ignore_index=True, axis=1)
```

[78]:
```
# Calculating Arrival Rate for Model # 1
rates_in_flat = rates_in.values.flatten()
rates_in_flat = rates_in_flat[~np.isnan(rates_in_flat)]
print('Model #1 Arrival Rate: ', np.median(rates_in_flat), ' customers/hr')
```

```
Model #1 Arrival Rate:  2880.0  customers/hr
```

[80]:
```
# Means & Medians after Removing Outliers

old_means = [np.mean(rates2.iloc[:,i]) for i in np.arange(0,3)]
old_medians = [np.median(rates2.iloc[:,i]) for i in np.arange(0,3)]

new_means = [np.mean(rates_in[i].dropna()) for i in np.arange(0,3)]
new_medians = [np.median(rates_in[i].dropna()) for i in np.arange(0,3)]

print('Old Means: ', np.round(old_means, 2))
print('New Means: ', np.round(new_means,2))

print('Old Medians: ', np.round(old_medians, 2))
print('New Medians: ', np.round(new_medians,2))
```

```
Old Means:  [1747.68 3540.   3372.86]
New Means:  [1651.28 3458.89 3168.  ]
Old Medians:  [1610. 3180. 3030.]
New Medians:  [1587. 3180. 2970.]
```

[85]:
```
## Find the Coefficients for our Fourth-Order Arrival Rate Function ##
p1 = [0.5, np.round(new_medians[0], 3)] #8:00 am
p2 = [3.5, np.round(new_medians[1], 3)] #11:00 am
```

```
p3 = [5.5, np.round(new_medians[1], 3)] #1:00 pm
p4 = [8.5, np.round(new_medians[2], 3)] #4:00 pm
p5 = [11.5,np.round(new_medians[0], 3)] #7:00 pm

A = [[p1[0]**i for i in np.arange(5)], [p2[0]**i for i in np.arange(5)],
 ↪[p3[0]**i for i in np.arange(5)],
     [p4[0]**i for i in np.arange(5)], [p5[0]**i for i in np.arange(5)]]
B = [p1[1], p2[1], p3[1], p4[1], p1[1]]
x = np.linalg.inv(A).dot(B)
```

[86]:
```
## Time-Varying Arrival Rate Function ##
def ArrivalRate_Function(t):
    return x[0] + x[1]*t + x[2]*t**2 + x[3]*t**3 + x[4]*t**4
```

[87]:
```
## Rejection Sampling to Simulate a Time-Inhomogeneous Poisson Process (arrival
 ↪rate varies continuously) ##
def RejectionSampling(rate_function):



    ## Differentiation of Rate Function  ##
    optimized = minimize(lambda t: -rate_function(t), 0, tol=1e-9)
    Max_ArrRate = -optimized.fun
    timeof_Max_ArrRate = optimized.x

    ## Original Arrival List (simulated with the maximum rate)  ##
    original_process = PoissonProcess(lam = Max_ArrRate, T = 11.999)
    original_arrival_ls = original_process.get_arrival_time()
    num_original_arrivals = len(original_arrival_ls)
    #print("num_original_arrivals = ",num_original_arrivals)

    ## Thinning Process  ##
    accepted_arrival_ls = []

    for arr_time in original_arrival_ls:
        u = np.random.uniform(0,1)
        keep_prob = np.round(rate_function(arr_time),3) / round(Max_ArrRate,3)
        if abs(keep_prob<0.00001):
            keep_prob=0

        binom = np.random.binomial(n = 1, p = keep_prob)
        if binom:
            accepted_arrival_ls = np.append(accepted_arrival_ls, np.
 ↪round(arr_time,3))

    return np.round(accepted_arrival_ls,3)
```

[88]:

```
## Iteratively generate 30 days of arrivals to use for models ##
for day in np.arange(1,31):
    arrivals = RejectionSampling(ArrivalRate_Function)
    dict = {"ArrivalTime":arrivals}
    df = pd.DataFrame(dict)
    df.to_csv('data/day'+str(day)+'_arrivals.csv', index=False)
```

[ ]:

```
```

6.4 Appendix D: Model #1

This notebook generates arrivals with constant arrival rate and performs analysis on the naive model.

model_1_final

December 14, 2020

[1]:
```python
## Importing Libraries ##

import numpy as np
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import math
from IPython.display import Markdown as md
import warnings
warnings.filterwarnings('ignore')
# %matplotlib inline
from scipy.optimize import minimize
import time
```

[2]:
```python
## This cell defines our Poission Process, Customer, and Customer List Classes␣
↪##

############################
class PoissonProcess():
    def __init__(self, lam, T):
        self.lam = lam
        self.T = T
        self.simulate()


    def simulate(self, method='inter_arrival_time'):
        if method == 'inter_arrival_time':
            N = int(self.lam * self.T * 1.3)
            inter_ls = np.random.exponential(1/self.lam, size=N)
            arrival_time_ls = np.cumsum(inter_ls)
            self.arrival_time_ls = arrival_time_ls[arrival_time_ls <= self.T]
        if method == 'uniformity_property':
            N = np.random.poisson(self.T * self.lam)
            arrival_time_ls = np.random.uniform(0, self.T, size=N)
            self.arrival_time_ls = np.sort(arrival_time_ls)
```

```python
    def get_arrival_time(self):
        return self.arrival_time_ls


    def print_parameter(self):
        print('lambda = {}, T = {}'.format(self.lam, self.T))


    def N_t(self, t):
        assert t >= 0
        assert t <= self.T
        if t == 0:
            return 0
        else:
            return np.argmax(self.arrival_time_ls > t)


    def plot_N_t(self, color='r',alpha=1):
        positive_inf = max(self.arrival_time_ls) * 1.2
        negative_inf = - max(self.arrival_time_ls) * 0.1
        n_arrival = len(self.arrival_time_ls)
        x_ls = np.concatenate([[negative_inf, 0], np.repeat(self.
 ↪arrival_time_ls,2), [positive_inf]])
        y_ls = np.concatenate([[0], np.repeat(np.arange(n_arrival + 1),2)])
        plt.plot(x_ls, y_ls, c=color, alpha=alpha)

#############################
class Customer():
    def __init__(self, arrival_time=0, ctype='normal',wait_time=None):
        self.arrival_time = arrival_time
        self.ctype = ctype
        self.wait_time = wait_time


    def abandon_prob(self,prev_cust_wait):

        abandon_probs = {"<40":.005, ">=40<50":0.015, ">=50<60":0.03, ">=60<75":
 ↪0.08, ">=75<90":0.1,
                                 ">=90<105":0.1, ">=105<120":0.1, ">=120<180":0.1,␣
 ↪">=180":0.15,
                                }
        ### 170 is <<< 1800 so so our approximation the people in the line is␣
 ↪okay
        if prev_cust_wait == 0:
            abandon_prob = 0
        elif prev_cust_wait<(40/60):
```

```
            abandon_prob = abandon_probs["<40"]
        elif prev_cust_wait>=(40/60) and prev_cust_wait<(50/60):
            abandon_prob = abandon_probs[">=40<50"]
        elif prev_cust_wait>=(50/60) and prev_cust_wait<(60/60):
            abandon_prob = abandon_probs[">=50<60"]
        elif prev_cust_wait>=(60/60) and prev_cust_wait<(75/60):
            abandon_prob = abandon_probs[">=60<75"]
        elif prev_cust_wait>=(75/60) and prev_cust_wait<(90/60):
            abandon_prob = abandon_probs[">=75<90"]
        elif prev_cust_wait>=(90/60) and prev_cust_wait<(105/60):
            abandon_prob = abandon_probs[">=90<105"]
        elif prev_cust_wait>=(105/60) and prev_cust_wait<(120/60):
            abandon_prob = abandon_probs[">=105<120"]
        elif prev_cust_wait>=(120/60) and prev_cust_wait<(180/60):
            abandon_prob = abandon_probs[">=120<180"]
        elif prev_cust_wait>=(180/60):
            abandon_prob = abandon_probs[">=180"]

        return abandon_prob


#############################
class Customer_ls():
    empty = ()

    def __init__(self, customer_ls=np.array([])):
        self.customer_ls = np.array(customer_ls)
        self.customer_ls = self.customer_ls[np.argsort([customer.arrival_time␣
 ↪for customer in customer_ls])]
        self.next = None if not customer_ls else self.customer_ls[0]


    def __len__(self):
        return len(self.customer_ls)


    def next_exits(self):
        if len(self)==1:
            next_cust, self.customer_ls = self.customer_ls[0], np.array([])
            self.next = None
        else:
            next_cust, self.customer_ls = self.customer_ls[0], self.
 ↪customer_ls[1:]
            self.next = self.customer_ls[0]
        return next_cust
```

```
    def add_to_sort(self, customer):
        self.customer_ls = np.append(self.customer_ls, customer)
        self.customer_ls = self.customer_ls[np.argsort([customer.arrival_time␣
↪for customer in self.customer_ls])]
        self.next = self.customer_ls[0]


    def add_to_nosort(self, customer):
        self.customer_ls = np.append(self.customer_ls, customer)
        self.next = self.customer_ls[0]


    def sort(self):
        self.customer_ls = self.customer_ls[np.argsort([customer.arrival_time␣
↪for customer in self.customer_ls])]
        self.next = self.customer_ls[0]


#############################
```

[3]:
```
## The cell contains a function to simulate a single day of the queueing system␣
↪##

def one_day_naive(arrival_ls):

    customer_arrivals = [Customer(arr) for arr in arrival_ls]
    customer_arrivals = Customer_ls(customer_arrivals)
    NormalQueue, cust_output = Customer_ls(), Customer_ls()

    switched_11, switched_1 = False, False
    capacity_8, capacity_11, capacity_1 = 136, 170, 204
    train_capacity = capacity_8

    train_finish_time, prev_wait_time = 0, 0
    while len(customer_arrivals) > 0 or len(NormalQueue) > 0:

        next_arr = customer_arrivals.next

        ## FIRST CAR DOESN"T LEAVE TILL FULL ##
        if train_finish_time == 0 and len(NormalQueue)<train_capacity:
            next_arrival = customer_arrivals.next_exits()
            NormalQueue.add_to_nosort(next_arrival)
            if len(NormalQueue)==train_capacity:
                train_finish_time = NormalQueue.customer_ls[-1].arrival_time

        ## ARRIVAL TO SYSTEM ##
```

```
        elif len(customer_arrivals)>0 and (len(NormalQueue)==0 or next_arr.
↪arrival_time<train_finish_time):
            next_arrival = customer_arrivals.next_exits()
            NormalQueue.add_to_nosort(next_arrival)

        ## SEND A TRAIN ##
        else:
            if train_finish_time > 3 and not switched_11:
                train_capacity = capacity_11
                switched_11 = True
            elif train_finish_time > 5 and not switched_1:
                train_capacity = capacity_1
                switched_1 = True

            load_min, load_max = 0.5/60, 1./60
            unload_min, unload_max = 0.25/60, 0.75/60

            load_time = np.random.uniform(load_min, load_max)
            unload_time = np.random.uniform(unload_min, unload_max)
            service_time = load_time + 3./60 + unload_time

            train_max = min(train_capacity,len(NormalQueue))
            train_count = 0
            while train_count < train_max:

                ## PROCESS THE NEXT CUSTOMER FROM NormalQueue ##
                if len(NormalQueue)!=0:
                    next_served = NormalQueue.next_exits()
                    abandon_prob = next_served.abandon_prob(prev_wait_time)

                    ## NO ABANDON ##
                    if np.random.binomial(n=1,p=1-abandon_prob):
                        new_wait_time = max(0,train_finish_time-next_served.
↪arrival_time)
                        next_served.wait_time = new_wait_time
                        cust_output.add_to_nosort(next_served)
                        prev_wait_time = new_wait_time
                        train_count += 1

                    ## ABANDON ##
                    else:
                        next_served.wait_time = -999
                        cust_output.add_to_nosort(next_served)

                ## EXIT IF BOTH QUEUES ARE EMPTY ##
                elif len(NormalQueue) == 0:
                    train_max = train_count
```

```
            train_finish_time = train_finish_time + service_time

    return cust_output
```

[4]:
```
## This cell simulates 30 days of our queuing system, and outputs the day,␣
↪arrival times, and wait times ##

arrival_rate_param = 2880

## Simulating Trials ##
trials = 31
df = pd.DataFrame()
for i in np.arange(1, trials):

    ## Simulating Arrivals ##
    process = PoissonProcess(lam = arrival_rate_param, T = 12.00)
    arrival_ls = process.get_arrival_time()
    customers = one_day_naive(arrival_ls).customer_ls

    ## Creating Data Frame ##
    data = {
        'day' : i*np.ones(len(arrival_ls)).astype(int),
        'arrival' : [customer.arrival_time for customer in customers],
        'wait' : [60*customer.wait_time for customer in customers]}
    df = df.append(pd.DataFrame(data))

## Cleaning the Data Frame ##
df['arrival hour'] = df['arrival'].astype(str).str.split('.').apply(lambda x:␣
↪x[0]).astype(int)
abandoned_df = df[df['wait'] < 0]
df = df[df['wait'] >= 0]
df.head(5)
```

[4]:

	day	arrival	wait	arrival hour
0	1	0.000080	2.625618	7
1	1	0.000333	2.610411	0
2	1	0.001225	2.556926	0
3	1	0.001355	2.549111	0
4	1	0.001633	2.532429	0

[18]:
```
## Creating Output Data Frame ##

day_grouped = df.groupby('day').count()['arrival']
hr_grouped = df.groupby('arrival hour').count()['arrival']/(trials-1)
abandon_percents = 100* (abandoned_df.groupby('day').size() / df.groupby('day').
↪size())
```

```
stats = [np.min, np.mean, np.max]
one = [np.round(stat(df['wait']), 3) for stat in stats]
two = [np.floor(stat(hr_grouped)) for stat in stats]
three = [np.floor(stat(day_grouped)) for stat in stats]
four = [np.round(stat(abandon_percents), 2) for stat in stats]
data = {'' :['Wait Time', '', '', 'Hourly Throughput', '', '', 'Daily␣
↪Throughput', '', '',
             'Abandonment Percentage', '', ''],
        'Statistic' : ['Minimum', 'Average', 'Maximum']*4,
        'Value' : np.append(one, [two, three, four])}
out = pd.DataFrame(data).set_index('')
out
```

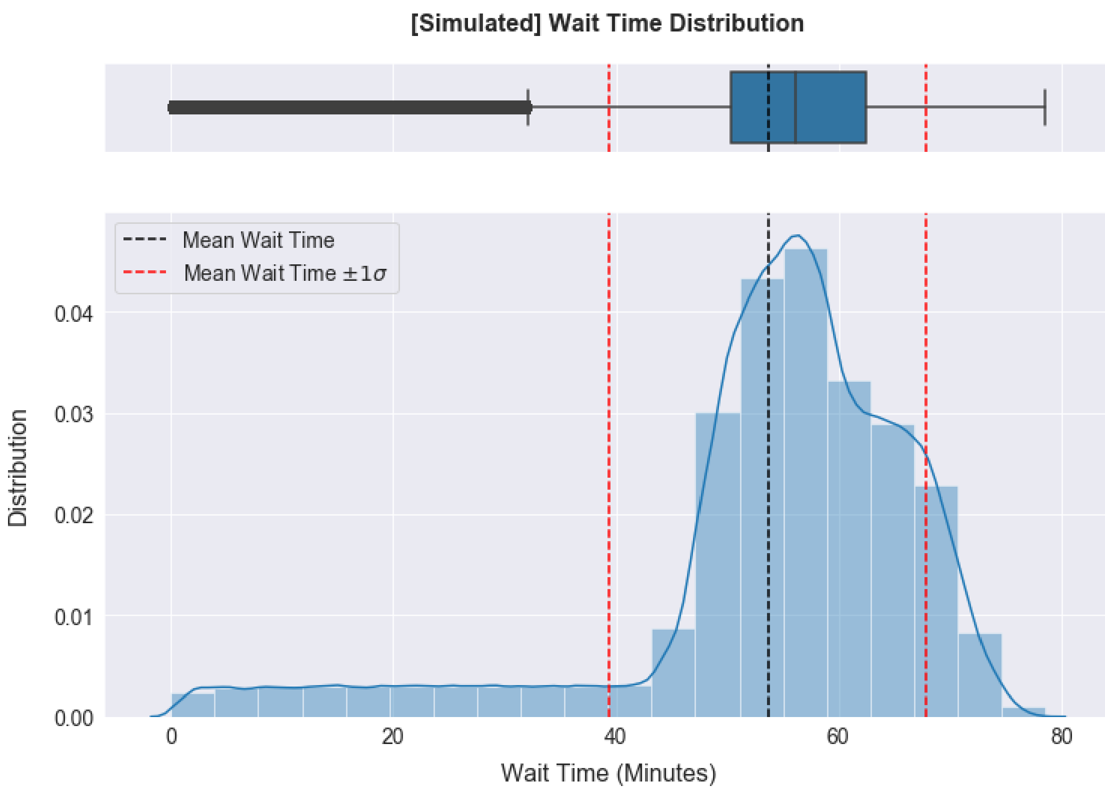
[18]:

	Statistic	Value
Wait Time	Minimum	0.000
	Average	53.638
	Maximum	78.562
Hourly Throughput	Minimum	2634.000
	Average	2755.000
	Maximum	2863.000
Daily Throughput	Minimum	32739.000
	Average	33063.000
	Maximum	33370.000
Abandonment Percentage	Minimum	3.590
	Average	4.440
	Maximum	5.290

[24]:

```
## Plotting Wait Time Distribution ##

sns.set_style('darkgrid')
fig1a, (ax_box, ax_hist) = plt.subplots(2, sharex=True,␣
↪gridspec_kw={"height_ratios": (.15, .85)}, figsize = (12,8))
plt.title('[Simulated] Wait Time Distribution', fontsize = 16, pad = 120,␣
↪fontweight = 'semibold')

avg = df['wait'].mean()
std = df['wait'].std()

sns.distplot(df['wait'], hist = True, bins = 20, kde = True, ax = ax_hist)
sns.boxplot(df['wait'], ax = ax_box)
ax_hist.axvline(avg, c = 'k', linestyle = 'dashed', label = 'Mean Wait Time')
ax_hist.axvline(avg + std, c = 'r', linestyle = 'dashed', label = 'Mean Wait␣
↪Time $ \pm 1 \sigma$')
ax_hist.axvline(avg - std, c = 'r', linestyle = 'dashed')
```

```
ax_box.axvline(avg,       c = 'k', linestyle = 'dashed')
ax_box.axvline(avg + std, c = 'r', linestyle = 'dashed')
ax_box.axvline(avg - std, c = 'r', linestyle = 'dashed')

ax_box.set(xlabel='')

plt.xlabel('Wait Time (Minutes)', fontsize = 16, labelpad = 10)
plt.ylabel('Distribution', fontsize = 16, labelpad = 15)
plt.yticks(fontsize = 14)
plt.xticks(fontsize = 14)
plt.legend(prop={'size': 14})
fig1a.savefig('fig1a.pdf')
```

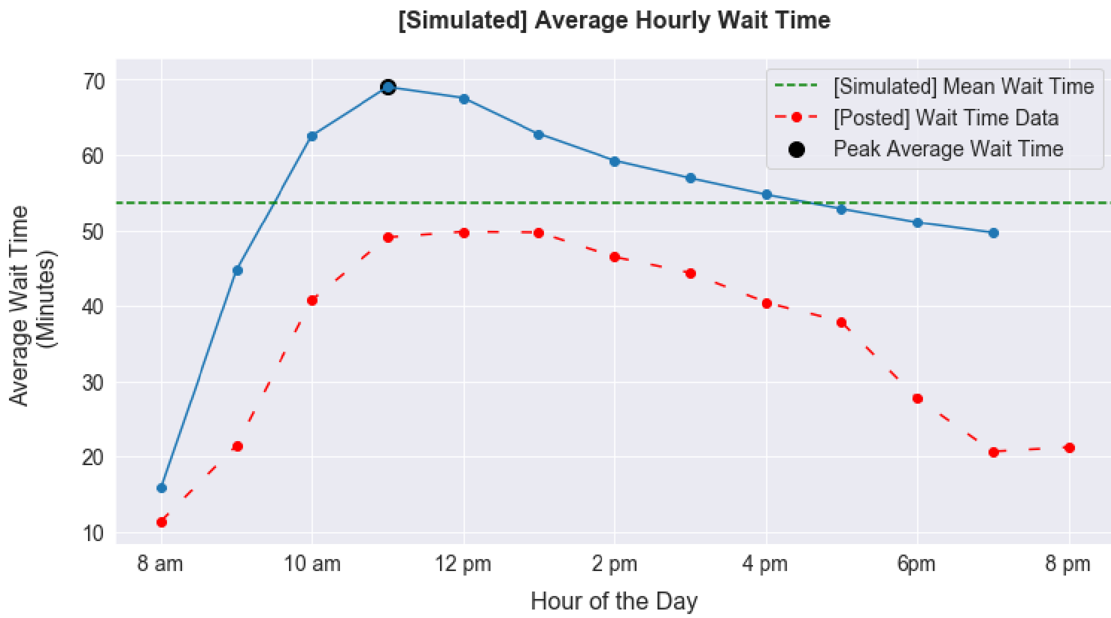


[12]:
```
## All the posted wait time stuff so I can plot it ##
olddf = pd.read_csv('Posted_Wait_Times.csv')
olddf['dt'] = pd.to_datetime(olddf['datetime'])
olddf.drop(['date', 'datetime', 'SACTMIN'], axis = 1, inplace = True)
olddf['date'] = olddf['dt'].dt.date
olddf['time'] = olddf['dt'].dt.time
olddf['hour'] = olddf['dt'].dt.hour
olddf['month'] = olddf['dt'].dt.month
olddf.drop(['date'], axis = 1, inplace = True)
```

```
olddf.rename(columns = {'SPOSTMIN': 'wait'}, inplace = True)
olddf.dropna(inplace = True)
olddf = olddf[olddf['wait'] > 0]
olddf = olddf[['dt', 'month', 'hour', 'time', 'wait']]

# Creating Data Frame for November ##
november = olddf[(olddf['month'] == 11) & (olddf['hour'] >= 8) & (olddf['hour']␣
↪<= 20)].reset_index().drop('index', axis = 1)
november.head(5)

november_grouped = november.groupby('hour').mean()
november_grouped = november_grouped.drop('month', axis = 1).reset_index()
```

[8]:
```
november_grouped = november_grouped.drop('month', axis = 1).reset_index()
```

[22]:
```
## Plotting Average Wait Time vs Time of Day ##
testdf = df.groupby('arrival hour')[['wait']].mean()

sns.set_style('darkgrid')
fig1b, ax1b = plt.subplots(figsize = (12, 6))

#sns.lineplot(data = df, x = 'arrival hour', y = 'wait', ci = 'sd')
plt.plot(testdf.index, testdf['wait'], marker = 'o')
ax1b.axhline(y = np.mean(df['wait']), c = 'g',linestyle = 'dashed', label =␣
↪'[Simulated] Mean Wait Time')
plt.plot(november_grouped.reset_index().index, november_grouped['wait'],␣
↪'o',linestyle = (0, (5,10)), c = 'r', label = '[Posted] Wait Time Data')
plt.scatter(df.groupby('arrival hour').mean()['wait'].argmax(), df.
↪groupby('arrival hour').mean()['wait'].max(), marker = 'o' ,
            s = 100, c = 'k', label = 'Peak Average Wait Time')

plt.title('[Simulated] Average Hourly Wait Time', fontsize = 16, pad = 20,␣
↪fontweight = 'semibold')
plt.xlabel('Hour of the Day', fontsize = 16, labelpad = 10)
plt.ylabel('Average Wait Time \n (Minutes)', fontsize = 16, labelpad = 15)
plt.xticks([0,2,4,6,8,10,12], ['8 am', '10 am', '12 pm', '2 pm', '4 pm', '6pm',␣
↪'8 pm'], fontsize = 14)
plt.yticks(fontsize = 14)
plt.legend(prop={'size': 14})
fig1b.savefig('fig1b.pdf')
```


[Simulated] Average Hourly Wait Time
70
60
50
40
30
20
10
Average Wait Time
(Minutes)
8 am
10 am
12 pm
2 pm
4 pm
6pm
8 pm
Hour of the Day
[Simulated] Mean Wait Time
[Posted] Wait Time Data
Peak Average Wait Time

[ ]:

[ ]:

6.5 Appendix E: Model #2

Here, we import the data generated in Appendix C and run the same analysis on it as Appendix D does.

model_2_final

December 14, 2020

[1]:
```
## Importing Libraries ##
import numpy as np
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import math
from IPython.display import Markdown as md
import warnings
warnings.filterwarnings('ignore')
# %matplotlib inline
from scipy.optimize import minimize
import time
```

[2]:
```
## This cell defines our Poission Process, Customer, and Customer List Classes␣
↪##

#############################
class PoissonProcess():
    def __init__(self, lam, T):
        self.lam = lam
        self.T = T
        self.simulate()


    def simulate(self, method='inter_arrival_time'):
        if method == 'inter_arrival_time':
            N = int(self.lam * self.T * 1.3)
            inter_ls = np.random.exponential(1/self.lam, size=N)
            arrival_time_ls = np.cumsum(inter_ls)
            self.arrival_time_ls = arrival_time_ls[arrival_time_ls <= self.T]
        if method == 'uniformity_property':
            N = np.random.poisson(self.T * self.lam)
            arrival_time_ls = np.random.uniform(0, self.T, size=N)
            self.arrival_time_ls = np.sort(arrival_time_ls)
```

```
    def get_arrival_time(self):
        return self.arrival_time_ls


    def print_parameter(self):
        print('lambda = {}, T = {}'.format(self.lam, self.T))


    def N_t(self, t):
        assert t >= 0
        assert t <= self.T
        if t == 0:
            return 0
        else:
            return np.argmax(self.arrival_time_ls > t)


    def plot_N_t(self, color='r',alpha=1):
        positive_inf = max(self.arrival_time_ls) * 1.2
        negative_inf = - max(self.arrival_time_ls) * 0.1
        n_arrival = len(self.arrival_time_ls)
        x_ls = np.concatenate([[negative_inf, 0], np.repeat(self.
↪arrival_time_ls,2), [positive_inf]])
        y_ls = np.concatenate([[0], np.repeat(np.arange(n_arrival + 1),2)])
        plt.plot(x_ls, y_ls, c=color, alpha=alpha)

#############################
class Customer():
    def __init__(self, arrival_time=0, ctype='normal',wait_time=None):
        self.arrival_time = arrival_time
        self.ctype = ctype
        self.wait_time = wait_time


    def abandon_prob(self,prev_cust_wait):

        abandon_probs = {"<40":.005, ">=40<50":0.015, ">=50<60":0.03, ">=60<75":
↪0.08, ">=75<90":0.1,
                         ">=90<105":0.1, ">=105<120":0.1, ">=120<180":0.1,␣
↪">=180":0.15,
                        }
        ### 170 is <<< 1800 so so our approximation the people in the line is␣
↪okay
        if prev_cust_wait == 0:
            abandon_prob = 0
        elif prev_cust_wait<(40/60):
            abandon_prob = abandon_probs["<40"]
```

```python
        elif prev_cust_wait>=(40/60) and prev_cust_wait<(50/60):
            abandon_prob = abandon_probs[">=40<50"]
        elif prev_cust_wait>=(50/60) and prev_cust_wait<(60/60):
            abandon_prob = abandon_probs[">=50<60"]
        elif prev_cust_wait>=(60/60) and prev_cust_wait<(75/60):
            abandon_prob = abandon_probs[">=60<75"]
        elif prev_cust_wait>=(75/60) and prev_cust_wait<(90/60):
            abandon_prob = abandon_probs[">=75<90"]
        elif prev_cust_wait>=(90/60) and prev_cust_wait<(105/60):
            abandon_prob = abandon_probs[">=90<105"]
        elif prev_cust_wait>=(105/60) and prev_cust_wait<(120/60):
            abandon_prob = abandon_probs[">=105<120"]
        elif prev_cust_wait>=(120/60) and prev_cust_wait<(180/60):
            abandon_prob = abandon_probs[">=120<180"]
        elif prev_cust_wait>=(180/60):
            abandon_prob = abandon_probs[">=180"]

        return abandon_prob


#############################
class Customer_ls():
    empty = ()

    def __init__(self, customer_ls=np.array([])):
        self.customer_ls = np.array(customer_ls)
        self.customer_ls = self.customer_ls[np.argsort([customer.arrival_time␣
 ↪for customer in customer_ls])]
        self.next = None if not customer_ls else self.customer_ls[0]


    def __len__(self):
        return len(self.customer_ls)


    def next_exits(self):
        if len(self)==1:
            next_cust, self.customer_ls = self.customer_ls[0], np.array([])
            self.next = None
        else:
            next_cust, self.customer_ls = self.customer_ls[0], self.
 ↪customer_ls[1:]
            self.next = self.customer_ls[0]
        return next_cust


    def add_to_sort(self, customer):
```

```
        self.customer_ls = np.append(self.customer_ls, customer)
        self.customer_ls = self.customer_ls[np.argsort([customer.arrival_time␣
↪for customer in self.customer_ls])]
        self.next = self.customer_ls[0]


    def add_to_nosort(self, customer):
        self.customer_ls = np.append(self.customer_ls, customer)
        self.next = self.customer_ls[0]


    def sort(self):
        self.customer_ls = self.customer_ls[np.argsort([customer.arrival_time␣
↪for customer in self.customer_ls])]
        self.next = self.customer_ls[0]


##############################
```

[3]:
```
## The cell contains a function to simulate a single day of the queueing system␣
↪##

def one_day_informed(arrival_ls):

    customer_arrivals = [Customer(arr) for arr in arrival_ls]
    customer_arrivals = Customer_ls(customer_arrivals)
    NormalQueue, cust_output = Customer_ls(), Customer_ls()

    switched_11, switched_1 = False, False
    capacity_8, capacity_11, capacity_1 = 136, 170, 204
    train_capacity = capacity_8

    train_finish_time, prev_wait_time = 0, 0
    while len(customer_arrivals) > 0 or len(NormalQueue) > 0:

        next_arr = customer_arrivals.next

        ## FIRST CAR DOESN"T LEAVE TILL FULL ##
        if train_finish_time == 0 and len(NormalQueue)<train_capacity:
            next_arrival = customer_arrivals.next_exits()
            NormalQueue.add_to_nosort(next_arrival)
            if len(NormalQueue)==train_capacity:
                train_finish_time = NormalQueue.customer_ls[-1].arrival_time

        ## ARRIVAL TO SYSTEM ##
        elif len(customer_arrivals)>0 and (len(NormalQueue)==0 or next_arr.
↪arrival_time<train_finish_time):
```

```
            next_arrival = customer_arrivals.next_exits()
            NormalQueue.add_to_nosort(next_arrival)

        ## SEND A TRAIN ##
        else:
            if train_finish_time > 3 and not switched_11:
                train_capacity = capacity_11
                switched_11 = True
            elif train_finish_time > 5 and not switched_1:
                train_capacity = capacity_1
                switched_1 = True

            load_min, load_max = 0.5/60, 1./60
            unload_min, unload_max = 0.25/60, 0.75/60

            load_time = np.random.uniform(load_min, load_max)
            unload_time = np.random.uniform(unload_min, unload_max)
            service_time = load_time + 3./60 + unload_time

            train_max = min(train_capacity,len(NormalQueue))
            train_count = 0
            while train_count < train_max:

                ## PROCESS THE NEXT CUSTOMER FROM NormalQueue ##
                if len(NormalQueue)!=0:
                    next_served = NormalQueue.next_exits()
                    abandon_prob = next_served.abandon_prob(prev_wait_time)

                    ## NO ABANDON ##
                    if np.random.binomial(n=1,p=1-abandon_prob):
                        new_wait_time = max(0,train_finish_time-next_served.
↪arrival_time)
                        next_served.wait_time = new_wait_time
                        cust_output.add_to_nosort(next_served)
                        prev_wait_time = new_wait_time
                        train_count += 1

                    ## ABANDON ##
                    else:
                        next_served.wait_time = -999
                        cust_output.add_to_nosort(next_served)

                ## EXIT IF BOTH QUEUES ARE EMPTY ##
                elif len(NormalQueue) == 0:
                    train_max = train_count

            train_finish_time = train_finish_time + service_time
```

```python
    return cust_output
```

[5]:
```python
## This cell simulates 30 days of our queuing system, and outputs the day,␣
↪arrival times, and wait times ##

## Simulating Trials ##
trials = 30
df = pd.DataFrame()
for i in np.arange(0, trials):

    ## Loading Arrival Data & Simulating 30 days ##
    arrival_ls = np.genfromtxt('data/day' + str(i+1) + '_arrivals.csv')[1:]
    customers = one_day_informed(arrival_ls).customer_ls

    ## Creating Data Frame ##
    data = {
        'day' : i*np.ones(len(arrival_ls)).astype(int),
        'arrival' : [customer.arrival_time for customer in customers],
        'wait' : [60*customer.wait_time for customer in customers]}
    df = df.append(pd.DataFrame(data))

## Cleaning the Data Frame ##
df['arrival hour'] = df['arrival'].astype(str).str.split('.').apply(lambda x:␣
↪x[0]).astype(int)
abandoned_df = df[df['wait'] < 0]
df = df[df['wait'] >= 0]
df.head(5)
```

[5]:

	day	arrival	wait	arrival hour
0	0	0.001	8.28	0
1	0	0.003	8.16	0
2	0	0.004	8.10	0
3	0	0.006	7.98	0
4	0	0.006	7.98	0

[9]:
```python
## Creating Output Data Frame ##

day_grouped = df.groupby('day').count()['arrival']
hr_grouped = df.groupby('arrival hour').count()['arrival']/(trials)
abandon_percents = 100* (abandoned_df.groupby('day').size() / df.groupby('day').
↪size())

stats = [np.min, np.mean, np.max]
one = [np.round(stat(df['wait']), 3) for stat in stats]
two = [np.floor(stat(hr_grouped)) for stat in stats]
three = [np.floor(stat(day_grouped)) for stat in stats]
```

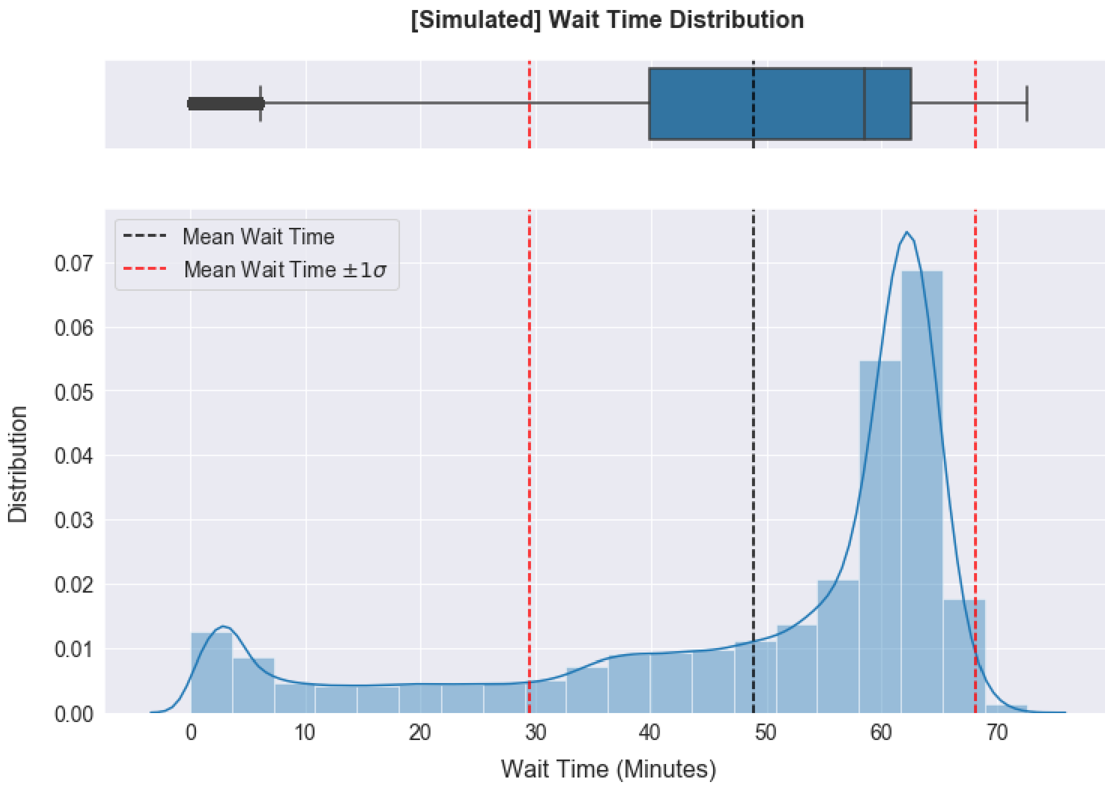
```
four = [np.round(stat(abandon_percents), 2) for stat in stats]
data = {'' :['Wait Time', '', '', 'Hourly Throughput', '', '', 'Daily␣
↪Throughput', '', '',
             'Abandonment Percentage', '', ''],
        'Statistic' : ['Minimum', 'Average', 'Maximum']*4,
        'Value' : np.append(one, [two, three, four])}
out = pd.DataFrame(data).set_index('')
out
```

[9]:

	Statistic	Value
Wait Time	Minimum	0.000
	Average	48.839
	Maximum	72.550
Hourly Throughput	Minimum	1563.000
	Average	2584.000
	Maximum	3087.000
Daily Throughput	Minimum	30776.000
	Average	31008.000
	Maximum	31252.000
Abandonment Percentage	Minimum	3.840
	Average	4.700
	Maximum	5.400

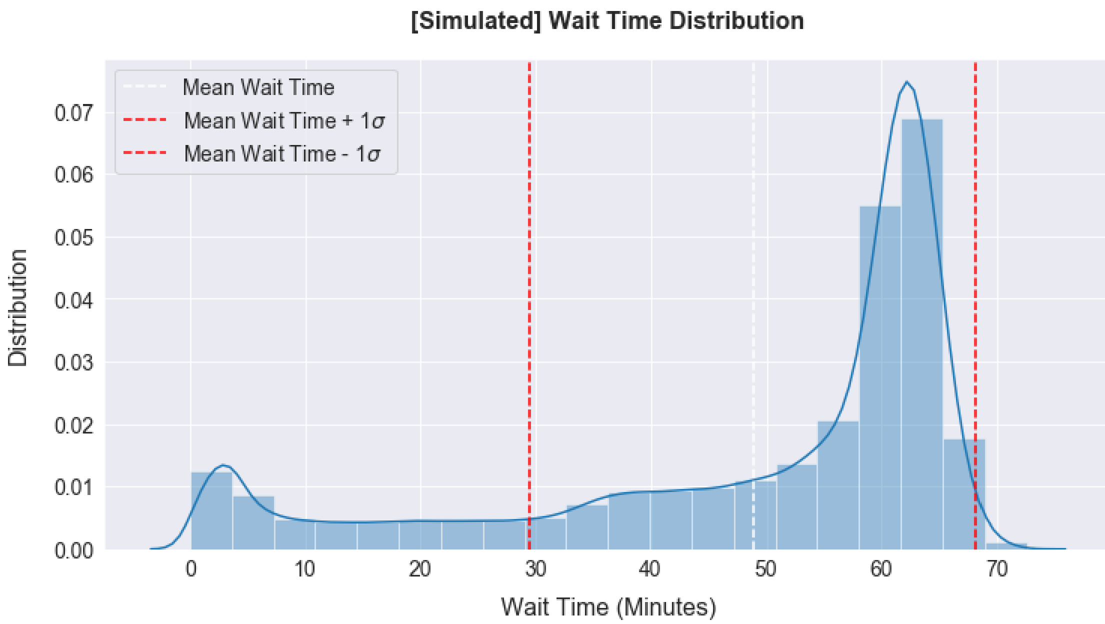
[61]:
```
## Plotting Wait Time Distribution ##

sns.set_style('darkgrid')
fig2a, (ax_box, ax_hist) = plt.subplots(2, sharex=True,␣
↪gridspec_kw={"height_ratios": (.15, .85)}, figsize = (12,8))
plt.title('[Simulated] Wait Time Distribution', fontsize = 16, pad = 120,␣
↪fontweight = 'semibold')

avg = df['wait'].mean()
std = df['wait'].std()

sns.distplot(df['wait'], hist = True, bins = 20, kde = True, ax = ax_hist)
sns.boxplot(df['wait'], ax = ax_box)
ax_hist.axvline(avg, c = 'k', linestyle = 'dashed', label = 'Mean Wait Time')
ax_hist.axvline(avg + std, c = 'r', linestyle = 'dashed', label = 'Mean Wait␣
↪Time $ \pm 1 \sigma$')
ax_hist.axvline(avg - std, c = 'r', linestyle = 'dashed')

ax_box.axvline(avg,       c = 'k', linestyle = 'dashed')
ax_box.axvline(avg + std, c = 'r', linestyle = 'dashed')
ax_box.axvline(avg - std, c = 'r', linestyle = 'dashed')

ax_box.set(xlabel='')
```

```
plt.xlabel('Wait Time (Minutes)', fontsize = 16, labelpad = 10)
plt.ylabel('Distribution', fontsize = 16, labelpad = 15)
plt.yticks(fontsize = 14)
plt.xticks(fontsize = 14)
plt.legend(prop={'size': 14})
fig2a.savefig('fig2a.pdf')
```



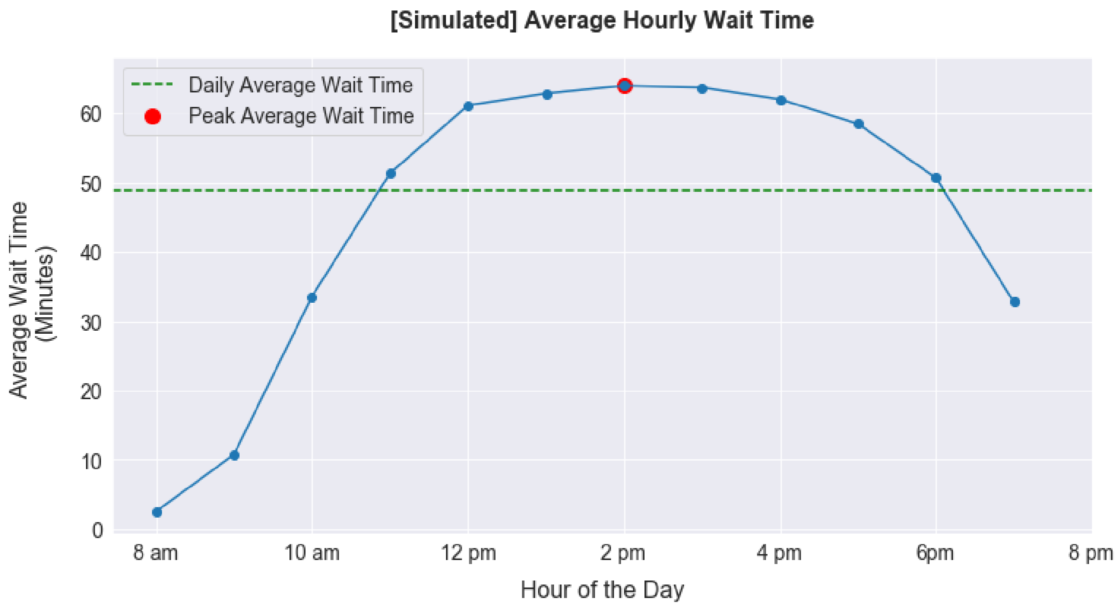
[36]:
```
## Plotting Wait Time Distribution ##

sns.set_style('darkgrid')
fig2a, ax2a = plt.subplots(figsize = (12, 6))

avg = df['wait'].mean()
std = df['wait'].std()

sns.distplot(df['wait'], hist = True, bins = 20, kde = True)
#sns.boxplot(df['wait'])

ax2a.axvline(avg,       c = 'g', linestyle = 'dashed', label = 'Mean Wait Time')
ax2a.axvline(avg + std, c = 'r', linestyle = 'dashed', label = 'Mean Wait Time␣
↪+ 1$\sigma$')
```
85

```
ax2a.axvline(avg - std, c = 'r', linestyle = 'dashed', label = 'Mean Wait Time␣
↪- 1$\sigma$')

plt.title('[Simulated] Wait Time Distribution', fontsize = 16, pad = 20,␣
↪fontweight = 'semibold')
plt.xlabel('Wait Time (Minutes)', fontsize = 16, labelpad = 10)
plt.ylabel('Distribution', fontsize = 16, labelpad = 15)
plt.yticks(fontsize = 14)
plt.xticks(fontsize = 14)
plt.legend(prop={'size': 14})
fig2a.savefig('fig2a.pdf')
```



[11]:
```
#Plotting Average Wait Time vs Time of Day

testdf = df.groupby('arrival hour')[['wait']].mean()

sns.set_style('darkgrid')
fig2b, ax2b = plt.subplots(figsize = (12, 6))

plt.plot(testdf.index, testdf['wait'], marker = 'o')
ax2b.axhline(y = np.mean(df['wait']), c = 'g',linestyle = 'dashed', label =␣
↪'Daily Average Wait Time')
plt.scatter(df.groupby('arrival hour').mean()['wait'].argmax(), df.
↪groupby('arrival hour').mean()['wait'].max(), marker = 'o' ,
            s = 100, c = 'r', label = 'Peak Average Wait Time')
```

```
plt.title('[Simulated] Average Hourly Wait Time', fontsize = 16, pad = 20,␣
↪fontweight = 'semibold')
plt.xlabel('Hour of the Day', fontsize = 16, labelpad = 10)
plt.ylabel('Average Wait Time \n (Minutes)', fontsize = 16, labelpad = 15)
plt.xticks([0,2,4,6,8,10,12], ['8 am', '10 am', '12 pm', '2 pm', '4 pm', '6pm',␣
↪'8 pm'], fontsize = 14)
plt.yticks(fontsize = 14)
plt.legend(prop={'size': 14})
fig2b.savefig('fig2b.pdf')
```




```
[ ]:
```

6.6 Appendix F: Model #3

Building on Appendix E, this iPython notebook implements the Express Queue and plots the various outputs.

model_3_final

December 14, 2020

[2]:
```
## Importing Libraries ##
import numpy as np
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import math
from IPython.display import Markdown as md
import warnings
warnings.filterwarnings('ignore')
# %matplotlib inline
from scipy.optimize import minimize
import time
```

[3]:
```
## This cell defines our Poission Process, Customer, and Customer List Classes␣
↪##

##############################
class PoissonProcess():
    def __init__(self, lam, T):
        self.lam = lam
        self.T = T
        self.simulate()


    def simulate(self, method='inter_arrival_time'):
        if method == 'inter_arrival_time':
            N = int(self.lam * self.T * 1.3)
            inter_ls = np.random.exponential(1/self.lam, size=N)
            arrival_time_ls = np.cumsum(inter_ls)
            self.arrival_time_ls = arrival_time_ls[arrival_time_ls <= self.T]
        if method == 'uniformity_property':
            N = np.random.poisson(self.T * self.lam)
            arrival_time_ls = np.random.uniform(0, self.T, size=N)
            self.arrival_time_ls = np.sort(arrival_time_ls)
```

```
    def get_arrival_time(self):
        return self.arrival_time_ls


    def print_parameter(self):
        print('lambda = {}, T = {}'.format(self.lam, self.T))


    def N_t(self, t):
        assert t >= 0
        assert t <= self.T
        if t == 0:
            return 0
        else:
            return np.argmax(self.arrival_time_ls > t)


    def plot_N_t(self, color='r',alpha=1):
        positive_inf = max(self.arrival_time_ls) * 1.2
        negative_inf = - max(self.arrival_time_ls) * 0.1
        n_arrival = len(self.arrival_time_ls)
        x_ls = np.concatenate([[negative_inf, 0], np.repeat(self.
 ↪arrival_time_ls,2), [positive_inf]])
        y_ls = np.concatenate([[0], np.repeat(np.arange(n_arrival + 1),2)])
        plt.plot(x_ls, y_ls, c=color, alpha=alpha)

#############################
class Customer():
    def __init__(self, arrival_time=0, ctype='normal',wait_time=None):
        self.arrival_time = arrival_time
        self.ctype = ctype
        self.wait_time = wait_time


    def abandon_prob(self,prev_cust_wait):

        abandon_probs = {"<40":.005, ">=40<50":0.015, ">=50<60":0.03, ">=60<75":
 ↪0.08, ">=75<90":0.1,
                         ">=90<105":0.1, ">=105<120":0.1, ">=120<180":0.1,␣
 ↪">=180":0.15,
                        }
        ### 170 is <<< 1800 so so our approximation the people in the line is␣
 ↪okay
        if prev_cust_wait == 0:
            abandon_prob = 0
        elif prev_cust_wait<(40/60):
            abandon_prob = abandon_probs["<40"]
```

```
        elif prev_cust_wait>=(40/60) and prev_cust_wait<(50/60):
            abandon_prob = abandon_probs[">=40<50"]
        elif prev_cust_wait>=(50/60) and prev_cust_wait<(60/60):
            abandon_prob = abandon_probs[">=50<60"]
        elif prev_cust_wait>=(60/60) and prev_cust_wait<(75/60):
            abandon_prob = abandon_probs[">=60<75"]
        elif prev_cust_wait>=(75/60) and prev_cust_wait<(90/60):
            abandon_prob = abandon_probs[">=75<90"]
        elif prev_cust_wait>=(90/60) and prev_cust_wait<(105/60):
            abandon_prob = abandon_probs[">=90<105"]
        elif prev_cust_wait>=(105/60) and prev_cust_wait<(120/60):
            abandon_prob = abandon_probs[">=105<120"]
        elif prev_cust_wait>=(120/60) and prev_cust_wait<(180/60):
            abandon_prob = abandon_probs[">=120<180"]
        elif prev_cust_wait>=(180/60):
            abandon_prob = abandon_probs[">=180"]

        return abandon_prob


#############################
class Customer_ls():
    empty = ()

    def __init__(self, customer_ls=np.array([])):
        self.customer_ls = np.array(customer_ls)
        self.customer_ls = self.customer_ls[np.argsort([customer.arrival_time␣
↪for customer in customer_ls])]
        self.next = None if not customer_ls else self.customer_ls[0]


    def __len__(self):
        return len(self.customer_ls)


    def next_exits(self):
        if len(self)==1:
            next_cust, self.customer_ls = self.customer_ls[0], np.array([])
            self.next = None
        else:
            next_cust, self.customer_ls = self.customer_ls[0], self.
↪customer_ls[1:]
            self.next = self.customer_ls[0]
        return next_cust


    def add_to_sort(self, customer):
```

```
        self.customer_ls = np.append(self.customer_ls, customer)
        self.customer_ls = self.customer_ls[np.argsort([customer.arrival_time␣
↪for customer in self.customer_ls])]
        self.next = self.customer_ls[0]


    def add_to_nosort(self, customer):
        self.customer_ls = np.append(self.customer_ls, customer)
        self.next = self.customer_ls[0]


    def sort(self):
        self.customer_ls = self.customer_ls[np.argsort([customer.arrival_time␣
↪for customer in self.customer_ls])]
        self.next = self.customer_ls[0]


#############################
```

0.1 Model #3

[4]:
```
## Loading Speed Factor Function ##
def speed_factor(proportion_express):
    factor = -0.5*proportion_express + 1

        #factor = 1 - 0.8*np.sqrt(proportion_express)
    return factor
```

[5]:
```
## ONE DAY FUNCTION FOR MODEL #3 ##

## The cell contains a function to simulate a single day of the queueing system␣
↪##

def one_day_express(arrival_ls):

    #### FOR TUNING ####
    ExpressPriority = 1.0
    express_prop = 0.15
    express_abandon_prob = 0.15
    ####################

    customer_arrivals = [Customer(arr) for arr in arrival_ls]
    customer_arrivals = Customer_ls(customer_arrivals)
```

```
    NormalQueue, ExpressQueue, cust_output= Customer_ls(), Customer_ls(),␣
↪Customer_ls()

    switched_11, switched_1 = False, False
    capacity_8, capacity_11, capacity_1 = 136, 170, 204
    train_capacity = capacity_8

    train_finish_time, prev_wait_time = 0, 0
    time = 0
    while len(customer_arrivals) > 0 or len(NormalQueue) > 0 or␣
↪len(ExpressQueue) > 0:

        next_arr = customer_arrivals.next

        ## FIRST CAR DOESN"T LEAVE TILL FULL ##
        if train_finish_time == 0 and len(NormalQueue)<train_capacity:
            if np.random.binomial(n=1,p=express_prop):
                new_arrival = customer_arrivals.next_exits()
                new_arrival.arrival_time += np.random.uniform(0.5,1.5)
                new_arrival.ctype = "express"
                customer_arrivals.add_to_sort(new_arrival)
            else:
                new_arrival = customer_arrivals.next_exits()
                NormalQueue.add_to_nosort(new_arrival)

            if len(NormalQueue)==train_capacity:
                train_finish_time = NormalQueue.customer_ls[-1].arrival_time

        ## ARRIVAL TO SYSTEM ##
        elif len(customer_arrivals)>0 and␣
↪(len(NormalQueue)+len(ExpressQueue)==0 or next_arr.
↪arrival_time<train_finish_time):

            if next_arr.arrival_time>time:
                customer_arrivals.sort()
                time+=0.4
                next_arr = customer_arrivals.next

            ## EXPRESS ARRIVAL
            if next_arr.ctype == "express":
                new_arrival = customer_arrivals.next_exits()
                ExpressQueue.add_to_nosort(new_arrival)

            ## NORMAL ARRIVAL ##
            elif next_arr.ctype == "normal":
                if next_arr.arrival_time <= 10.5 and np.random.
↪binomial(n=1,p=express_prop):
```

```
                new_arrival = customer_arrivals.next_exits()
                new_arrival.arrival_time += np.random.uniform(0.5,1.5)
                new_arrival.ctype = "express"
                customer_arrivals.add_to_nosort(new_arrival)
            else:
                new_arrival = customer_arrivals.next_exits()
                NormalQueue.add_to_nosort(new_arrival)

    ## SEND A TRAIN ##
    else:
        if train_finish_time > 3 and not switched_11:
            train_capacity = capacity_11
            switched_11 = True
        elif train_finish_time > 5 and not switched_1:
            train_capacity = capacity_1
            switched_1 = True

        train_max = min(train_capacity,len(NormalQueue)+len(ExpressQueue))
        rider_types = np.array([])

        train_count = 0
        last_abandon = False
        load_next = np.random.
↪choice(['express','normal'],p=[ExpressPriority,1-ExpressPriority])
        while train_count < train_max:

            ## EXIT IF BOTH QUEUES ARE EMPTY ##
            if len(ExpressQueue)==0 and len(NormalQueue)==0:
                train_max = train_count

            ## PROCESS THE NEXT CUSTOMER FROM EITHER EXPRESS OR NORMAL ##
            else:
                load_next = np.random.
↪choice(['express','normal'],p=[ExpressPriority,1-ExpressPriority]) if␣
↪not(last_abandon) else load_next

                ## NEXT IS EXPRESS ##
                if len(ExpressQueue)!=0 and load_next=='express':
                    next_served = ExpressQueue.next_exits()
                    abandon_prob = express_abandon_prob

                ## NEXT IS NORMAL ##
                elif (len(ExpressQueue)==0 and len(NormalQueue)>0) or␣
↪(len(NormalQueue)!=0 and load_next=='normal'):
                    next_served = NormalQueue.next_exits()
                    abandon_prob = next_served.abandon_prob(prev_wait_time)
```

```
                    ## NO ABANDON ##
                    if np.random.binomial(n=1,p=1-abandon_prob):
                        new_wait_time = max(0,train_finish_time-next_served.
↪arrival_time)
                        next_served.wait_time = new_wait_time
                        cust_output.add_to_nosort(next_served)
                        prev_wait_time = new_wait_time
                        train_count += 1
                        last_abandon=False
                        rider_types = np.append(rider_types, next_served.ctype)

                    ## ABANDON ##
                    else:
                        next_served.wait_time = -999
                        cust_output.add_to_nosort(next_served)
                        last_abandon=True



            ## SPEED FACTOR AND SERVICE TIME ##
            proportion_express = 0 if len(rider_types)==0 else np.
↪count_nonzero(rider_types == 'express') / len(rider_types)
            factor = speed_factor(proportion_express)

            load_min, load_max = 0.5/60, 1./60
            unload_min, unload_max = 0.25/60, 0.75/60

            load_time = factor * np.random.uniform(load_min, load_max)
            unload_time = np.random.uniform(unload_min, unload_max)
            service_time = load_time + 3./60 + unload_time

            train_finish_time = train_finish_time + service_time

    return cust_output
```

[117]:
```
## SIMULATING w/ MODEL #3 AND OUTPUT TABLE ##

# Simulating Trials
trials = 30
df = pd.DataFrame()
for i in np.arange(0, trials):

    # Loading Arrival Data & Simulating 30 days
    arrival_ls = np.genfromtxt('data/day' + str(i+1) + '_arrivals.csv')[1:]
    customers = one_day_express(arrival_ls).customer_ls

    # Creating Data Frame
```

```
    data = {
        'day' : i*np.ones(len(arrival_ls)).astype(int),
        'arrival' : [customer.arrival_time for customer in customers],
        'wait' : [60*customer.wait_time for customer in customers],
    'Customer Type' : [customer.ctype for customer in customers]}
    df = df.append(pd.DataFrame(data))

# Cleaning the Data Frame
df['arrival hour'] = df['arrival'].astype(str).str.split('.').apply(lambda x:␣
 ↪x[0]).astype(int)
abandoned_df = df[df['wait'] < 0]
df = df[df['wait'] >= 0]
df = df.replace( {'normal' : 'Standard',
                  'express' : 'Express'})
```

[118]:
```
df
```

[118]:
```
       day  arrival       wait Customer Type  arrival hour
0        0    0.006  10.260000      Standard             0
1        0    0.006  10.260000      Standard             0
2        0    0.007  10.200000      Standard             0
3        0    0.009  10.080000      Standard             0
4        0    0.012   9.900000      Standard             0
…      …        …          …             …             …
32519   29   11.998  10.870627      Standard            11
32520   29   11.998  10.870627      Standard            11
32521   29   11.998  10.870627      Standard            11
32522   29   11.998  10.870627      Standard            11
32523   29   11.999  10.810627      Standard            11

[927337 rows x 5 columns]
```

[120]:
```
df.groupby('arrival hour').count()['arrival']
```

[120]:
```
arrival hour
0      40416
1      68870
2      85365
3      90478
4      90701
5      88664
6      87136
7      86121
8      84359
9      79789
10     73695
11     51743
```

```
Name: arrival, dtype: int64
```

[125]:
```
#Output Table for Model #3

# Separating the DF into separate Data Frames for Standard & Express
standard_df = df[df['Customer Type'] == 'Standard']
express_df = df[df['Customer Type'] == 'Express']
standard_abandoned_df = abandoned_df[abandoned_df['Customer Type'] ==␣
↪'Standard']
express_abandoned_df = abandoned_df[abandoned_df['Customer Type'] == 'Express']

# Aggregate Groupings
day_grouped = df.groupby('day').count()['arrival']
hr_grouped = df.groupby('arrival hour').count()['arrival']
abandon_percents = 100* (abandoned_df.groupby('day').size() / df.groupby('day').
↪size())

# Standard Groupings
standard_day_grouped = standard_df.groupby('day').count()['arrival']
standard_hr_grouped = standard_df.groupby('arrival hour').count()['arrival']
standard_abandon_percents = 100* (standard_abandoned_df.groupby('day').size() /␣
↪standard_df.groupby('day').size())

# Express Groupings
express_day_grouped = express_df.groupby('day').count()['arrival']
express_hr_grouped = express_df.groupby('arrival hour').count()['arrival']
express_abandon_percents = 100* (express_abandoned_df.groupby('day').size() /␣
↪express_df.groupby('day').size())

stats = [np.min, np.mean, np.max]

# Aggregate Columns
agg_one = [np.round(stat(df['wait']), 3) for stat in stats]
agg_two = [np.floor(stat(hr_grouped)) for stat in stats]
agg_three = [np.floor(stat(day_grouped)) for stat in stats]
agg_four = [np.round(stat(abandon_percents), 2) for stat in stats]

# Standard Columns
standard_one = [np.round(stat(standard_df['wait']), 3) for stat in stats]
standard_two = [np.floor(stat(standard_hr_grouped)) for stat in stats]
standard_three = [np.floor(stat(standard_day_grouped)) for stat in stats]
standard_four = [np.round(stat(standard_abandon_percents), 2) for stat in stats]

# Express Columns
express_one = [np.round(stat(express_df['wait']), 3) for stat in stats]
express_two = [np.floor(stat(express_hr_grouped)) for stat in stats]
express_three = [np.floor(stat(express_day_grouped)) for stat in stats]
```

```
express_four = [np.round(stat(express_abandon_percents), 2) for stat in stats]

data = {'' :['Wait Time', '', '', 'Hourly Throughput', '', '', 'Daily␣
↪Throughput', '', '',
              'Abandonment Percentage', '', ''],
        'Statistic' : ['Minimum', 'Average', 'Maximum']*4,
        'Standard' : np.append(standard_one, [standard_two, standard_three,␣
↪standard_four]),
        'Express' : np.append(express_one, [express_two, express_three,␣
↪express_four]),
        'Aggregate Value' : np.append(agg_one, [agg_two, agg_three, agg_four])}
out = pd.DataFrame(data).set_index('')
out
```

[125]:

	Statistic	Standard	Express	Aggregate Value
Wait Time	Minimum	0.000	0.000	0.000
	Average	45.200	2.106	39.897
	Maximum	67.312	4.658	67.312
Hourly Throughput	Minimum	39876.000	540.000	40416.000
	Average	67768.000	9509.000	77278.000
	Maximum	79150.000	12308.000	90701.000
Daily Throughput	Minimum	26842.000	3658.000	30588.000
	Average	27107.000	3803.000	30911.000
	Maximum	27355.000	4021.000	31187.000
Abandonment Percentage	Minimum	NaN	NaN	4.060
	Average	NaN	NaN	5.030
	Maximum	NaN	NaN	6.020

0.2 Model #2

[126]:

```
## ONE DAY FUNCTION FOR MODEL #2 (to compare these values to Model #3) ##

# The cell contains a function to simulate a single day of the queueing system.

def one_day_informed(arrival_ls):



    customer_arrivals = [Customer(arr) for arr in arrival_ls]
    customer_arrivals = Customer_ls(customer_arrivals)
    num_abandoned = 0


    switched_11 = False
```

```
    switched_1 = False
    capacity_8 = 136
    capacity_11 = 170
    capacity_1 = 204
    train_capacity = capacity_8

    NormalQueue = Customer_ls()
    train_finish_time = 0
    cust_output = Customer_ls()
    prev_wait_time = 0

    while len(customer_arrivals) > 0 or len(NormalQueue) > 0:
        next_arr = customer_arrivals.next
        #FIRST CAR DOESN"T LEAVE TILL FULL
        if train_finish_time == 0 and len(NormalQueue)<train_capacity:
            next_arrival = customer_arrivals.next_exits()
            NormalQueue.add_to_nosort(next_arrival)
            if len(NormalQueue)==train_capacity:
                train_finish_time = NormalQueue.customer_ls[-1].arrival_time


        #ARRIVAL TO SYSTEM
        elif len(customer_arrivals)>0 and (len(NormalQueue)==0 or next_arr.
↪arrival_time<train_finish_time):
            next_arrival = customer_arrivals.next_exits()
            NormalQueue.add_to_nosort(next_arrival)

        #SEND A TRAIN
        else:
            if train_finish_time > 3 and not switched_11:
                train_capacity = capacity_11
                switched_11 = True
            elif train_finish_time > 5 and not switched_1:
                train_capacity = capacity_1
                switched_1 = True





            load_min = 0.5/60
            load_max = 1/60

            unload_min = 0.25/60
            unload_max = .75/60

            load_time = np.random.uniform(load_min, load_max)
```

```
            unload_time = np.random.uniform(unload_min, unload_max)

            service_time = load_time + 3./60 + unload_time


            train_max = min(train_capacity,len(NormalQueue))
            train_count = 0
            while train_count < train_max:
                if len(NormalQueue)!=0:
                    next_served = NormalQueue.next_exits()
                    abandon_prob = next_served.abandon_prob(prev_wait_time)
                    if np.random.binomial(n=1,p=1-abandon_prob):
                        new_wait_time = max(0,train_finish_time-next_served.
↪arrival_time)
                        next_served.wait_time = new_wait_time
                        cust_output.add_to_nosort(next_served)
                        prev_wait_time = new_wait_time
                        train_count += 1
                    else:
                        num_abandoned+=1
                        next_served.wait_time = -999
                        cust_output.add_to_nosort(next_served)
                else:
                    train_max = train_count

            train_finish_time = train_finish_time + service_time

    return cust_output
```

[127]:
```
## SIMULATING 5 DAYS w/ MODEL #2 AND OUTPUT TABLE ##

# Simulating Trials
trials = 30

df2 = pd.DataFrame()
for i in np.arange(0, trials):

    # Loading Arrival Data & Simulating 30 days
    arrival_ls2 = np.genfromtxt('data/day' + str(i+1) + '_arrivals.csv')[1:]
    customers2 = one_day_informed(arrival_ls2).customer_ls

    # Creating Data Frame
    data2 = {
        'day' : i*np.ones(len(arrival_ls2)).astype(int),
        'arrival' : [customer2.arrival_time for customer2 in customers2],
        'wait' : [60*customer2.wait_time for customer2 in customers2]}
    df2 = df2.append(pd.DataFrame(data2))
```

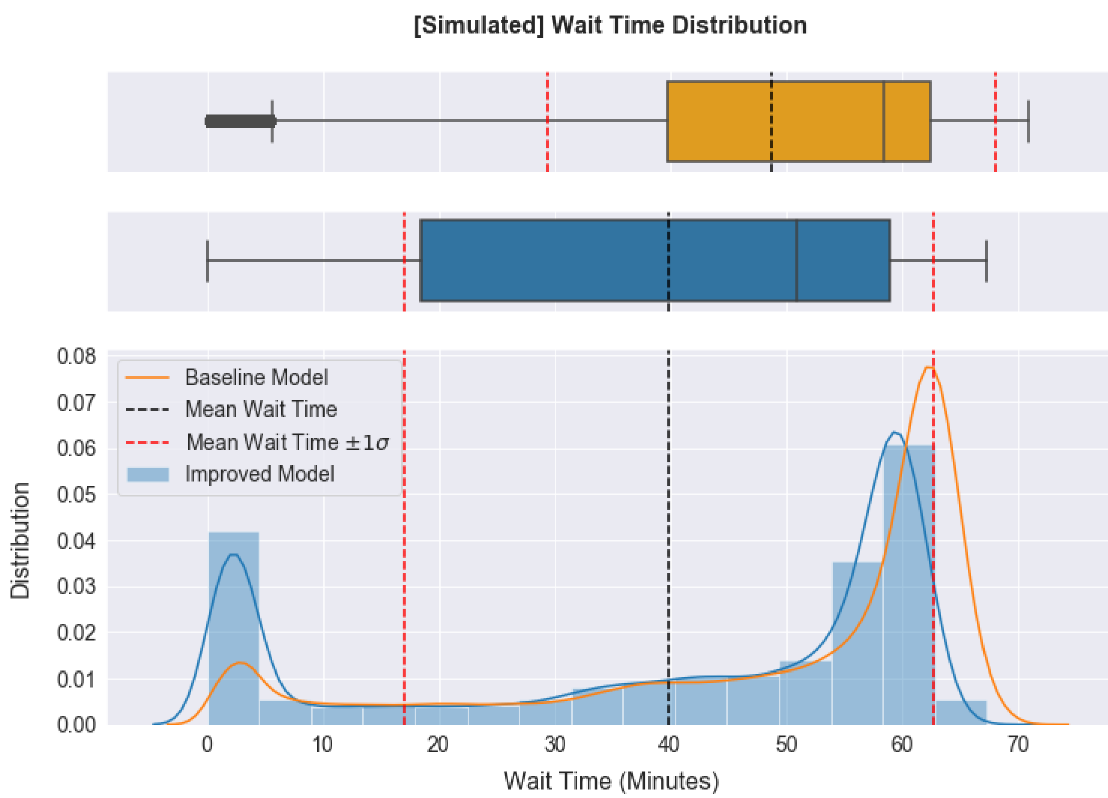
```
# Cleaning the Data Frame
df2['arrival hour'] = df2['arrival'].astype(str).str.split('.').apply(lambda x:␣
↪x[0]).astype(int)
abandoned_df2 = df2[df2['wait'] < 0]
df2 = df2[df2['wait'] >= 0]


day_grouped = df2.groupby('day').count()['arrival']
hr_grouped = df2.groupby('arrival hour').count()['arrival']/ (trials - 1)
abandon_percents = 100* (abandoned_df2.groupby('day').size() / df2.
↪groupby('day').size())

stats = [np.min, np.mean, np.max]
one = [np.round(stat(df2['wait']), 3) for stat in stats]
two = [np.floor(stat(hr_grouped)) for stat in stats]
three = [np.floor(stat(day_grouped)) for stat in stats]
four = [np.round(stat(abandon_percents), 2) for stat in stats]
data = {'' :['Wait Time', '', '', 'Hourly Throughput', '', '', 'Daily␣
↪Throughput', '', '',
             'Abandonment Percentage', '', ''],
        'Statistic' : ['Minimum', 'Average', 'Maximum']*4,
        'Value' : np.append(one, [two, three, four])}
out2 = pd.DataFrame(data).set_index('')
out2
```

[127]:

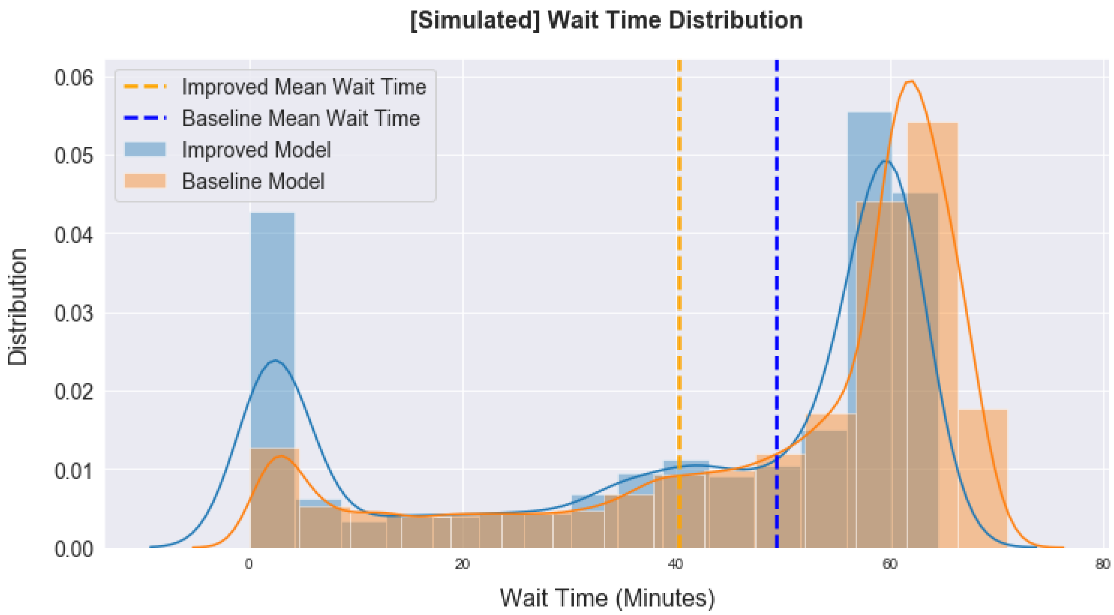
	Statistic	Value
Wait Time	Minimum	0.000
	Average	48.733
	Maximum	70.935
Hourly Throughput	Minimum	1618.000
	Average	2673.000
	Maximum	3194.000
Daily Throughput	Minimum	30758.000
	Average	31011.000
	Maximum	31230.000
Abandonment Percentage	Minimum	4.160
	Average	4.690
	Maximum	5.390

0.3 Wait Times Plotted Together

[128]:
```
## Plotting Wait Time Distribution ##

sns.set_style('darkgrid')
fig3a, (ax_box_base, ax_box_new, ax_hist) = plt.subplots(3, sharex=True,␣
↪gridspec_kw={"height_ratios": (.175,.175, .65)}, figsize = (12,8))
plt.title('[Simulated] Wait Time Distribution', fontsize = 16, pad = 210,␣
↪fontweight = 'semibold')

avg = df['wait'].mean()
std = df['wait'].std()

avg2 = df2['wait'].mean()
std2 = df2['wait'].std()

sns.distplot(df['wait'], hist = True, bins = 15, kde = True, label = 'Improved␣
↪Model', ax = ax_hist)
sns.distplot(df2['wait'], hist = False, bins = 15, kde = True, label =␣
↪'Baseline Model', ax = ax_hist)

sns.boxplot(df['wait'], ax = ax_box_new)
sns.boxplot([df2['wait']], ax = ax_box_base, color = 'orange', labels =␣
↪['Baseline Model'])

ax_box_base.set(xlabel='')
ax_box_new.set(xlabel='')

ax_hist.axvline(avg, c = 'k', linestyle = 'dashed', label = 'Mean Wait Time')
ax_hist.axvline(avg + std, c = 'r', linestyle = 'dashed', label = 'Mean Wait␣
↪Time $ \pm 1 \sigma$')
ax_hist.axvline(avg - std, c = 'r', linestyle = 'dashed')
ax_box_base.axvline(avg2,       c = 'k', linestyle = 'dashed')
ax_box_base.axvline(avg2 + std2, c = 'r', linestyle = 'dashed')
ax_box_base.axvline(avg2 - std2, c = 'r', linestyle = 'dashed')
ax_box_new.axvline(avg,       c = 'k', linestyle = 'dashed')
ax_box_new.axvline(avg + std, c = 'r', linestyle = 'dashed')
ax_box_new.axvline(avg - std, c = 'r', linestyle = 'dashed')

plt.xlabel('Wait Time (Minutes)', fontsize = 16, labelpad = 10)
plt.ylabel('Distribution', fontsize = 16, labelpad = 15)
plt.yticks(fontsize = 14)
plt.xticks(fontsize = 14)
plt.legend(prop={'size': 14})
fig3a.savefig('fig3a.pdf')
```



[10]:
```
## Plotting Wait Time Distribution ##

sns.set_style('darkgrid')
fig3a, ax3a = plt.subplots(figsize = (12, 6))

sns.distplot(df['wait'], hist = True, bins = 15, kde = True, label = 'Improved␣
↪Model')
sns.distplot(df2['wait'], hist = True, bins = 15, kde = True, label = 'Baseline␣
↪Model')
ax3a.axvline(df['wait'].mean(), c = 'orange',linestyle = 'dashed', lw = 2.5,␣
↪label = 'Improved Mean Wait Time')
ax3a.axvline(df2['wait'].mean(), c = 'blue',linestyle = 'dashed', lw = 2.5,␣
↪label = 'Baseline Mean Wait Time')

plt.title('[Simulated] Wait Time Distribution', fontsize = 16, pad = 20,␣
↪fontweight = 'semibold')
plt.xlabel('Wait Time (Minutes)', fontsize = 16, labelpad = 10)
plt.ylabel('Distribution', fontsize = 16, labelpad = 15)
plt.yticks(fontsize = 14);
plt.legend(prop={'size': 14})
fig3a.savefig('fig3a.pdf')
```

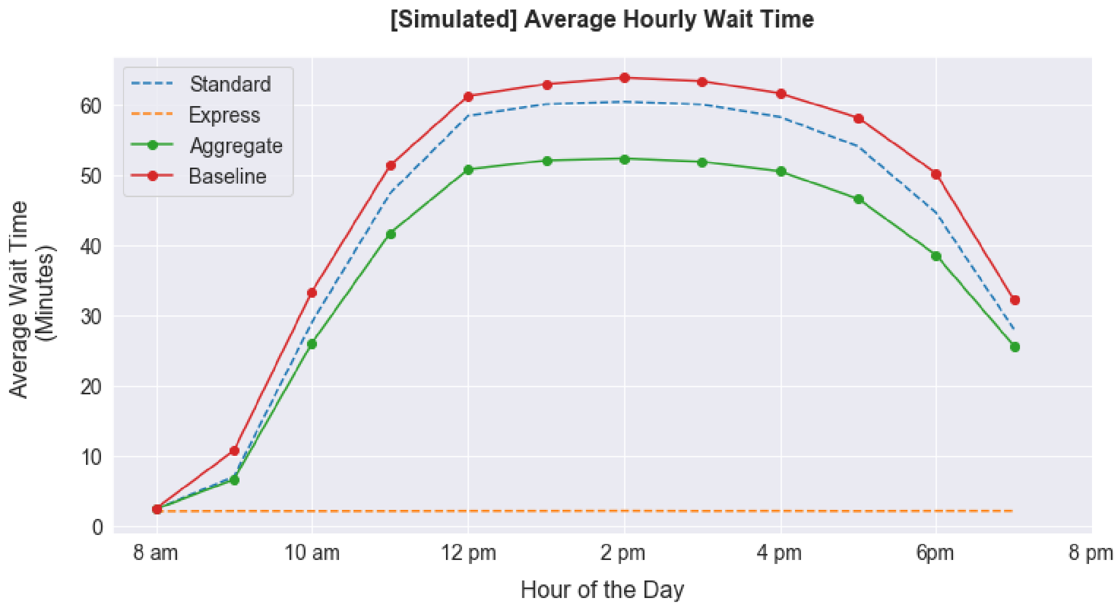


0.4 Average Wait Times w/ Model #2 Baseline

[129]:
```
## Plotting Average Wait Time vs Time of Day ##

standard_df2 = standard_df.groupby('arrival hour')[['wait']].mean()
express_df2 = express_df.groupby('arrival hour')[['wait']].mean()
agg_df2 = df.groupby('arrival hour')[['wait']].mean()
df2_plot = df2.groupby('arrival hour')[['wait']].mean()

sns.set_style('darkgrid')
fig3b, ax3b = plt.subplots(figsize = (12, 6))

plt.plot(standard_df2.index, standard_df2['wait'], linestyle = 'dashed', label␣
 ↪= 'Standard')
plt.plot(express_df2.index, express_df2['wait'], linestyle = 'dashed', label =␣
 ↪'Express')
plt.plot(agg_df2.index, agg_df2['wait'], marker = 'o', label = 'Aggregate')
plt.plot(df2_plot.index, df2_plot['wait'], marker = 'o', label = 'Baseline')
#plt.scatter(df.groupby('arrival hour').mean()['wait'].argmax(), df.
 ↪groupby('arrival hour').mean()['wait'].max(), marker = 'o' , s = 100, c =␣
 ↪'r', label = 'Peak Aggregate Wait Time')

plt.title('[Simulated] Average Hourly Wait Time', fontsize = 16, pad = 20,␣
 ↪fontweight = 'semibold')
plt.xlabel('Hour of the Day', fontsize = 16, labelpad = 10)
```

104

```
plt.ylabel('Average Wait Time \n (Minutes)', fontsize = 16, labelpad = 15)
plt.xticks([0,2,4,6,8,10,12], ['8 am', '10 am', '12 pm', '2 pm', '4 pm', '6pm',␣
↪'8 pm'], fontsize = 14)
plt.yticks(fontsize = 14)
plt.legend(prop={'size': 14}, loc = 'upper left')
fig3b.savefig('fig3b.pdf')
```

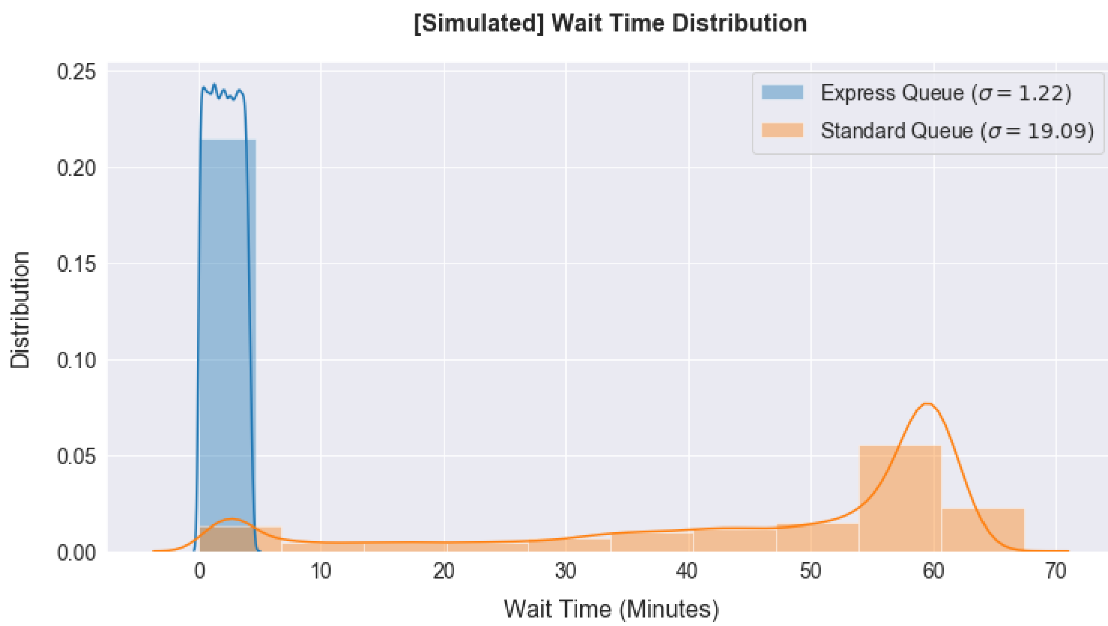


[135]:
```
print(np.std(express_df['wait']))
print(np.std(standard_df['wait']))
```

```
1.2221985353332214
19.085775123828753
```

[139]:
```
sns.set_style('darkgrid')
fig3c, ax3c = plt.subplots(figsize = (12, 6))

sns.distplot(express_df['wait'], bins = 1, label = 'Express Queue ($\sigma = 1.
↪22$)')
sns.distplot(standard_df['wait'], bins = 10, label = 'Standard Queue ($\sigma =␣
↪19.09$)')


plt.title('[Simulated] Wait Time Distribution', fontsize = 16, pad = 20,␣
↪fontweight = 'semibold')
plt.xlabel('Wait Time (Minutes)', fontsize = 16, labelpad = 10)
plt.ylabel('Distribution', fontsize = 16, labelpad = 15)
plt.yticks(fontsize = 14)
```

```
plt.xticks(fontsize = 14)
plt.legend(prop={'size': 14})
fig3c.savefig('fig3c.pdf')
```



[ ]:

6.7 Appendix G: Model #4 - Part I

This is the first appendix of our tuning and sensitivity analysis. It performs the analysis on the system assumptions and opening probability parameter.

model_4a_final

December 14, 2020

[20]:
```
## Importing Libraries ##
import numpy as np
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import math
from IPython.display import Markdown as md
import warnings
warnings.filterwarnings('ignore')
# %matplotlib inline
from scipy.optimize import minimize
import time
```

[21]:
```
## This cell defines our Poission Process, Customer, and Customer List Classes␣
↪##

##############################
class PoissonProcess():
    def __init__(self, lam, T):
        self.lam = lam
        self.T = T
        self.simulate()


    def simulate(self, method='inter_arrival_time'):
        if method == 'inter_arrival_time':
            N = int(self.lam * self.T * 1.3)
            inter_ls = np.random.exponential(1/self.lam, size=N)
            arrival_time_ls = np.cumsum(inter_ls)
            self.arrival_time_ls = arrival_time_ls[arrival_time_ls <= self.T]
        if method == 'uniformity_property':
            N = np.random.poisson(self.T * self.lam)
            arrival_time_ls = np.random.uniform(0, self.T, size=N)
            self.arrival_time_ls = np.sort(arrival_time_ls)
```

```
    def get_arrival_time(self):
        return self.arrival_time_ls


    def print_parameter(self):
        print('lambda = {}, T = {}'.format(self.lam, self.T))


    def N_t(self, t):
        assert t >= 0
        assert t <= self.T
        if t == 0:
            return 0
        else:
            return np.argmax(self.arrival_time_ls > t)


    def plot_N_t(self, color='r',alpha=1):
        positive_inf = max(self.arrival_time_ls) * 1.2
        negative_inf = - max(self.arrival_time_ls) * 0.1
        n_arrival = len(self.arrival_time_ls)
        x_ls = np.concatenate([[negative_inf, 0], np.repeat(self.
↪arrival_time_ls,2), [positive_inf]])
        y_ls = np.concatenate([[0], np.repeat(np.arange(n_arrival + 1),2)])
        plt.plot(x_ls, y_ls, c=color, alpha=alpha)

#############################
class Customer():
    def __init__(self, arrival_time=0, ctype='normal',wait_time=None):
        self.arrival_time = arrival_time
        self.ctype = ctype
        self.wait_time = wait_time


    def abandon_prob(self,prev_cust_wait):

        abandon_probs = {"<40":.005, ">=40<50":0.015, ">=50<60":0.03, ">=60<75":
↪0.08, ">=75<90":0.1,
                         ">=90<105":0.1, ">=105<120":0.1, ">=120<180":0.1,␣
↪">=180":0.15,
                        }
        ### 170 is <<< 1800 so so our approximation the people in the line is␣
↪okay
        if prev_cust_wait == 0:
            abandon_prob = 0
        elif prev_cust_wait<(40/60):
            abandon_prob = abandon_probs["<40"]
```

```
        elif prev_cust_wait>=(40/60) and prev_cust_wait<(50/60):
            abandon_prob = abandon_probs[">=40<50"]
        elif prev_cust_wait>=(50/60) and prev_cust_wait<(60/60):
            abandon_prob = abandon_probs[">=50<60"]
        elif prev_cust_wait>=(60/60) and prev_cust_wait<(75/60):
            abandon_prob = abandon_probs[">=60<75"]
        elif prev_cust_wait>=(75/60) and prev_cust_wait<(90/60):
            abandon_prob = abandon_probs[">=75<90"]
        elif prev_cust_wait>=(90/60) and prev_cust_wait<(105/60):
            abandon_prob = abandon_probs[">=90<105"]
        elif prev_cust_wait>=(105/60) and prev_cust_wait<(120/60):
            abandon_prob = abandon_probs[">=105<120"]
        elif prev_cust_wait>=(120/60) and prev_cust_wait<(180/60):
            abandon_prob = abandon_probs[">=120<180"]
        elif prev_cust_wait>=(180/60):
            abandon_prob = abandon_probs[">=180"]

        return abandon_prob


#############################
class Customer_ls():
    empty = ()

    def __init__(self, customer_ls=np.array([])):
        self.customer_ls = np.array(customer_ls)
        self.customer_ls = self.customer_ls[np.argsort([customer.arrival_time␣
↪for customer in customer_ls])]
        self.next = None if not customer_ls else self.customer_ls[0]


    def __len__(self):
        return len(self.customer_ls)


    def next_exits(self):
        if len(self)==1:
            next_cust, self.customer_ls = self.customer_ls[0], np.array([])
            self.next = None
        else:
            next_cust, self.customer_ls = self.customer_ls[0], self.
↪customer_ls[1:]
            self.next = self.customer_ls[0]
        return next_cust


    def add_to_sort(self, customer):
```

```
        self.customer_ls = np.append(self.customer_ls, customer)
        self.customer_ls = self.customer_ls[np.argsort([customer.arrival_time␣
↪for customer in self.customer_ls])]
        self.next = self.customer_ls[0]


    def add_to_nosort(self, customer):
        self.customer_ls = np.append(self.customer_ls, customer)
        self.next = self.customer_ls[0]


    def sort(self):
        self.customer_ls = self.customer_ls[np.argsort([customer.arrival_time␣
↪for customer in self.customer_ls])]
        self.next = self.customer_ls[0]


#############################
```

0.1 Model #3

[22]:
```
## Loading Speed Factor Function ##
def speed_factor(proportion_express):
    factor = -0.5*proportion_express + 1

        #factor = 1 - 0.8*np.sqrt(proportion_express)
    return factor
```

[272]:
```
## ONE DAY FUNCTION FOR MODEL #3 ##

## The cell contains a function to simulate a single day of the queueing system␣
↪##

def one_day_express(arrival_ls, express_prop, express_abandon_prob,␣
↪ExpressPriority = 1.0):

    #### FOR TUNING ####
    ExpressPriority = 1.0
#     express_prop = 0.15
#     express_abandon_prob = 0.15
    ####################

    customer_arrivals = [Customer(arr) for arr in arrival_ls]
    customer_arrivals = Customer_ls(customer_arrivals)
```

```
    NormalQueue, ExpressQueue, cust_output= Customer_ls(), Customer_ls(),␣
↪Customer_ls()

    switched_11, switched_1 = False, False
    capacity_8, capacity_11, capacity_1 = 136, 170, 204
    train_capacity = capacity_8

    train_finish_time, prev_wait_time = 0, 0
    time = 0
    while len(customer_arrivals) > 0 or len(NormalQueue) > 0 or␣
↪len(ExpressQueue) > 0:

        next_arr = customer_arrivals.next

        ## FIRST CAR DOESN"T LEAVE TILL FULL ##
        if train_finish_time == 0 and len(NormalQueue)<train_capacity:
            if np.random.binomial(n=1,p=express_prop):
                new_arrival = customer_arrivals.next_exits()
                new_arrival.arrival_time += np.random.uniform(0.5,1.5)
                new_arrival.ctype = "express"
                customer_arrivals.add_to_sort(new_arrival)
            else:
                new_arrival = customer_arrivals.next_exits()
                NormalQueue.add_to_nosort(new_arrival)

            if len(NormalQueue)==train_capacity:
                train_finish_time = NormalQueue.customer_ls[-1].arrival_time

        ## ARRIVAL TO SYSTEM ##
        elif len(customer_arrivals)>0 and␣
↪(len(NormalQueue)+len(ExpressQueue)==0 or next_arr.
↪arrival_time<train_finish_time):

            if next_arr.arrival_time>time:
                customer_arrivals.sort()
                time+=0.4
                next_arr = customer_arrivals.next

            ## EXPRESS ARRIVAL
            if next_arr.ctype == "express":
                new_arrival = customer_arrivals.next_exits()
                ExpressQueue.add_to_nosort(new_arrival)

            ## NORMAL ARRIVAL ##
            elif next_arr.ctype == "normal":
                if next_arr.arrival_time <= 10.5 and np.random.
↪binomial(n=1,p=express_prop):
```

```
                    new_arrival = customer_arrivals.next_exits()
                    new_arrival.arrival_time += np.random.uniform(0.5,1.5)
                    new_arrival.ctype = "express"
                    customer_arrivals.add_to_nosort(new_arrival)
                else:
                    new_arrival = customer_arrivals.next_exits()
                    NormalQueue.add_to_nosort(new_arrival)

        ## SEND A TRAIN ##
        else:
            if train_finish_time > 3 and not switched_11:
                train_capacity = capacity_11
                switched_11 = True
            elif train_finish_time > 5 and not switched_1:
                train_capacity = capacity_1
                switched_1 = True

            train_max = min(train_capacity,len(NormalQueue)+len(ExpressQueue))
            rider_types = np.array([])

            train_count = 0
            last_abandon = False
            load_next = np.random.
↪choice(['express','normal'],p=[ExpressPriority,1-ExpressPriority])
            while train_count < train_max:

                ## EXIT IF BOTH QUEUES ARE EMPTY ##
                if len(ExpressQueue)==0 and len(NormalQueue)==0:
                    train_max = train_count

                ## PROCESS THE NEXT CUSTOMER FROM EITHER EXPRESS OR NORMAL ##
                else:
                    load_next = np.random.
↪choice(['express','normal'],p=[ExpressPriority,1-ExpressPriority]) if␣
↪not(last_abandon) else load_next

                    ## NEXT IS EXPRESS ##
                    if len(ExpressQueue)!=0 and load_next=='express':
                        next_served = ExpressQueue.next_exits()
                        abandon_prob = express_abandon_prob

                    ## NEXT IS NORMAL ##
                    elif (len(ExpressQueue)==0 and len(NormalQueue)>0) or␣
↪(len(NormalQueue)!=0 and load_next=='normal'):
                        next_served = NormalQueue.next_exits()
                        abandon_prob = next_served.abandon_prob(prev_wait_time)
```

```
                    ## NO ABANDON ##
                    if np.random.binomial(n=1,p=1-abandon_prob):
                        new_wait_time = max(0,train_finish_time-next_served.
↪arrival_time)
                        next_served.wait_time = new_wait_time
                        cust_output.add_to_nosort(next_served)
                        prev_wait_time = new_wait_time
                        train_count += 1
                        last_abandon=False
                        rider_types = np.append(rider_types, next_served.ctype)

                    ## ABANDON ##
                    else:
                        next_served.wait_time = -999
                        cust_output.add_to_nosort(next_served)
                        last_abandon=True



            ## SPEED FACTOR AND SERVICE TIME ##
            proportion_express = 0 if len(rider_types)==0 else np.
↪count_nonzero(rider_types == 'express') / len(rider_types)
            factor = speed_factor(proportion_express)

            load_min, load_max = 0.5/60, 1./60
            unload_min, unload_max = 0.25/60, 0.75/60

            load_time = factor * np.random.uniform(load_min, load_max)
            unload_time = np.random.uniform(unload_min, unload_max)
            service_time = load_time + 3./60 + unload_time

            train_finish_time = train_finish_time + service_time

    return cust_output
```

[122]:
```
from itertools import product
# obtain_list, unused_list = [.05, .1, .15, .2, .25], [.025, .05, .1, .2, .25]
obtain_list, unused_list = [.05, .15, .35], [.05, .15, .35]

output = list(product(obtain_list, unused_list))
```

[124]:
```
df = pd.DataFrame()
for pair in output:

    trials = 5
    df2 = pd.DataFrame()
    for i in range(0, trials):
```

```
        arrival_ls = np.genfromtxt('data/day' + str(i+1) + '_arrivals.csv')[1:]
        customers = one_day_express(arrival_ls, express_prop = pair[0],␣
↪express_abandon_prob = pair[1]).customer_ls

        data = {
        'day' : i*np.ones(len(arrival_ls)).astype(int),
        'Obtain Probability' : np.repeat(pair[0], len(customers)),
        'Unused Probability' : np.repeat(pair[1], len(customers)),
        'Customer Type' : [customer.ctype for customer in customers],
        'wait' : [60*customer.wait_time for customer in customers],}

        df2 = df2.append(pd.DataFrame(data))
    df = df.append(df2)
    df = df[df['wait'] > 0]
```

```
Day 0 complete
Day 1 complete
Day 2 complete
Day 3 complete
Day 4 complete
Day 0 complete
Day 1 complete
Day 2 complete
Day 3 complete
Day 4 complete
Day 0 complete
Day 1 complete
Day 2 complete
Day 3 complete
Day 4 complete
Day 0 complete
Day 1 complete
Day 2 complete
Day 3 complete
Day 4 complete
Day 0 complete
Day 1 complete
Day 2 complete
Day 3 complete
Day 4 complete
Day 0 complete
Day 1 complete
Day 2 complete
Day 3 complete
Day 4 complete
Day 0 complete
```

```
Day 1 complete
Day 2 complete
Day 3 complete
Day 4 complete
Day 0 complete
Day 1 complete
Day 2 complete
Day 3 complete
Day 4 complete
Day 0 complete
Day 1 complete
Day 2 complete
Day 3 complete
Day 4 complete
```

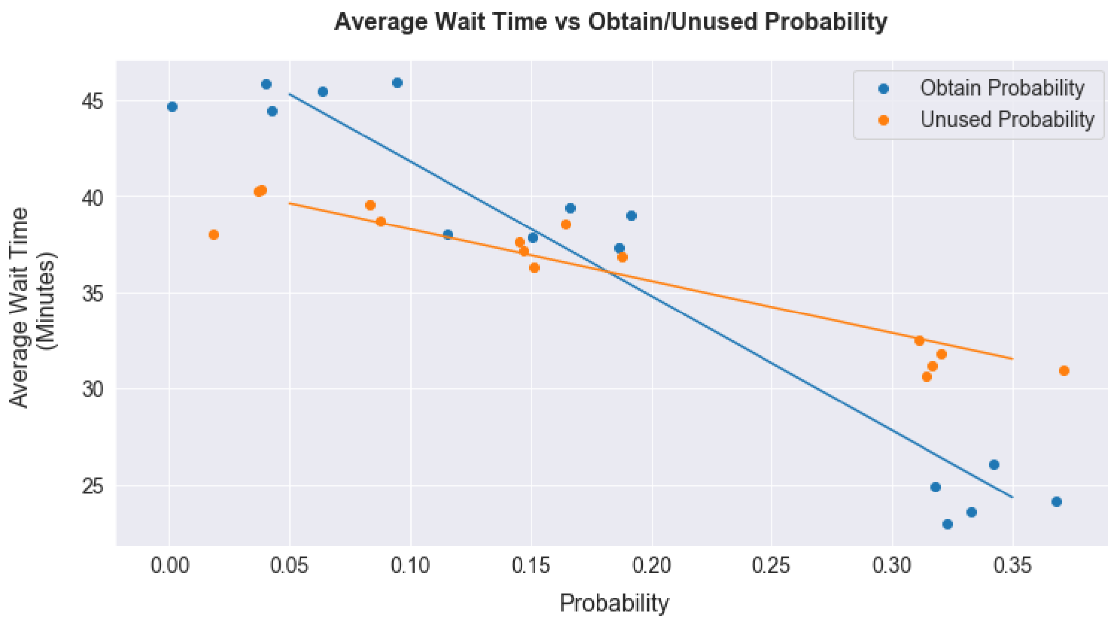
[183]:
```
df_test_obtain = df.groupby(['day', 'Obtain Probability']).mean().drop('Unused␣
↪Probability', axis = 1).reset_index()
df_test_unused = df.groupby(['day', 'Unused Probability']).mean().drop('Obtain␣
↪Probability', axis = 1).reset_index()
```

[327]:
```
from scipy import stats
stats.pearsonr(x2a, y2_p1)
```

[327]:
```
(-0.9711219659089175, 1.8069477717028603e-09)
```

[397]:
```
sns.set_style('darkgrid')
fig4a, ax4a = plt.subplots(figsize = (12, 6))

x1a = df_test_obtain['Obtain Probability']
x2a = df_test_unused['Unused Probability']

x1_p1 = df_test_obtain['Obtain Probability'] + np.random.uniform(-0.05, .05,␣
↪len(df_test_obtain))
y1_p1 = df_test_obtain['wait']

x2_p1 = df_test_unused['Unused Probability'] + np.random.uniform(-0.05, .05,␣
↪len(df_test_unused))
y2_p1 = df_test_unused['wait']

plt.scatter(x1_p1, y1_p1, label = 'Obtain Probability')
plt.scatter(x2_p1, y2_p1, label = 'Unused Probability')
plt.plot(np.unique(x1a), np.poly1d(np.polyfit(x1a, y1_p1, 1))(np.unique(x1a)))
plt.plot(np.unique(x2a), np.poly1d(np.polyfit(x2a, y2_p1, 1))(np.unique(x2a)))


plt.title('Average Wait Time vs Obtain/Unused Probability ', fontsize = 16, pad␣
↪= 20, fontweight = 'semibold')
```

```
plt.xlabel('Probability', fontsize = 16, labelpad = 10)
plt.ylabel('Average Wait Time \n (Minutes)', fontsize = 16, labelpad = 15)
plt.yticks(fontsize = 14)
plt.xticks(fontsize = 14)
plt.legend(prop={'size': 14})
fig4a.savefig('fig4a.pdf')
```

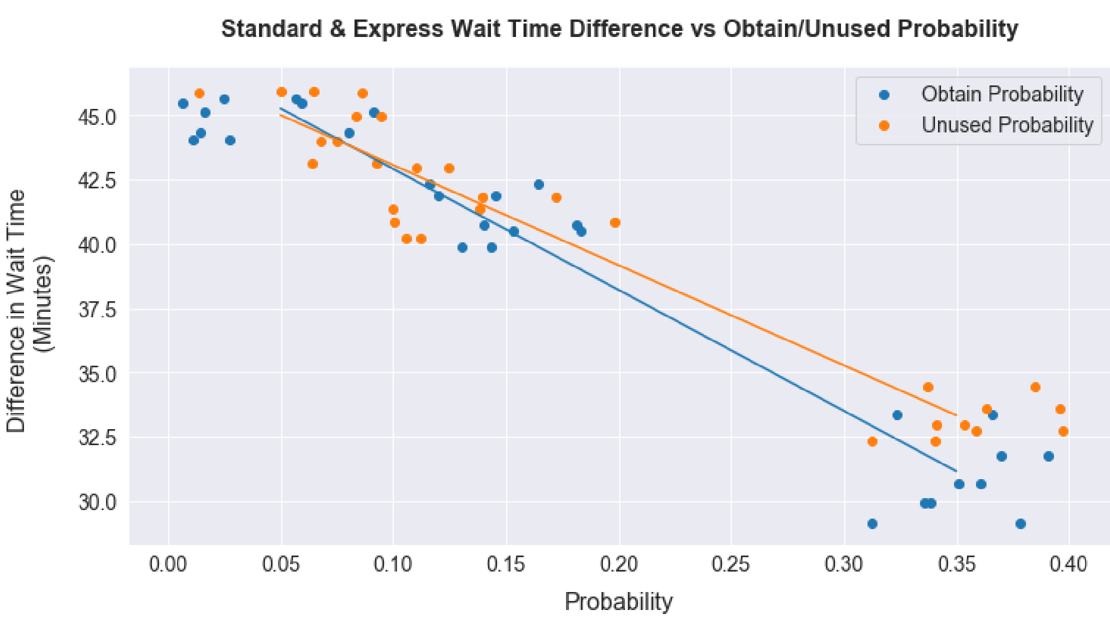


[240]:
```
df_test_obtain_2 = df.groupby(['day', 'Obtain Probability', 'Customer Type']).
↪mean().drop('Unused Probability', axis = 1).reset_index()
df_test_unused_2 = df.groupby(['day', 'Unused Probability', 'Customer Type']).
↪mean().drop('Obtain Probability', axis = 1).reset_index()
```

[317]:
```
a = df_test_obtain_2
b = df_test_unused_2
val_obtain_1 = np.array(a[a['Customer Type'] == 'normal']['wait'].to_list())
val_obtain_2 = np.array(a[a['Customer Type'] == 'express']['wait'].to_list())
val_obtain = val_obtain_1 - val_obtain_2

val_unused_1 = np.array(b[b['Customer Type'] == 'normal']['wait'].to_list())
val_unused_2 = np.array(b[b['Customer Type'] == 'express']['wait'].to_list())
val_unused = val_unused_1 - val_unused_2
```

[396]:
```
sns.set_style('darkgrid')
fig4b, ax4b = plt.subplots(figsize = (12, 6))

x1a_p2 = a['Obtain Probability']
```

```
x2a_p2 = b['Unused Probability']

x1_p2 = a['Obtain Probability'] + np.random.uniform(-0.05, .05, len(a))
y1_p2 = np.repeat(val_obtain, 2)

x2_p2 = b['Unused Probability'] + np.random.uniform(-0.05, .05, len(b))
y2_p2 = np.repeat(val_unused, 2)

plt.scatter(x1_p2, y1_p2, label = 'Obtain Probability')
plt.scatter(x2_p2, y2_p2, label = 'Unused Probability')

plt.plot(np.unique(x1a_p2), np.poly1d(np.polyfit(x1a_p2, y1_p2, 1))(np.
 ↪unique(x1a_p2)))
plt.plot(np.unique(x1a_p2), np.poly1d(np.polyfit(x1a_p2, y2_p2, 1))(np.
 ↪unique(x1a_p2)))


plt.title('Standard & Express Wait Time Difference vs Obtain/Unused␣
 ↪Probability', fontsize = 16, pad = 20, fontweight = 'semibold')
plt.xlabel('Probability', fontsize = 16, labelpad = 10)
plt.ylabel('Difference in Wait Time \n (Minutes)', fontsize = 16, labelpad = 15)
plt.yticks(fontsize = 14)
plt.xticks(fontsize = 14)
plt.legend(prop={'size': 14})
fig4b.savefig('fig4b.pdf')
```



[352]:
```
df = pd.DataFrame()
for probability in [0.25, 0.45, 0.5, 0.65, 0.75, 0.9, 1]:
    trials = 5
    df2 = pd.DataFrame()
    for i in range(0, trials):
        arrival_ls = np.genfromtxt('data/day' + str(i+1) + '_arrivals.csv')[1:]
        customers = one_day_express(arrival_ls, express_prop = .15,
                                    express_abandon_prob = .15,ExpressPriority␣
 ↪= probability).customer_ls

        data = {
        'day' : i*np.ones(len(arrival_ls)).astype(int),
        'Opening Probability' : np.repeat(probability, len(customers)),
        'Customer Type' : [customer.ctype for customer in customers],
        'wait' : [60*customer.wait_time for customer in customers],}

        df2 = df2.append(pd.DataFrame(data))
        print('Day', i, 'Complete')
    df = df.append(df2)
    df = df[df['wait'] > 0]
```

```
Day 0 Complete
Day 1 Complete
Day 2 Complete
Day 3 Complete
Day 4 Complete
Day 0 Complete
Day 1 Complete
Day 2 Complete
Day 3 Complete
Day 4 Complete
Day 0 Complete
Day 1 Complete
Day 2 Complete
Day 3 Complete
Day 4 Complete
Day 0 Complete
Day 1 Complete
Day 2 Complete
Day 3 Complete
Day 4 Complete
Day 0 Complete
Day 1 Complete
Day 2 Complete
Day 3 Complete
Day 4 Complete
```

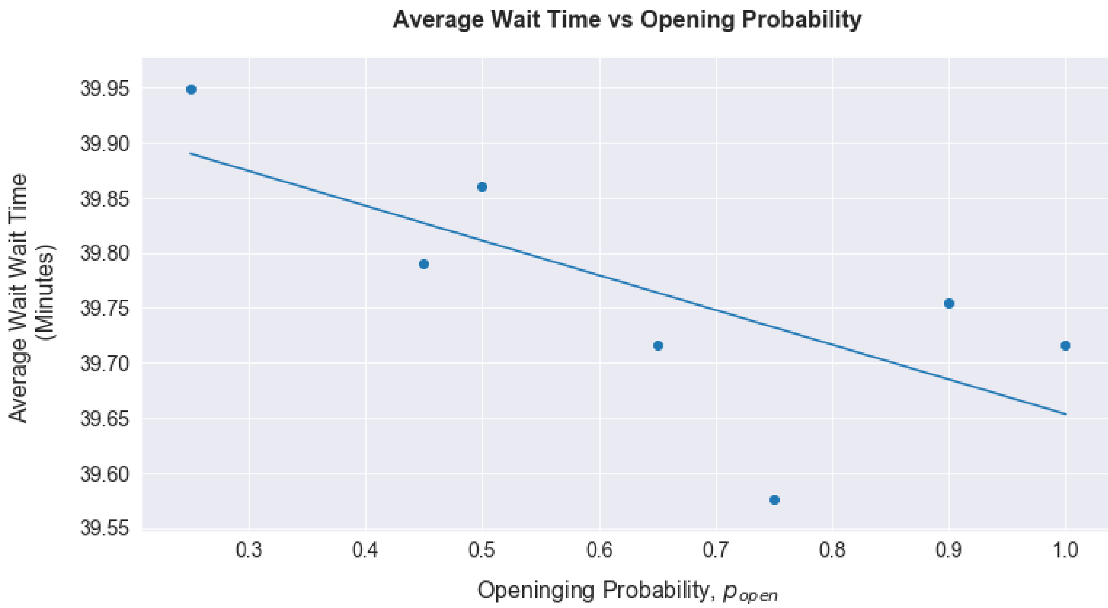
```
Day 0 Complete
Day 1 Complete
Day 2 Complete
Day 3 Complete
Day 4 Complete
Day 0 Complete
Day 1 Complete
Day 2 Complete
Day 3 Complete
Day 4 Complete
```

[380]:
```
df_open = df.groupby(['day', 'Opening Probability']).mean().reset_index()
df_test_open = df_open.groupby('Opening Probability').mean()[['wait']].
 ↪reset_index()
df_standard = df[df['Customer Type'] == 'normal'].groupby('Opening␣
 ↪Probability').mean()[['wait']].reset_index()
df_express = df[df['Customer Type'] == 'express'].groupby('Opening␣
 ↪Probability').mean()[['wait']]
```

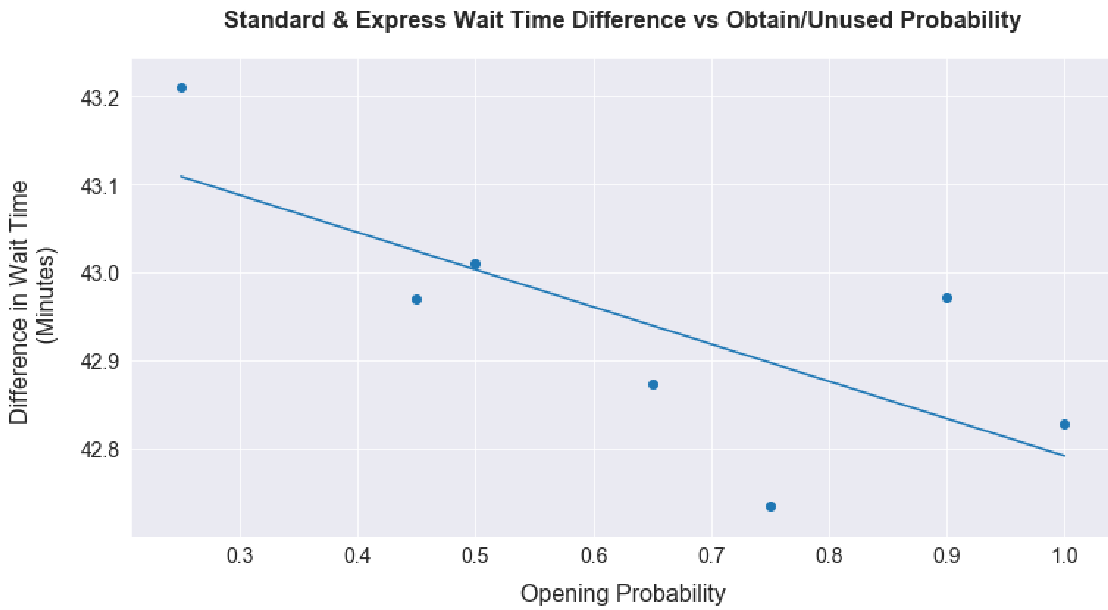
[380]:

	day	Opening Probability	wait
24	3	0.65	38.100527
25	3	0.75	38.740807
22	3	0.45	38.741014
5	0	0.90	38.817005
4	0	0.75	38.825012

[415]:
```
# Plotting Avg Wait Time vs Opening
sns.set_style('darkgrid')
fig4c, ax4c = plt.subplots(figsize = (12, 6))

xx = df_test_open['Opening Probability'] #+ np.random.uniform(-0.025, .025,␣
 ↪len(df_open))
yy = df_test_open['wait']


plt.scatter(xx, yy)
plt.plot(np.unique(xx), np.poly1d(np.polyfit(xx, yy, 1))(np.unique(xx)))


plt.title('Average Wait Time vs Opening Probability', fontsize = 16, pad = 20,␣
 ↪fontweight = 'semibold')
plt.xlabel('Openinging Probability, $p_{open}$', fontsize = 16, labelpad = 10)
plt.ylabel('Average Wait Wait Time \n (Minutes)', fontsize = 16, labelpad = 15)
plt.yticks(fontsize = 14)
plt.xticks(fontsize = 14)
#plt.legend(prop={'size': 14})
fig4c.savefig('fig4c.pdf')
```



[414]:
```
# Plotting Avg Wait Time vs Opening
sns.set_style('darkgrid')
fig4d, ax4d = plt.subplots(figsize = (12, 6))

xx2 = df_standard['Opening Probability']
yy2 = df_standard['wait'].values - df_express['wait'].values


plt.scatter(xx2, yy2)
plt.plot(np.unique(xx2), np.poly1d(np.polyfit(xx2, yy2, 1))(np.unique(xx2)))


plt.title('Standard & Express Wait Time Difference vs Obtain/Unused␣
↪Probability', fontsize = 16, pad = 20, fontweight = 'semibold')
plt.xlabel('Opening Probability', fontsize = 16, labelpad = 10)
plt.ylabel('Difference in Wait Time \n (Minutes)', fontsize = 16, labelpad = 15)
plt.yticks(fontsize = 14)
plt.xticks(fontsize = 14)
fig4d.savefig('fig4d.pdf')
```



[368]:
```
df_open_2 = df.groupby(['day', 'Opening Probability', 'Customer Type']).mean().
 ↪reset_index()
df_open_2_normal = df_open_2[df_open_2['Customer Type'] == 'normal'].
 ↪sort_values('Opening Probability', ascending = True)
df_open_2_express = df_open_2[df_open_2['Customer Type'] == 'express'].
 ↪sort_values('Opening Probability', ascending = True)
val_open_1 = np.array(df_open_2_normal['wait'].to_list())
val_open_2 = np.array(df_open_2_express['wait'].to_list())
val_open = val_open_1 - val_open_2
```

[ ]:

[ ]:

6.8 Appendix H: Model #4 - Part II

This is the second appendix of our tuning and sensitivity analysis. It performs the analysis on the system parameters and outputs the combinations of parameters and corresponding wait times.

model_4b_final

December 14, 2020

[4]:
```
## Importing Libraries ##
import numpy as np
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import math
from IPython.display import Markdown as md
import warnings
warnings.filterwarnings('ignore')
# %matplotlib inline
from scipy.optimize import minimize
import time
```

[5]:
```
## This cell defines our Poission Process, Customer, and Customer List Classes␣
↪##

#############################
class PoissonProcess():
    def __init__(self, lam, T):
        self.lam = lam
        self.T = T
        self.simulate()


    def simulate(self, method='inter_arrival_time'):
        if method == 'inter_arrival_time':
            N = int(self.lam * self.T * 1.3)
            inter_ls = np.random.exponential(1/self.lam, size=N)
            arrival_time_ls = np.cumsum(inter_ls)
            self.arrival_time_ls = arrival_time_ls[arrival_time_ls <= self.T]
        if method == 'uniformity_property':
            N = np.random.poisson(self.T * self.lam)
            arrival_time_ls = np.random.uniform(0, self.T, size=N)
            self.arrival_time_ls = np.sort(arrival_time_ls)
```

```
    def get_arrival_time(self):
        return self.arrival_time_ls


    def print_parameter(self):
        print('lambda = {}, T = {}'.format(self.lam, self.T))


    def N_t(self, t):
        assert t >= 0
        assert t <= self.T
        if t == 0:
            return 0
        else:
            return np.argmax(self.arrival_time_ls > t)


    def plot_N_t(self, color='r',alpha=1):
        positive_inf = max(self.arrival_time_ls) * 1.2
        negative_inf = - max(self.arrival_time_ls) * 0.1
        n_arrival = len(self.arrival_time_ls)
        x_ls = np.concatenate([[negative_inf, 0], np.repeat(self.
 ↪arrival_time_ls,2), [positive_inf]])
        y_ls = np.concatenate([[0], np.repeat(np.arange(n_arrival + 1),2)])
        plt.plot(x_ls, y_ls, c=color, alpha=alpha)

##############################
class Customer():
    def __init__(self, arrival_time=0, ctype='normal',wait_time=None):
        self.arrival_time = arrival_time
        self.ctype = ctype
        self.wait_time = wait_time


    def abandon_prob(self,prev_cust_wait):

        abandon_probs = {"<40":.005, ">=40<50":0.015, ">=50<60":0.03, ">=60<75":
 ↪0.08, ">=75<90":0.1,
                         ">=90<105":0.1, ">=105<120":0.1, ">=120<180":0.1,␣
 ↪">=180":0.15,
                        }
        ### 170 is <<< 1800 so so our approximation the people in the line is␣
 ↪okay
        if prev_cust_wait == 0:
            abandon_prob = 0
        elif prev_cust_wait<(40/60):
            abandon_prob = abandon_probs["<40"]
```

```
        elif prev_cust_wait>=(40/60) and prev_cust_wait<(50/60):
            abandon_prob = abandon_probs[">=40<50"]
        elif prev_cust_wait>=(50/60) and prev_cust_wait<(60/60):
            abandon_prob = abandon_probs[">=50<60"]
        elif prev_cust_wait>=(60/60) and prev_cust_wait<(75/60):
            abandon_prob = abandon_probs[">=60<75"]
        elif prev_cust_wait>=(75/60) and prev_cust_wait<(90/60):
            abandon_prob = abandon_probs[">=75<90"]
        elif prev_cust_wait>=(90/60) and prev_cust_wait<(105/60):
            abandon_prob = abandon_probs[">=90<105"]
        elif prev_cust_wait>=(105/60) and prev_cust_wait<(120/60):
            abandon_prob = abandon_probs[">=105<120"]
        elif prev_cust_wait>=(120/60) and prev_cust_wait<(180/60):
            abandon_prob = abandon_probs[">=120<180"]
        elif prev_cust_wait>=(180/60):
            abandon_prob = abandon_probs[">=180"]

        return abandon_prob


#############################
class Customer_ls():
    empty = ()

    def __init__(self, customer_ls=np.array([])):
        self.customer_ls = np.array(customer_ls)
        self.customer_ls = self.customer_ls[np.argsort([customer.arrival_time␣
 ↪for customer in customer_ls])]
        self.next = None if not customer_ls else self.customer_ls[0]


    def __len__(self):
        return len(self.customer_ls)


    def next_exits(self):
        if len(self)==1:
            next_cust, self.customer_ls = self.customer_ls[0], np.array([])
            self.next = None
        else:
            next_cust, self.customer_ls = self.customer_ls[0], self.
 ↪customer_ls[1:]
            self.next = self.customer_ls[0]
        return next_cust


    def add_to_sort(self, customer):
```

```
        self.customer_ls = np.append(self.customer_ls, customer)
        self.customer_ls = self.customer_ls[np.argsort([customer.arrival_time␣
↪for customer in self.customer_ls])]
        self.next = self.customer_ls[0]


    def add_to_nosort(self, customer):
        self.customer_ls = np.append(self.customer_ls, customer)
        self.next = self.customer_ls[0]


    def sort(self):
        self.customer_ls = self.customer_ls[np.argsort([customer.arrival_time␣
↪for customer in self.customer_ls])]
        self.next = self.customer_ls[0]


#############################
## Loading Speed Factor Function ##
def speed_factor(proportion_express):
    factor = -0.5*proportion_express + 1
    return factor
```

[6]:
```
## ONE DAY FUNCTION FOR MODEL #3 ##

##The cell contains a function to simulate a single day of the queueing system␣
↪##

def one_day_duration(arrival_ls, duration):

    ###  FOR TUNING  ###
    ExpressPriority = 1.0

    express_prop = 0.15
    express_abandon_prob = 0.15

    ####################

    customer_arrivals = [Customer(arr) for arr in arrival_ls]
    customer_arrivals = Customer_ls(customer_arrivals)

    NormalQueue, ExpressQueue, cust_output= Customer_ls(), Customer_ls(),␣
↪Customer_ls()

    switched_11, switched_1 = False, False
    capacity_8, capacity_11, capacity_1 = 136, 170, 204
    train_capacity = capacity_8
```

```
    train_finish_time, prev_wait_time = 0, 0
    time = 0
    while len(customer_arrivals) > 0 or len(NormalQueue) > 0 or␣
↪len(ExpressQueue) > 0:

        next_arr = customer_arrivals.next

        ## FIRST CAR DOESN"T LEAVE TILL FULL ##
        if train_finish_time == 0 and len(NormalQueue)<train_capacity:
            if np.random.binomial(n=1,p=express_prop):
                new_arrival = customer_arrivals.next_exits()
                new_arrival.arrival_time += np.random.uniform(0.5,1.5)
                new_arrival.ctype = "express"
                customer_arrivals.add_to_sort(new_arrival)
            else:
                new_arrival = customer_arrivals.next_exits()
                NormalQueue.add_to_nosort(new_arrival)

            if len(NormalQueue)==train_capacity:
                train_finish_time = NormalQueue.customer_ls[-1].arrival_time

        ## ARRIVAL TO SYSTEM ##
        elif len(customer_arrivals)>0 and␣
↪(len(NormalQueue)+len(ExpressQueue)==0 or next_arr.
↪arrival_time<train_finish_time):

            if next_arr.arrival_time>time:
                customer_arrivals.sort()
                time+=0.4
                next_arr = customer_arrivals.next

            ## EXPRESS ARRIVAL
            if next_arr.ctype == "express":
                new_arrival = customer_arrivals.next_exits()
                ExpressQueue.add_to_nosort(new_arrival)

            ## NORMAL ARRIVAL ##
            elif next_arr.ctype == "normal":

                ## ACCEPT EXPRESS PASS ##
                if np.random.binomial(n=1,p=express_prop) and next_arr.
↪arrival_time <= 10.5:
                    if next_arr.arrival_time <= 12 - (0.5 + duration):
                        new_arrival = customer_arrivals.next_exits()
                        new_arrival.arrival_time += np.random.uniform(0.5,␣
↪duration+0.5)
```

```
                        new_arrival.ctype = "express"
                        customer_arrivals.add_to_nosort(new_arrival)
                    elif next_arr.arrival_time > 12 - (0.5 + duration):
                        new_arrival = customer_arrivals.next_exits()
                        new_arrival.arrival_time += np.random.uniform(0.
↪5,12-new_arrival.arrival_time)
                        new_arrival.ctype = "express"
                        customer_arrivals.add_to_nosort(new_arrival)

                ## ALWAYS STOP GIVING OUT EXPRESS PASSES AT 10.5 ##
                else:
                    new_arrival = customer_arrivals.next_exits()
                    NormalQueue.add_to_nosort(new_arrival)

#                 if np.random.binomial(n=1,p=express_prop):
#                     if next_arr.arrival_time <= 12 - (0.5 + duration):
#                         new_arrival = customer_arrivals.next_exits()
#                         new_arrival.arrival_time += np.random.uniform(0.
↪5,duration+0.5)
#                         new_arrival.ctype = "express"
#                         customer_arrivals.add_to_nosort(new_arrival)                    ␣
↪
#                     elif next_arr.arrival_time > 12 - (0.5+duration):
#                         new_arrival = customer_arrivals.next_exits()
#                         new_arrival.arrival_time += np.random.uniform(0.
↪5,12-new_arrival.arrival_time)
#                         new_arrival.ctype = "express"
#                         customer_arrivals.add_to_nosort(new_arrival)

        ## SEND A TRAIN ##
        else:
            if train_finish_time > 3 and not switched_11:
                train_capacity = capacity_11
                switched_11 = True
            elif train_finish_time > 5 and not switched_1:
                train_capacity = capacity_1
                switched_1 = True

            train_max = min(train_capacity,len(NormalQueue)+len(ExpressQueue))
            rider_types = np.array([])

            train_count = 0
            last_abandon = False
            load_next = np.random.
↪choice(['express','normal'],p=[ExpressPriority,1-ExpressPriority])
            while train_count < train_max:
```

```
                ## EXIT IF BOTH QUEUES ARE EMPTY ##
                if len(ExpressQueue)==0 and len(NormalQueue)==0:
                    train_max = train_count

                ## PROCESS THE NEXT CUSTOMER FROM EITHER EXPRESS OR NORMAL ##
                else:
                    load_next = np.random.
↪choice(['express','normal'],p=[ExpressPriority,1-ExpressPriority]) if␣
↪not(last_abandon) else load_next

                    ## NEXT IS EXPRESS ##
                    if len(ExpressQueue)!=0 and load_next=='express':
                        next_served = ExpressQueue.next_exits()
                        abandon_prob = express_abandon_prob

                    ## NEXT IS NORMAL ##
                    elif (len(ExpressQueue)==0 and len(NormalQueue)>0) or␣
↪(len(NormalQueue)!=0 and load_next=='normal'):
                        next_served = NormalQueue.next_exits()
                        abandon_prob = next_served.abandon_prob(prev_wait_time)

                    ## NO ABANDON ##
                    if np.random.binomial(n=1,p=1-abandon_prob):
                        new_wait_time = max(0,train_finish_time-next_served.
↪arrival_time)
                        next_served.wait_time = new_wait_time
                        cust_output.add_to_nosort(next_served)
                        prev_wait_time = new_wait_time
                        train_count += 1
                        last_abandon=False
                        rider_types = np.append(rider_types, next_served.ctype)

                    ## ABANDON ##
                    else:
                        next_served.wait_time = -999
                        cust_output.add_to_nosort(next_served)
                        last_abandon=True



            ## SPEED FACTOR AND SERVICE TIME ##
            proportion_express = 0 if len(rider_types)==0 else np.
↪count_nonzero(rider_types == 'express') / len(rider_types)
            factor = speed_factor(proportion_express)

            load_min, load_max = 0.5/60, 1./60
            unload_min, unload_max = 0.25/60, 0.75/60
```

```
            load_time = factor * np.random.uniform(load_min, load_max)
            unload_time = np.random.uniform(unload_min, unload_max)
            service_time = load_time + 3./60 + unload_time

            train_finish_time = train_finish_time + service_time

    return cust_output
```

[8]:
```
agg_avg, agg_max, norm_avg, norm_max, express_avg, express_max =␣
 ↪[],[],[],[],[],[]
for duration in np.arange(1.0,11.51,0.5):
    # Simulating Trials
    all_days, all_arrival_hours, all_arrivals, all_waits, all_custtypes =␣
 ↪[],[],[],[],[]
    trials = 2
    for i in np.arange(1, trials + 1):
        arrival_ls = np.genfromtxt('data/day' + str(i) + '_arrivals.csv')[1:]
        customers = one_day_duration(arrival_ls, duration).customer_ls

        all_days = np.append(all_days, i*np.ones(len(customers))).astype(int)
        all_arrival_hours = np.append(all_arrival_hours, [int(customer.
 ↪arrival_time) for customer in customers]).astype(int)
        all_arrivals = np.append(all_arrivals, [customer.arrival_time for␣
 ↪customer in customers])
        all_waits = np.append(all_waits, [60*customer.wait_time for customer in␣
 ↪customers])
        all_custtypes = np.append(all_custtypes, [customer.ctype for customer␣
 ↪in customers])
    df_data = {'day':all_days,'arrival hour':all_arrival_hours,'arrival':
 ↪all_arrivals,
                'wait':np.round(all_waits,3),'Customer Type':all_custtypes}
    df = pd.DataFrame(df_data)

    ## CLEANING OUT DATAFRAME ##
    df = df[df['wait'] >= 0]
    df = df.replace( {'normal' : 'Normal', 'express' : 'Express'})
    ## SEPARATE NORMAL & EXPRESS ##
    normal_df, express_df = df[df['Customer Type'] == 'Normal'],␣
 ↪df[df['Customer Type'] == 'Express']

    agg_avg = np.append(agg_avg,np.round(np.mean(df['wait']),3))
    agg_max = np.append(agg_max,np.round(np.max(df['wait']),3))

    norm_avg = np.append(norm_avg,np.round(np.mean(normal_df['wait']),3))
    norm_max = np.append(norm_max,np.round(np.max(normal_df['wait']),3))
```

```
    express_avg = np.append(express_avg,np.round(np.mean(express_df['wait']),3))
    express_max = np.append(express_max,np.round(np.max(express_df['wait']),3))

all_durations = np.arange(1.00,11.51,0.50)
df2_data = {
    'ExpressPass Duration':all_durations,
    'Aggregate Avg':agg_avg,
    'Express Avg':express_avg,
    'Normal Avg':norm_avg,
    'Aggregate Max':agg_max,
    'Express Max':express_avg,
    'Normal Max':norm_max
}
df2 = pd.DataFrame(df2_data).set_index('ExpressPass Duration')
df2
```

[8]:
```
                      Aggregate Avg  Express Avg  Normal Avg  Aggregate Max  \
ExpressPass Duration
1.0                          40.085        2.113      45.343         64.532
1.5                          39.312        2.105      44.516         64.226
2.0                          38.522        2.081      43.649         62.845
2.5                          36.776        2.080      41.616         62.311
3.0                          36.501        2.098      41.228         61.605
3.5                          35.793        2.113      40.453         60.275
4.0                          35.879        2.107      40.522         61.662
4.5                          34.574        2.094      38.996         58.165
5.0                          32.342        2.090      36.552         53.792
5.5                          33.834        2.101      38.053         58.078
6.0                          31.660        2.081      35.701         53.278
6.5                          33.492        2.118      37.752         54.959
7.0                          31.648        2.079      35.729         54.381
7.5                          33.213        2.109      37.519         58.290
8.0                          31.138        2.079      35.041         56.600
8.5                          30.949        2.121      34.951         53.332
9.0                          30.441        2.118      34.294         51.295
9.5                          30.385        2.132      34.271         57.594
10.0                         30.070        2.067      33.878         52.140
10.5                         30.153        2.088      33.956         56.673
11.0                         28.410        2.091      32.012         50.754
11.5                         30.302        2.130      34.172         53.737

                      Express Max  Normal Max
ExpressPass Duration
1.0                         2.113      64.532
1.5                         2.105      64.226
2.0                         2.081      62.845
```

```
2.5                            2.080      62.311
3.0                            2.098      61.605
3.5                            2.113      60.275
4.0                            2.107      61.662
4.5                            2.094      58.165
5.0                            2.090      53.792
5.5                            2.101      58.078
6.0                            2.081      53.278
6.5                            2.118      54.959
7.0                            2.079      54.381
7.5                            2.109      58.290
8.0                            2.079      56.600
8.5                            2.121      53.332
9.0                            2.118      51.295
9.5                            2.132      57.594
10.0                           2.067      52.140
10.5                           2.088      56.673
11.0                           2.091      50.754
11.5                           2.130      53.737
```

[13]:
```
df2_short = df2.iloc[:, [0,1,2]]
df2_short.head()
```

[13]:
```
                      Aggregate Avg  Express Avg  Normal Avg
ExpressPass Duration
1.0                          40.085        2.113      45.343
1.5                          39.312        2.105      44.516
2.0                          38.522        2.081      43.649
2.5                          36.776        2.080      41.616
3.0                          36.501        2.098      41.228
```

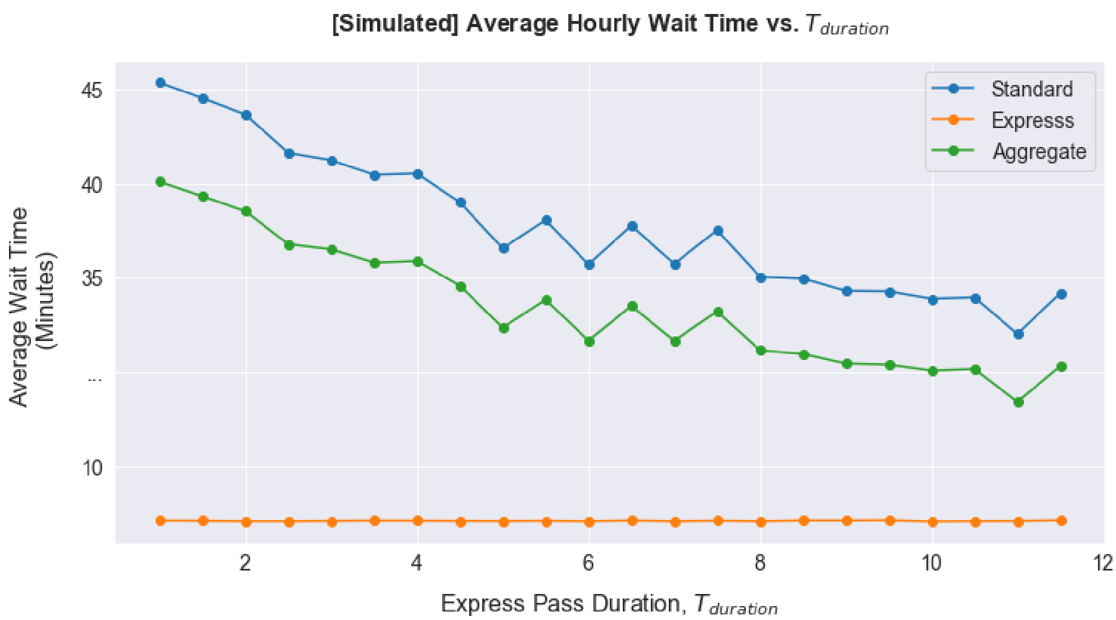
[41]:
```
np.arange(27.5, 45.1, 2.5)
```

[41]:
```
array([27.5, 30. , 32.5, 35. , 37.5, 40. , 42.5, 45. ])
```

[45]:
```
sns.set_style('darkgrid')
fig5atest, ax = plt.subplots(figsize = (12, 6))

plt.plot(df2_short.index, df2_short.iloc[:, 2], label = 'Standard', marker =␣
 ↪'o')
plt.plot(df2_short.index, df2_short.iloc[:, 1] + 20, label = 'Expresss', marker␣
 ↪= 'o')
plt.plot(df2_short.index, df2_short.iloc[:, 0], label = 'Aggregate', marker =␣
 ↪'o')
```

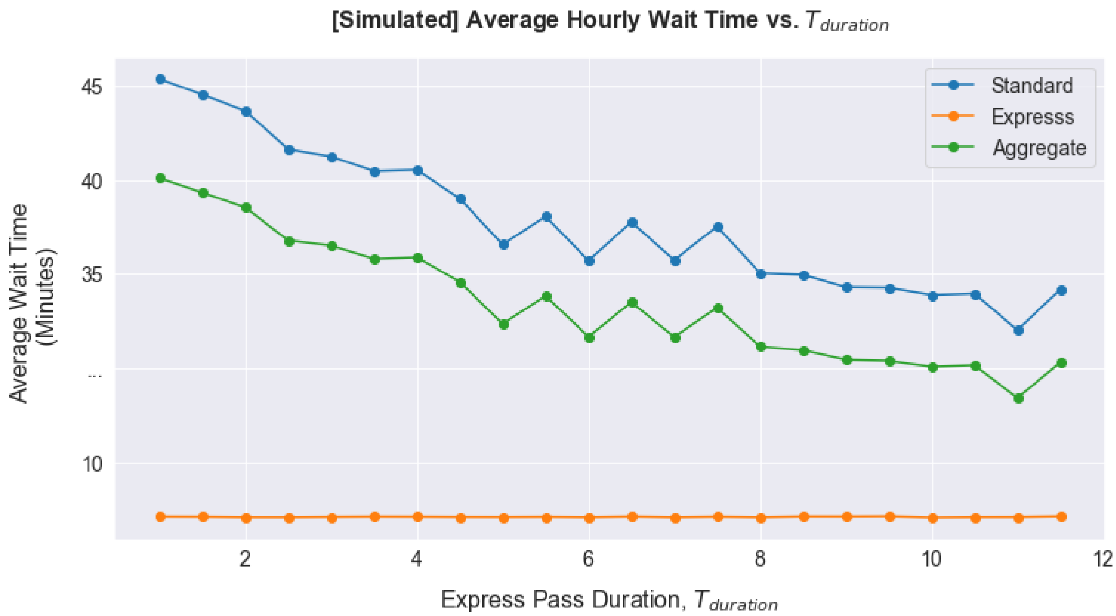
```
plt.title('[Simulated] Average Hourly Wait Time vs. $T_{duration}$', fontsize =␣
↪16, pad = 20, fontweight = 'semibold')
plt.xlabel('Express Pass Duration, $T_{duration}$', fontsize = 16, labelpad =␣
↪10)
plt.ylabel('Average Wait Time \n (Minutes)', fontsize = 16, labelpad = 15)
plt.yticks(np.arange(25, 50, 5), ['10', '...', '35', '40', '45'], fontsize = 14)
#plt.yticks(np.arange(27.5, 45.1, 2.5), ['10', '...', '20', '20', '30', '40',␣
↪'50'], fontsize = 14)
plt.xticks(fontsize = 14)
plt.legend(prop={'size': 14})
```

[45]: <matplotlib.legend.Legend at 0x1a209f4cd0>



[47]:
```
sns.set_style('darkgrid')
fig5a, ax5a = plt.subplots(figsize = (12, 6))

plt.plot(df2_short.index, df2_short.iloc[:, 2], label = 'Standard', marker =␣
↪'o')
plt.plot(df2_short.index, df2_short.iloc[:, 1] + 20, label = 'Expresss', marker␣
↪= 'o')
plt.plot(df2_short.index, df2_short.iloc[:, 0], label = 'Aggregate', marker =␣
↪'o')


plt.title('[Simulated] Average Hourly Wait Time vs. $T_{duration}$', fontsize =␣
↪16, pad = 20, fontweight = 'semibold')
```

```
plt.xlabel('Express Pass Duration, $T_{duration}$', fontsize = 16, labelpad =␣
 ↪10)
plt.ylabel('Average Wait Time \n (Minutes)', fontsize = 16, labelpad = 15)
plt.yticks(np.arange(25, 50, 5), ['10', '...', '35', '40', '45'], fontsize = 14)
#plt.yticks(np.arange(27.5, 45.1, 2.5), ['10', '...', '20', '20', '30', '40',␣
 ↪'50'], fontsize = 14)
plt.xticks(fontsize = 14)
plt.legend(prop={'size': 14})
fig5a.savefig('fig5a.pdf')
```



[21]:
```
## ONE DAY FUNCTION FOR MODEL #3 ##

##The cell contains a function to simulate a single day of the queueing system␣
 ↪##

def one_day_delay_duration(arrival_ls, delay, duration):

    ###  FOR TUNING  ###
    ExpressPriority = 1.0

    express_prop = 0.15
    express_abandon_prob = 0.15

    ####################

    customer_arrivals = [Customer(arr) for arr in arrival_ls]
```

```
    customer_arrivals = Customer_ls(customer_arrivals)

    NormalQueue, ExpressQueue, cust_output= Customer_ls(), Customer_ls(),␣
↪Customer_ls()

    switched_11, switched_1 = False, False
    capacity_8, capacity_11, capacity_1 = 136, 170, 204
    train_capacity = capacity_8

    train_finish_time, prev_wait_time = 0, 0
    time = 0
    while len(customer_arrivals) > 0 or len(NormalQueue) > 0 or␣
↪len(ExpressQueue) > 0:

        next_arr = customer_arrivals.next

        ## FIRST CAR DOESN"T LEAVE TILL FULL ##
        if train_finish_time == 0 and len(NormalQueue)<train_capacity:
            if np.random.binomial(n=1,p=express_prop):
                new_arrival = customer_arrivals.next_exits()
                new_arrival.arrival_time += np.random.
↪uniform(delay,duration+delay)
                new_arrival.ctype = "express"
                customer_arrivals.add_to_nosort(new_arrival)
            else:
                new_arrival = customer_arrivals.next_exits()
                NormalQueue.add_to_nosort(new_arrival)

            if len(NormalQueue)==train_capacity:
                train_finish_time = NormalQueue.customer_ls[-1].arrival_time

        ## ARRIVAL TO SYSTEM ##
        elif len(customer_arrivals)>0 and␣
↪(len(NormalQueue)+len(ExpressQueue)==0 or next_arr.
↪arrival_time<train_finish_time):

            if next_arr.arrival_time>time:
                customer_arrivals.sort()
                time+=delay-0.05
                next_arr = customer_arrivals.next

            ## EXPRESS ARRIVAL
            if next_arr.ctype == "express":
                new_arrival = customer_arrivals.next_exits()
                ExpressQueue.add_to_nosort(new_arrival)

            ## NORMAL ARRIVAL ##
```

```
            elif next_arr.ctype == "normal":

                ## ACCEPT EXPRESS PASS ##
                shutoff_time = 11- delay
                if np.random.binomial(n=1,p=express_prop) and next_arr.
↪arrival_time <= shutoff_time:
                    if next_arr.arrival_time <= 12 - (delay + duration):
                        new_arrival = customer_arrivals.next_exits()
                        new_arrival.arrival_time += np.random.uniform(delay,␣
↪duration+delay)
                        new_arrival.ctype = "express"
                        customer_arrivals.add_to_nosort(new_arrival)
                    elif next_arr.arrival_time > 12 - (delay + duration):
                        new_arrival = customer_arrivals.next_exits()
                        new_arrival.arrival_time += np.random.
↪uniform(delay,12-new_arrival.arrival_time)
                        new_arrival.ctype = "express"
                        customer_arrivals.add_to_nosort(new_arrival)

                ## ALWAYS STOP GIVING OUT EXPRESS PASSES AT 10.5 ##
                else:
                    new_arrival = customer_arrivals.next_exits()
                    NormalQueue.add_to_nosort(new_arrival)


        ## SEND A TRAIN ##
        else:
            if train_finish_time > 3 and not switched_11:
                train_capacity = capacity_11
                switched_11 = True
            elif train_finish_time > 5 and not switched_1:
                train_capacity = capacity_1
                switched_1 = True

            train_max = min(train_capacity,len(NormalQueue)+len(ExpressQueue))
            rider_types = np.array([])

            train_count = 0
            last_abandon = False
            load_next = np.random.
↪choice(['express','normal'],p=[ExpressPriority,1-ExpressPriority])
            while train_count < train_max:

                ## EXIT IF BOTH QUEUES ARE EMPTY ##
                if len(ExpressQueue)==0 and len(NormalQueue)==0:
                    train_max = train_count
```

```
            ## PROCESS THE NEXT CUSTOMER FROM EITHER EXPRESS OR NORMAL ##
            else:
                load_next = np.random.
↪choice(['express','normal'],p=[ExpressPriority,1-ExpressPriority]) if␣
↪not(last_abandon) else load_next

                ## NEXT IS EXPRESS ##
                if len(ExpressQueue)!=0 and load_next=='express':
                    next_served = ExpressQueue.next_exits()
                    abandon_prob = express_abandon_prob

                ## NEXT IS NORMAL ##
                elif (len(ExpressQueue)==0 and len(NormalQueue)>0) or␣
↪(len(NormalQueue)!=0 and load_next=='normal'):
                    next_served = NormalQueue.next_exits()
                    abandon_prob = next_served.abandon_prob(prev_wait_time)

                ## NO ABANDON ##
                if np.random.binomial(n=1,p=1-abandon_prob):
                    new_wait_time = max(0,train_finish_time-next_served.
↪arrival_time)
                    next_served.wait_time = new_wait_time
                    cust_output.add_to_nosort(next_served)
                    prev_wait_time = new_wait_time
                    train_count += 1
                    last_abandon=False
                    rider_types = np.append(rider_types, next_served.ctype)

                ## ABANDON ##
                else:
                    next_served.wait_time = -999
                    cust_output.add_to_nosort(next_served)
                    last_abandon=True



        ## SPEED FACTOR AND SERVICE TIME ##
        proportion_express = 0 if len(rider_types)==0 else np.
↪count_nonzero(rider_types == 'express') / len(rider_types)
        factor = speed_factor(proportion_express)

        load_min, load_max = 0.5/60, 1./60
        unload_min, unload_max = 0.25/60, 0.75/60

        load_time = factor * np.random.uniform(load_min, load_max)
        unload_time = np.random.uniform(unload_min, unload_max)
        service_time = load_time + 3./60 + unload_time
```

```
            train_finish_time = train_finish_time + service_time

    return cust_output
```

[22]:
```
all_durations = []
all_delays = []
agg_avg, agg_max, norm_avg, norm_max, express_avg, express_max =␣
↪[],[],[],[],[],[]

for delay in np.arange(0.5,3.01,0.5):
    durations = np.array([])
    duration = 1
    while duration + delay <= 12:

        # Simulating Trials
        all_days, all_arrival_hours, all_arrivals, all_waits, all_custtypes =␣
↪[],[],[],[],[]
        trials = 2
        for i in np.arange(1, trials + 1):
            arrival_ls = np.genfromtxt('data/day' + str(i) + '_arrivals.csv')[1:
↪]
            customers = one_day_delay_duration(arrival_ls, delay, duration).
↪customer_ls

            all_days = np.append(all_days, i*np.ones(len(customers))).
↪astype(int)
            all_arrival_hours = np.append(all_arrival_hours, [int(customer.
↪arrival_time) for customer in customers]).astype(int)
            all_arrivals = np.append(all_arrivals, [customer.arrival_time for␣
↪customer in customers])
            all_waits = np.append(all_waits, [60*customer.wait_time for␣
↪customer in customers])
            all_custtypes = np.append(all_custtypes, [customer.ctype for␣
↪customer in customers])
        df_data = {'day':all_days,'arrival hour':all_arrival_hours,'arrival':
↪all_arrivals,
                'wait':np.round(all_waits,3),'Customer Type':all_custtypes}
        df = pd.DataFrame(df_data)

        ## CLEANING OUT DATAFRAME ##
        df = df[df['wait'] >= 0]
        df = df.replace( {'normal' : 'Normal', 'express' : 'Express'})
        ## SEPARATE NORMAL & EXPRESS ##
        normal_df, express_df = df[df['Customer Type'] == 'Normal'],␣
↪df[df['Customer Type'] == 'Express']
```

```
        agg_avg = np.append(agg_avg,np.round(np.mean(df['wait']),3))
        agg_max = np.append(agg_max,np.round(np.max(df['wait']),3))

        norm_avg = np.append(norm_avg,np.round(np.mean(normal_df['wait']),3))
        norm_max = np.append(norm_max,np.round(np.max(normal_df['wait']),3))

        express_avg = np.append(express_avg,np.round(np.
 ↪mean(express_df['wait']),3))
        express_max = np.append(express_max,np.round(np.
 ↪max(express_df['wait']),3))

        durations = np.append(durations,duration)
        duration += 1.0

    all_durations = np.append(all_durations,durations)
    delays = [delay] + (len(durations)-1)*[delay]

    all_delays = np.append(all_delays,delays)



df3_data = {
    'ExpressPass Delay': all_delays,
    'ExpressPass Duration':all_durations,
    'Aggregate Avg':agg_avg,
    'Express Avg':express_avg,
    'Normal Avg':norm_avg,
    'Aggregate Max':agg_max,
    'Express Max':express_avg,
    'Normal Max':norm_max
}
df3 = pd.DataFrame(df3_data).set_index('ExpressPass Delay')
df3
```

[22]:

```
                   ExpressPass Duration  Aggregate Avg  Express Avg  \
ExpressPass Delay
0.5                                 1.0         39.984        2.127
0.5                                 2.0         38.939        2.118
0.5                                 3.0         36.871        2.095
0.5                                 4.0         36.707        2.098
0.5                                 5.0         33.057        2.092
0.5                                 6.0         32.190        2.085
0.5                                 7.0         32.508        2.114
0.5                                 8.0         31.889        2.091
0.5                                 9.0         30.425        2.090
0.5                                10.0         31.964        2.115
```

0.5	11.0	28.812	2.081
1.0	1.0	38.860	2.088
1.0	2.0	36.646	2.107
1.0	3.0	36.422	2.125
1.0	4.0	33.935	2.111
1.0	5.0	31.944	2.108
1.0	6.0	31.127	2.094
1.0	7.0	29.495	2.087
1.0	8.0	29.475	2.112
1.0	9.0	28.837	2.112
1.0	10.0	29.052	2.056
1.0	11.0	28.615	2.092
1.5	1.0	37.859	2.126
1.5	2.0	36.598	2.121
1.5	3.0	32.838	2.084
1.5	4.0	33.879	2.107
1.5	5.0	31.548	2.099
1.5	6.0	29.678	2.098
1.5	7.0	29.966	2.100
1.5	8.0	28.124	2.052
1.5	9.0	29.338	2.078
1.5	10.0	28.629	2.119
2.0	1.0	36.901	2.125
2.0	2.0	34.435	2.097
2.0	3.0	33.182	2.115
2.0	4.0	32.209	2.121
2.0	5.0	30.622	2.102
2.0	6.0	29.797	2.088
2.0	7.0	31.155	2.109
2.0	8.0	29.646	2.127
2.0	9.0	29.053	2.114
2.0	10.0	27.804	2.086
2.5	1.0	35.389	2.115
2.5	2.0	35.017	2.103
2.5	3.0	32.338	2.108
2.5	4.0	30.996	2.118
2.5	5.0	30.175	2.103
2.5	6.0	28.277	2.103
2.5	7.0	29.142	2.072
2.5	8.0	28.369	2.105
2.5	9.0	27.144	2.115
3.0	1.0	35.035	2.105
3.0	2.0	34.859	2.158
3.0	3.0	32.579	2.110
3.0	4.0	29.722	2.126
3.0	5.0	29.342	2.097
3.0	6.0	28.602	2.068

3.0	7.0	29.801	2.105
3.0	8.0	27.831	2.119
3.0	9.0	27.894	2.077

	Normal Avg	Aggregate Max	Express Max	Normal Max
ExpressPass Delay				
0.5	45.301	64.845	2.127	64.845
0.5	44.107	63.731	2.118	63.731
0.5	41.674	61.395	2.095	61.395
0.5	41.560	61.678	2.098	61.678
0.5	37.260	53.941	2.092	53.941
0.5	36.397	54.426	2.085	54.426
0.5	36.810	55.110	2.114	55.110
0.5	35.936	56.539	2.091	56.539
0.5	34.333	55.169	2.090	55.169
0.5	35.930	54.205	2.115	54.205
0.5	32.450	50.755	2.081	50.755
1.0	43.665	64.483	2.088	64.483
1.0	41.128	60.420	2.107	60.420
1.0	40.854	60.239	2.125	60.239
1.0	38.027	57.274	2.111	57.274
1.0	35.812	52.417	2.108	52.417
1.0	34.946	53.309	2.094	53.309
1.0	33.138	54.707	2.087	54.707
1.0	32.947	53.247	2.112	53.247
1.0	32.325	52.714	2.112	52.714
1.0	32.596	57.486	2.056	57.486
1.0	32.049	53.741	2.092	53.741
1.5	42.250	61.579	2.126	61.579
1.5	40.942	59.082	2.121	59.082
1.5	36.668	52.783	2.084	52.783
1.5	37.830	55.186	2.107	55.186
1.5	35.163	57.382	2.099	57.382
1.5	33.115	52.083	2.098	52.083
1.5	33.383	56.188	2.100	56.188
1.5	31.362	55.160	2.052	55.160
1.5	32.693	57.745	2.078	57.745
1.5	31.864	58.309	2.119	58.309
2.0	40.915	60.259	2.125	60.259
2.0	38.165	58.512	2.097	58.512
2.0	36.860	60.369	2.115	60.369
2.0	35.691	61.191	2.121	61.191
2.0	33.864	57.995	2.102	57.995
2.0	33.016	58.577	2.088	58.577
2.0	34.495	62.626	2.109	62.626
2.0	32.836	61.579	2.127	61.579
2.0	32.211	62.321	2.114	62.321

2.0	30.799	57.484	2.086	57.484
2.5	39.062	60.878	2.115	60.878
2.5	38.641	63.262	2.103	63.262
2.5	35.616	60.227	2.108	60.227
2.5	34.156	59.357	2.118	59.357
2.5	33.293	61.972	2.103	61.972
2.5	31.139	57.888	2.103	57.888
2.5	32.060	57.746	2.072	57.746
2.5	31.256	63.676	2.105	63.676
2.5	29.889	61.044	2.115	61.044
3.0	38.381	65.449	2.105	65.449
3.0	38.207	63.745	2.158	63.745
3.0	35.627	62.609	2.110	62.609
3.0	32.563	63.074	2.126	63.074
3.0	32.107	59.817	2.097	59.817
3.0	31.300	60.986	2.068	60.986
3.0	32.624	65.401	2.105	65.401
3.0	30.424	62.618	2.119	62.618
3.0	30.570	65.171	2.077	65.171

[180]:
```
df3_short = df3.iloc[:, [0, 3,2,1]]
```

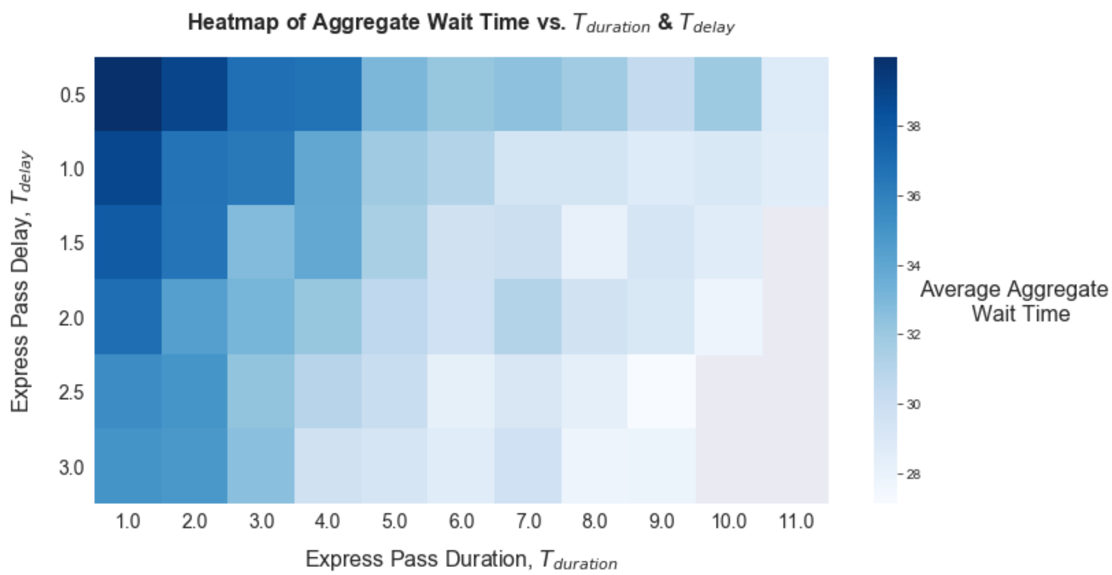
[122]:
```
df4 = df3_short.reset_index().iloc[:, [0,1,4]]
df4.head(3)
```

[122]:

	ExpressPass Delay	ExpressPass Duration	Aggregate Avg
0	0.5	1.0	39.984
1	0.5	2.0	38.939
2	0.5	3.0	36.871

[176]:
```
sns.set_style('darkgrid')
fig5b, ax5b = plt.subplots(figsize = (12, 6))

df5 = df4.pivot('ExpressPass Delay', 'ExpressPass Duration', 'Aggregate Avg')
sns.heatmap(data = df5, cmap = "Blues")

# cbar_kws={"orientation": "vertical", 'label' : 'Aggregate Wait Time',␣
↪fontsize = 16})


plt.title('Heatmap of Aggregate Wait Time vs. $T_{duration}$ & $T_{delay}$',␣
↪fontsize = 16, pad = 20, fontweight = 'semibold')
plt.xlabel('Express Pass Duration, $T_{duration}$', fontsize = 16, labelpad =␣
↪10)
plt.ylabel('Express Pass Delay, $T_{delay}$', fontsize = 16, labelpad = 15)
plt.xticks(fontsize = 14)
plt.yticks(fontsize = 14, rotation = 0)
```

```
ax5b.collections[0].colorbar.set_label("Average Aggregate \n Wait Time",␣
↪fontsize = 16,
                                        rotation = 0, labelpad = 70)
fig5b.savefig('fig5b.pdf')
```




```
[ ]:
```