Table of Contents

Project Definition & Introduction

Background & Context

The Nickelodeon Universe (NU) theme park at the Mall of America (MOA) has many different rides and attractions tailored to give their guests the best possible customer experience. The Mall of America needs to forecast the ridership in the park for each day in order to provide accurate staffing levels, and long term strategic plans. *Ridership,* in this problem, is defined as the number of times that guests at the park ride the different rides over a specified period of time. Mall of America's process for predicting park ridership at the daily level is currently not automated, and not as accurate as they would prefer it to be.

Problem Statement

The goal of this project was to improve the ridership forecast for the Nickelodeon Universe theme park at the Mall of America in such a way that the forecasting process could be easily automated. This goal was separated into three separate tasks, each with their own business use case. First, we created a static prediction model which makes more accurate forecasts for over a year in advance. This model should be used to assist management with long term strategic planning, revenue forecasting from ticket sales, and maintenance scheduling. The second task was to build a model that was more accurate, but only can predict up to fourteen days ahead of time. This forecast will be utilized by the scheduling team to attempt to minimize the number of over or under staffed days. Finally, we built an even more accurate one day prediction model that can be used for any last minute staffing changes, and to keep park supervisors vigilant of any problems that may arise due to over or under staffing. That way they may be proactive about solving those problems. Each of the three models are saved and can operationalized for real time forecasts as new information is gathered.

Data Sources

All of the data used for this analysis was provided by the Mall of America in a SQL database at the Carlson School of Management. Some of this information came from the Mall of America's internal records, and other tables were hand collected/scraped from the internet. Four different data sets were provided for this analysis:

1) Usage - This dataset contains daily overall ridership from 1st January 2013 to 10th October 2017. Later in the project the 2012 ridership information was provided in a comma separated values file.

2) School - This included the District name of the school, school year, and number of students in the school; along with the status of school ranging for days in August 2014 to June 2018. This data set was provided at the district-date level and only included days which schools were supposed to be on, but had the day off.
3) Weather - This dataset gave information regarding the weather conditions at the daily level. It included High temperature, Low temperature, Total Precipitation, Total Snow for each day. It starts with information for Jan 1st 2013 and ends with 21st October 2017.
4) Calendar - Beginning with Jan 1st 2013 and going until 31st December 2018, this data set maps each day this date last year, and 2 years prior. This data set also contained information regarding which days were specific holidays.

Technical Specifications & Solution Overview

Software & Tools

For this project all of the data cleaning was performed in R version 3.4.1; and all of the model building, including training, testing, and validation was coded in Python 3.6.1. Each tool needs to have the following additional packages installed:

R version 3.4.1	Python version 3.6.1
• RODBC • lubridate • dummies • magrittr • dplyr	• Numpy version 1.12.1 • Pandas version 0.20.1 • Scikit-learn version 0.18.1 or higher

Necessary Files & Organization Structure

All of the files required to repeat and continue our analysis are in the *Final Predictions* folder for section one team five. Each of the three analysis are broken up into their respective sub-directories: *Static Year Out Prediction*, *14 Day Prediction*, and *One Day Prediction*. Each folder contains the R, python, and csv files required to repeat the analysis, and each is numbered in the order that they need to be ran to replicate the results from scratch. An outline of the file structure, as well as an overview of each file can be found in the appendix.

Analysis

Methodology

Scikit-learn was a main tool we used for modeling, therefore we only used algorithm options that were available within that package. This includes: ridge regression, lasso regression, support vector machines, k-nearest neighbors, gradient boosted regression trees, and neural networks. After the data was prepared for the static model, we tuned the parameters of each algorithm, and validated the model, with the data before October 10th, 2016, and tested the results on all of the data after that date. The best performing algorithm was then selected as the final static model.

Our strategy for the dynamic model was to utilize and improve upon the results from the static prediction model. Instead of predicting ridership directly, we used the same algorithms to try to predict what the error from the first model would be. That error could then be added back onto the predictions of the static model to get a boost in accuracy and performance. We tried to predict the error of the results for every algorithm of the static prediction, with each of the same six algorithms that were mentioned above. The best combination was then selected as the final fourteen day, and one day final models.

Data Preparation

Static Model

The first step for the data preparation was to read the data into memory in R from the database (*read_data_1.R*). This was done using the RODBC package, and then selecting all of the observations from all of the given tables in the predictive project database. The *db* variables represents the name of the database, and *dbconn* variable represents the connection from R to the SQL server.

```
# Connect to database and read in tables
db <- "driver={SQL Server};server=CSOM-MSBA-LC5;database=MOA-Predictive;trusted_connection=true"
dbconn <-odbcDriverConnect(db)
usage <- sqlQuery(dbconn, 'SELECT * FROM Usage')
school <- sqlQuery(dbconn, 'SELECT * FROM School')
calendar <- sqlQuery(dbconn, 'SELECT * FROM Calendar')
weather <- sqlQuery(dbconn, 'SELECT * FROM Weather')
close(dbconn)
```

Next, we added the 2012 ridership information which was provided later in the project in comma separated values format. In the future, this step may not be necessary because all of the required past ridership data will be in the same place. When reading in the data from the csv, it is important to ensure that your working directory is set to the same place where

the 2012 ridership file is located. That file is then converted into the same format as the data from database, and then added on by binding the rows of the two data frames together.

```
# Read in usage 2012 data
usage2012 <- read.csv('usage2012.csv')
usage2012$Date %<>% as.Date(format = "%m/%d/%Y")
usage2012 %<>% group_by(Date) %>% summarise(Ridership = sum(Total.Uses))
usage <- rbind(usage2012,usage)
```

After all of the information was in memory in R, we started to clean and transform the data into useful features for machine learning models. The following steps are in the *transform_data_2.R* file.

Our first step was to get the ridership for each day's previous year's equivalent date using the given date mappings in the *Calendar* data. This was accomplished by joining the *Ridership* data to each of the last year, and two years ago date mapping columns, and then selecting the required columns: date and the last years/ two years ago column. These two variables are stored in data frames named *LY* for last year's ridership and *TY* for two years ago ridership.

```
## Get Previous Year Ridership ----
# Get Last Year Ridership
LY <- calendar %>%
  left_join(usage, by = c("LYDate" = "Date")) %>%
  mutate(LYRidership = Ridership) %>%
  select(Date,LYRidership)

# Get Last Year Ridership
TY <- calendar %>%
  left_join(usage, by = c("2YDate" = "Date")) %>%
  mutate(TYRidership = Ridership) %>%
  select(Date,TYRidership)
```

The next step was to get the school's information into useful features at the daily level. This was somewhat challenging because not all of the dates were included in the *School* data, only the dates which a school was supposed to be on, but had the day off. This excluded all of the weekends and summer. We first attempted to represent this information as 46 binary variables which indicated if a school was "off" on a particular day. We found this to be an inefficient representation of this information, so we compressed those variables into one feature, which represented the number of children who have school off on that day.

The first step was to find the average population of each school and store the information for later. This was done by averaging the district population over all of the years in the data. This assumes that district size does not change much from year to year.

```
## Clean up Schools Data  -----
# Save School Population information
school_pop <- school %>% group_by(DistrictName) %>%
  summarise(pop = mean(Students))
```

Then we needed to create a column for each school. This was done using the *dummy()* function in R, which takes a categorical variables (like district name) and turns each category into a new binary column. We then grouped by the date to get each row of the data to represent one day.

```
# Create dummy variables for each school that is off
school %<>% cbind(dummy(school$DistrictName))
school %<>% group_by(Date) %>%
  summarise_if(is.numeric,funs(sum)) %>%
  select(-SchoolYear) %>%
  select(-Students)
```

Since some of the dates were excluded from the original data, we had to join this data frame to a column of dates that included every single day. This introduce many rows which were filled with NA values.

```
# Add missing dates into the data
all_dates <- data.frame(seq.Date(as.Date(min(school$Date)),
                                 as.Date(max(school$Date)),by="day"))
colnames(all_dates) <- c("Date")
all_dates %<>% left_join(school)
school <- all_dates
```

We then added two binary variables onto the data frame which represented if that day was a weekend, or during the summer for the public school schedule (code in *transform_data_2.R*). To simplify the imputation of these missing values, we made the assumption that all schools start and end according to the public school schedule. This assumption holds for all but six of the districts in the data (the 6 private school districts).

We filled in the NA values by looping through every row of the data frame, and if the values in that row are *NA*, the values are replaced with a 1, if it is a weekend or summer (kids are out of school) and a 0 otherwise.

```
# Fill the missing information
for (i in seq(nrow(school)))
{
  for (j in seq(ncol(school)))
  {
    if(is.na(school[i,j]))
    {school[i,j]<- max(school$summer[i],school$weekend[i])}
  }
}
```

After this binary representation was completed, all of the unnecessary columns were removed and then each of the district variables were multiplied by their school population. Then, by summing over the columns of the school populations, we get a single variable which represents the number of students off any each day in the data set.

```
# Multiply each school by it's population
students <- data.frame(as.matrix(just_schools) %*% as.matrix(school_pop$pop))
colnames(students) <- c("pop")
school %<>% mutate(studentOff = students$pop)
```

Variables for the month and weekday were also added to the student population data. This was done so that when all of the information is combined together at the end, these features would already be in place. There is binary variable for each month and each day of the week. The code for this can be found in the *transform_data_2.R* file.

Next we cleaned the holiday information in the *calendar* data set. Similarly to the school information, the holiday information was originally represented as a set of binary variables, one for each holiday. This transformation was also done with the R *dummy()* function as illustrated below. The 'not a holiday' holidayNA column is then removed because most days are not holidays.

```
# Make Holiday Dummy Variables
holidays <- calendar %>% select(Date, Holiday)
holidays$Holiday[which(holidays$Holiday == "NULL")] <- NA
holidays %<>% cbind(dummy(holidays$Holiday))
holidays %<>% select(-holidaysNA) %>% select(-Holiday)
```

We then wanted to compress this information into two variables that only included the holidays that had a significant effect on ridership. One of the variables represented holidays with a positive impact on ridership, the other represented holidays with a significant negative impact on ridership.

We found out which holiday variables were significant through a linear regression that included just the holiday variables and ridership. The results of this linear regression can be found in the appendix. Holidays that did not have at least 10% significance were eliminated. The remaining variables were split into categories of positive holiday and negative holiday based off of their coefficients. The code for this analysis is shown below.

```
## Get Holiday Informaiton ----

# Make Holiday Dummy Variables
holidays <- calendar %>% select(Date, Holiday)
holidays$Holiday[which(holidays$Holiday == "NULL")] <- NA
holidays %<>% cbind(dummy(holidays$Holiday))
holidays %<>% select(-holidaysNA) %>% select(-Holiday)

# See which holidays matter
holidaystest <- left_join(usage,holidays) %>% select(-Date)
summary(lm(Ridership ~ .,data = holidaystest))

# Only keep significant holidays
holidays %<>% select(Date,`holidaysChristmas - Day After`,`holidaysEid Al-Fitr`,
                     `holidaysEid Al-Fitr - Day After`,`holidaysGood Friday`,`holidaysMEA Friday`,
                     `holidaysMEA Saturday`,`holidaysMLK Day`,`holidaysNew Year's Eve`,
                     `holidaysPresident's Day`,`holidaysSt. Patrick's Day`,
                     `holidaysChristmas Day`,`holidaysColumbus Day`)

# Split holidays into positive and negative holidays
holidays %<>% mutate(PosHoliday = rowSums(holidays[,2:11]),
                     NegHoliday = rowSums(holidays[,12:13]))
holidays %<>% select(Date, PosHoliday, NegHoliday)
```

The last data set to be included was the weather information. Since accurate weather forecasts are not available a year in advance, we could only include information about what the average weather is like each day of the year. We wanted to see which weather variables should be included, so like the holiday information, we did a linear regression of just the weather variables to predict ridership. The result of this regression can be found in the appendix.

```
## Get average High Temp by day of year
weather %<>% left_join(usage %>% filter(Date < '2016-10-10'))
summary(lm(Ridership ~ HighTemp_F + LowTemp_F + PrecipTotal_In + SnowTotal_In, data = weather))
weather %<>% mutate(dayofyear = yday(Date))
```

The only weather variable that had a significance of at least 10% was the high temperature. Therefore we eliminated low temperature, snow total, and precipitation total from further analysis. We took the average weather on each day of the year (days from 1 to 366). In both the regression analysis and taking the average temperature, we filtered the data to just the testing observations (before October 10th 2016) to avoid any information from the testing set being incorporated in the training variables

```
# Get the average weather for each day of the year
daily_temps <- weather %>% group_by(dayofyear) %>%
  summarise(avgTemp = mean(HighTemp_F))
```

Finally, after all of this information was cleaned into useful features at the daily level, we combined all of the data sets together to be exported to a comma separated values file for model building in python. We originally had three options for incorporating the previous year's ridership variables. Each year of previous ridership removed a year's worth of training observations for which we would not have the information for going that far back. This forced us to decide between using just last year's ridership and losing one year's worth of observation, using two year's ago ridership and losing two years of data, or not using any variables of past ridership and keeping all of the observations.

```
# Create final data table and write it to a csv
predictive_data2 <- usage %>%
  left_join(LY) %>%
  left_join(holidays) %>%
  left_join(school) %>%
  mutate(closed = ifelse(Ridership==0,1,0)) %>%
  mutate(dayofyear = yday(Date)) %>%
  na.omit()

write.csv(predictive_data2,"predictive_data_v10.01.csv",row.names = FALSE)
```

```
## Try with two years previous
predictive_data <- usage %>%
  left_join(LY) %>%
  left_join(TY) %>%
  left_join(holidays) %>%
  left_join(school) %>%
  mutate(closed = ifelse(Ridership==0,1,0)) %>%
  mutate(dayofyear = yday(Date)) %>%
  left_join(daily_temps) %>%
  na.omit()

write.csv(predictive_data,"predictive_data_v10.02.csv",row.names = FALSE)
```

We found that keeping the extra year and not adding the two years ago ridership data yielded more accurate results. This is most likely due to the two variables being highly correlated and capturing the exact same variation of ridership.

Two more variables were added while all of the data was combined. We wanted a way for the model to know, and learn what happens when the park is closed. The park was assumed closed if the ridership was equal to zero. Therefore we added a binary variable to indicate that the park was closed. Also, the day of the year (1 -366) was also added. The

day of the year was a useful feature and also needed to join the daily average temperature variable.

Dynamic Model

Since the dynamic models utilized the same variables from the static models, and we only added information, the data preparation process was much more simple. This process started with reading in the results from the comma separated values file which was written from the static prediction. We also ensure that both of the date variables are the same *Date* type since we are going to join the weather information on later.

```
# Read in the initial predictions file
static_model <- read.csv('initial_prediction.csv')
static_model$Date %<>% as.Date()
weather$Date %<>% as.Date()
```

With the dynamic models, we wanted to better capture local trends in ridership. To do this, we utilized the one week lead and lag variables from the predictions of the static model. Here is how the lead and lag variables for the 14 day prediction were added. If the predicted lag value was equal to zero, then we took the day prior because the park would've been closed on that day. Seven rows at the beginning and end which ended up with NA values of lead and lag were removed as well. The following code comes from the *transform_data_4.R* files.

```
# Add Lead and Lag
pred_data <- static_model %>% mutate(resid = Ridership - Predictions) %>%
                              mutate(lag1 = ifelse(lag(Predictions,1) == 0,lag(Predictions,2),lag(Predictions,1)),
                                     lag2 = ifelse(lag(Predictions,2) == 0,lag(Predictions,3),lag(Predictions,2)),
                                     lag3 = ifelse(lag(Predictions,3) == 0,lag(Predictions,4),lag(Predictions,3)),
                                     lag4 = ifelse(lag(Predictions,4) == 0,lag(Predictions,5),lag(Predictions,4)),
                                     lag5 = ifelse(lag(Predictions,5) == 0,lag(Predictions,6),lag(Predictions,5)),
                                     lag6 = ifelse(lag(Predictions,6) == 0,lag(Predictions,7),lag(Predictions,6)),
                                     lag7 = ifelse(lag(Predictions,7) == 0,lag(Predictions,8),lag(Predictions,7)),
                                     lead1 = lead(Predictions,1),
                                     lead2 = lead(Predictions,2),
                                     lead3 = lead(Predictions,3),
                                     lead4 = lead(Predictions,4),
                                     lead5 = lead(Predictions,5),
                                     lead6 = lead(Predictions,6),
                                     lead7 = lead(Predictions,7)) %>% na.omit()
```

The same code is used to create the lead and lag variables for the one day predictions, except the seven lag variables (*lag1, lag2, lag3 …. lag7)* the actual ridership is used instead of the predicted ridership. This can be done because when you are predicting tomorrow's ridership the information of the past seven days is now known and available.

The above code also added the residual variable to the data set, which is what will be predicted in the dynamic prediction instead of the ridership.

Finally, we add the real time high temperature from the weather data for the dynamic model. The resulting data is written to a comma separated values file to be used for model building in python.

```
# Add high temperature data
temp <- weather %>% select(Date,HighTemp_F)
pred_data %<>% left_join(temp)

write.csv(pred_data,"gradient_boost_data.csv",row.names = FALSE)
```

Model Selection & Validation

The predictive modeling process in scikit-learn was very similar for both the static and dynamic models. The only changes are the variables that are being used to predict, and being predicted. For each task, we evaluated the performance of six different numeric prediction algorithms that scikit-learn offers: ridge regression, lasso regression, support vector machine, k-nearest neighbor regression, gradient boosted trees, and neural networks.

The first step was to import the required packages and objects from numpy, pandas, and scikit-learn. Scikit-learn does not have a pre-packaged function to calculate the percentage error of a predictive model because percent error does not work in cases which the actual value being predicted is zero (like when the park is closed and ridership is zero). So we defined a percentage error function to handle this case, and assumed that when the park is closed the percent error will be zero because no predictions will be needed for those days.

```
import pandas as pd
import numpy as np

from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import GridSearchCV

from sklearn.linear_model import Lasso
from sklearn.linear_model import Ridge
from sklearn.svm import SVR
from sklearn.neural_network import MLPRegressor
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.neighbors import KNeighborsRegressor

from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error
def mean_abs_pct_err(true, preds):
    # Handle the exception for when true ridership is 0
    true = np.array(true)
    preds[true == 0] = 1
    true[true == 0] = 1
    return np.mean(np.abs((true - preds) / true))
```

The second step was to read in the data that will be used for the prediction; and split the data into X and Y variables, and train and test observations. The *Y* variable is the variable that is being predicted, while the *X* variables are the predictors. The testing data set is the last year of observations, from 2016-10-10 to 2017-10-10; while the training set includes all of the observations before 2016-10-10.

In the static model, the *Y* variable is ridership, so we need to remove ridership from the set of *X* variables. Whereas in the dynamic model we are trying to predict the residual of the static model (residual is the *Y* variable), so we need to drop both of the ridership and residual variables from the set of predictors *X*.

Static

```
riders = pd.read_csv('predictive_data_v10.01.csv',index_col=0, parse_dates=True)
Y = riders['Ridership']
X = riders.copy().drop('Ridership',axis=1)

train = riders[riders.index < '2016-10-10']
test = riders[riders.index >= '2016-10-10']

Xtrain = train.drop('Ridership',axis=1)
Xtest = test.drop('Ridership',axis=1)

Ytrain = train['Ridership']
Ytest = test['Ridership']
```

Dynamic

```
riders = pd.read_csv('gradient_boost_data.csv',index_col=0, parse_dates=True)
Y = riders['resid']
X = riders.copy().drop('Ridership',axis=1).drop('resid',axis=1)

train = riders[riders.index < '2016-10-10']
test = riders[riders.index >= '2016-10-10']

actual = test['Ridership']

Xtrain = train.drop('Ridership',axis=1).drop('resid',axis=1)
Xtest = test.drop('Ridership',axis=1).drop('resid',axis=1)

Ytrain = train['resid']
Ytest = test['resid']
```

The next step was to normalize the data so that all of the variables were on the same scale. This is required for some, but not all of the algorithms. Since some of the variables we had were binary, min-max scaling was used. This type of normalization transforms each variable to have the exact same distribution as it did before, but on a zero to one scale. This does not affect binary attributes. The same transformations needed to be done on the

training and testing sets. Therefore we trained the scikit-learn *MinMaxScaler* object with the training set, and then applied that transformation to both the training and test sets.

```
minmaxer = MinMaxScaler().fit(Xtrain)
XtrainMINMAX = minmaxer.transform(Xtrain)
XtestMINMAX = minmaxer.transform(Xtest)
XMINMAX = minmaxer.transform(X)
```

Next we had to parameter tune the six different algorithms, Each algorithm has a different set of parameters, and the parameters have different optimal levels for different predictive problems. This parameter tuning was done with grid search cross validation. Grid search cross validation searches through all possible combinations of parameters that are given, and then does a k-fold cross validation with each parameter set. This allows you to find the optimal combination of parameters for that specific algorithm on the training dataset.

The first step in this process was to define the parameter grid. The parameter grid includes the parameter names and a list of values that you would like that parameter to search through, all stored in a python dictionary. Here is an example of the support vector regression parameter grid. There are two parameters: *C* and *epsilon*.

```
# Support Vector Model
svr_params = {'C' : [90000,100000,20000,30000],
              'epsilon' : [800,1000,1200]}
```

After the parameters are defined, the next step is to initialize the scikit-learn *GridSearchCV* object with the model and parameters you want to tune. We will continue with the support vector regression algorithm example. Here *SVR()* is the scikit-learn object of algorithm we are training (support vector regressor).

```
# Support Vector Model
svr_params = {'C' : [90000,100000,20000,30000],
              'epsilon' : [800,1000,1200]}

svr_gs = GridSearchCV(SVR(),
                      scoring='neg_mean_squared_error',
                      param_grid=svr_params,cv=10)
```

Along with the type of model and parameter grid, you can also specify the number of folds in your cross validation with *cv,* and the evaluation metric that you want to consider for the 'best estimator'. We chose 10-fold cross validation and to consider mean squared error as

the primary evaluation metric. More information on evaluation metrics, cross validation, and estimators can be found in the scikit-learn documentation.

Once the *GridSearchCV* object was initialized, we can the call the *.fit* method to start the training process. This method is standard across all scikit-learn predictor objects. After the object is fit, we can the see the best parameters, best score on the cross validation, as well as use a trained best estimator that is saved to determine the performance on the testing set.

```
svr_gs.fit(XtrainMINMAX,Ytrain)
print(svr_gs.best_params_)
print("SVR Best Cross Validated RMSE: {}".format(np.sqrt(svr_gs.best_score_*-1)))

svr_predictions = svr_gs.best_estimator_.predict(XtestMINMAX)
svr_rmse = np.sqrt(mean_squared_error(Ytest,svr_predictions))
svr_mae = mean_absolute_error(Ytest,svr_predictions)
svr_mape = mean_abs_pct_err(Ytest,svr_predictions)
```

We used the model with the optimal parameters found in the cross validation to find the root mean squared error, mean absolute error, and mean percentage error on the testing set. This process is then repeated for all six of the numeric prediction algorithms. The code for this can be found in any of the predictive python files: *static_predict_3.py, inital_predict_3.py,* or *boosted_prediction_5.py*.

Results and Future Steps

Results

Finally after all the algorithms have been validated, trained, and tested, the model with the best overall performance on the test set should be the one chosen for deployment. Each model was evaluated with three performance metrics: root mean squared error, mean absolute error, and mean absolute percentage error. Comprehensive tables with the performances of the six algorithms on all three models can be found in the appendix. All of the performances were benchmarked against Mall of America's predictive performances over the same period which were: a root mean squared error (RMSE) of 4,533, a mean absolute error (MAE) of 3,480, and a mean absolute percentage error (MAPE) of 19%.

The best performing static model algorithm was a gradient boosted tree, which had a mean absolute error of 3,251, a 7% improvement over the baseline. Both the fourteen day and one day dynamic prediction models had the best performance using a simple ridge regression for the static predictors, and then using a neural network to predict the error of

the ridge regression. The fourteen day forecast had a mean absolute error of 3,071, which is a 12% improvement from the baseline. And the one day forecast had a mean absolute error of 2608, or a 25% improvement from the baseline.

Recommended Deployment

Implementation Method

The three models: static, fourteen day dynamic, and one day dynamic should all be implemented together in a standalone application. The application should be designed with the visualizations that best communicate the information required for the long term strategy objectives, and for the short term scheduling decisions. The app should be connected to the Mall of America's database to transform the data and implement the prediction algorithms in real-time. At the end of each day, the one-day prediction should be updated to give managers a heads up intuition if they will need to be proactive about staffing adjustments. Also, the two week, and one year forecasts should be plotted on a timeline to effortlessly visualize the model results. The timeline on the plot should also go back in time (for any specified amount of time after implementation) to visualize the actual versus projected ridership each day in the past. This will help ensure that the model is still a valid representation of ridership in the park.

Potential Issues

Currently all of the data cleaning is done in R, and the model building is performed in python. All of the code and the coefficients from the model will need to be translated or incorporated into a single language for an app development platform. The implemented application will need to be easily accessible by decision makers, and easily updatable by the app developer.

Deployment Risks and Mitigation

The world is constantly changing. The current models were trained to represent the data generating process of park ridership from 2014 to 2016. Events will inevitably happen in the future that may change the this process, and the patterns in daily ridership. To help mitigate this, new data should be incorporated into the training set as frequently as possible to account for new patterns in the current variables. Another way to mitigate this is to collect and incorporate new variables that would represent new variables, such as different ticket promotions that have occurred on each day.

Appendix

File System

- Final Predictions
  - Static Year Out Predictions
    - read_data_1.R
    - transform_data_2.R
    - static_predict_3.py
    - usage2012.csv
    - predictive_data_v10.01.csv
    - static_predictions.csv
  - 14 Day Prediction
    - read_data_1.R
    - transform_data_2.R
    - initial_predict_3.py
    - transform_data_4.R
    - boosted_prediction_5.py
    - usage2012.csv
    - predictive_data_v8.02.csv
    - initial_prediction.csv
    - gradient_boost_data.csv
    - dynamic14day_predictions.csv
  - One Day Prediction
    - read_data_1.R
    - transform_data_2.R
    - initial_predict_3.py
    - transform_data_4.R
    - boosted_prediction_5.py
    - usage2012.csv
    - predictive_data_v10.02.csv
    - initial_prediction.csv
    - gradient_boost_data.csv
    - dynamicONEday_predictions.csv

File Descriptions

read_data_1.R - This file uses the RODBC package in R to connect to the database and load the data into memory. This is the first step for all three models

transform_data_2.R - Does the majority of the transformations described in the data preparation section, puts all of the information required for a static model in a 'tidy' format, the *predictive_data_v10.01* csv file is exported from here to be used for modeling in python

static_predict_3.py - This file reads in the *predictive_data_v10.01* csv file and create the models for the static prediction. All of the parameter tuning, and modeling validation/training/testing is done here. The predictions for the best performing model are output to *static_predict_3.py*

usage2012.csv - Ridership data for 2012 which was extra information given as a csv later in the project

predictive_data_v10.01.csv - This csv has the 'tidy' data for the static predictions created by *transform_data_2.R* for the

static_predictions.csv - this is the csv contains our static predictions for the test data set

initial_predict_3.py - This prediction file is similar to the *static_predict.py* file, except this file is used as a base model for the two dynamic prediction models. This exports the *initial_prediction.csv*

initial_prediction.csv - Contains the initial predictions of a static model whose error will be predicted by *boosted_prediction.py*

transform_data_4.R - This script takes the *static_predictions.csv* file and adds the new variables for the dynamic prediction (for both 14 day and 1 day). This will write the *gradient_boost_data.csv* to be read into the *boosted_prediction_5.py* script.

predictive_data_v8.02.csv - This is the input for *boosted_prediction_5.py* file to predict the error of the static prediction for the 14 day model

boosted_prediction_5.py - This creates (trains/tests/validates) the dynamic prediction model, which tries to predict the error in the initial predictions

gradient_boost_data.csv - Contains the variables for the dynamic prediction (both 14 and 1 day models)

dynamic14day_predictions.csv - FInal 14 day predicted values

predictive_data_v10.02.csv - This is the input for *boosted_prediction_5.py* file to predict the error of the static prediction for the 1 day model

dynamicONEday_predictions.csv - Final 1 day predictive models

Exploratory Linear Model Results

Holidays

```
Call:
lm(formula = Ridership ~ ., data = holidaystest)

Residuals:
   Min     1Q Median     3Q    Max
-22130 -12167    412   9711  32543

Coefficients:
                                  Estimate Std. Error t value Pr(>|t|)
(Intercept)                        22800.5      320.0  71.256  < 2e-16 ***
`holidaysBlack Friday`              6982.3     6437.5   1.085 0.278239
`holidaysChristmas - Day After`    15288.5     6437.5   2.375 0.017662 *
`holidaysChristmas Day`           -22800.5     6437.5  -3.542 0.000408 ***
`holidaysChristmas Eve`            -6673.2     6437.5  -1.037 0.300058
`holidaysColumbus Day`            -11725.5     5759.6  -2.036 0.041924 *
holidaysEaster                       -648.7     5759.6  -0.113 0.910341
`holidaysEid al-Adha`                6199.5     5759.6   1.076 0.281912
`holidaysEid al-Adha - Day After`    -572.1     5759.6  -0.099 0.920891
`holidaysEid Al-Fitr`               19561.3     5759.6   3.396 0.000699 ***
`holidaysEid Al-Fitr - Day After`   13626.3     5759.6   2.366 0.018100 *
`holidaysFourth of July`             1948.7     5759.6   0.338 0.735148
`holidaysGood Friday`               26012.9     5759.6   4.516 6.72e-06 ***
holidaysHalloween                   -8999.0     6437.5  -1.398 0.162322
`holidaysLabor Day`                  7084.5     5759.6   1.230 0.218855
`holidaysMEA Friday`                18798.5     6437.5   2.920 0.003544 **
`holidaysMEA Saturday`              13536.3     5759.6   2.350 0.018876 *
`holidaysMEA Thursday`               9341.8     6437.5   1.451 0.146920
`holidaysMemorial Day`               8799.5     5759.6   1.528 0.126749
`holidaysMLK Day`                   15826.7     5759.6   2.748 0.006061 **
`holidaysMpls School Patrol`        -2562.5     5759.6  -0.445 0.656445
`holidaysNew Year's Day`             4195.5     7431.1   0.565 0.572425
`holidaysNew Year's Eve`            24722.3     6437.5   3.840 0.000127 ***
`holidaysPalm Sunday`                6083.9     5759.6   1.056 0.290979
`holidaysPresident's Day`           20690.3     5759.6   3.592 0.000337 ***
`holidaysSt. Patrick's Day`         10679.9     5759.6   1.854 0.063872 .
`holidaysSuburban School Patrol Day 1`  -3386.7     5759.6  -0.588 0.556608
`holidaysSuburban School Patrol Day 2`  -2300.9     5759.6  -0.399 0.689586
holidaysThanksgiving                -8973.2     6437.5  -1.394 0.163527
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 12860 on 1715 degrees of freedom
Multiple R-squared:  0.06865,   Adjusted R-squared:  0.05344
F-statistic: 4.515 on 28 and 1715 DF,  p-value: 7.459e-14
```

Weather

```
Call:
lm(formula = Ridership ~ HighTemp_F + LowTemp_F + PrecipTotal_In +
    SnowTotal_In, data = weather)

Residuals:
   Min     1Q Median     3Q    Max
-23097 -12498    269   9753  33443

Coefficients:
               Estimate Std. Error t value Pr(>|t|)
(Intercept)    19092.07    1167.75  16.349   <2e-16 ***
HighTemp_F       104.54      56.05   1.865   0.0624 .
LowTemp_F        -33.29      59.69  -0.558   0.5771
PrecipTotal_In   818.07    1268.30   0.645   0.5190
SnowTotal_In    -803.40     573.06  -1.402   0.1612
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 13310 on 1373 degrees of freedom
  (377 observations deleted due to missingness)
Multiple R-squared:  0.02313,   Adjusted R-squared:  0.02028
F-statistic: 8.127 on 4 and 1373 DF,  p-value: 1.78e-06
```

Model Results

Static Model

Algorithm	Best Parameters	Cross Validation RMSE	Test RMSE	Test MAE	Test MAPE
Ridge	Alpha = 0.1	5551.52	4924.70	3718.14	0.238
Lasso	Alpha = 11	5520.74	4922.68	3716.23	0.228
SVR	C =100000 epsilon=1000	5432.98	4817.05	3420.40	0.240
Neural Net	Not Converging	NA	NA	NA	NA
Grad Boost Tree	Learning_rate=0.05 max_depth=2 n_estimators=400	5122.22	4394.63	3251.78	0.202
K-NN	n_neighbors=20 weights=distance	5690.07	5516.72	3904.79	0.220

Dynamic 14 day model

Algorithm	Best Parameters	Test RMSE	Test MAE	Test MAPE
Ridge	Alpha = 0.2	4712.25	3562.89	0.223
Lasso	Alpha = 9	4734.29	3581.85	0.222
SVR	C = .0001, epsilon = 4500	5017.59	3838.26	0.249
Neural Net	Hidden layer = 40,40 alpha=.00001, Activation = relu Learning rate = .001, Tol = .000001	4173.80	3071.13	0.190
Grad Boost Tree	Learning_rate = 0.005, max_depth = 2, N_estimators = 300	4644.48	3508.90	0.216
K-NN	N_neighbors = 200, weights = Uniform	4957.32	3715.33	0.233

Dynamic 1 day model

Algorithm	Best Parameters	Test RMSE	Test MAE	Test MAPE
Ridge	Alpha = 0.2	4213.29	3092.80	0.193
Lasso	Alpha = 6	4221.44	3083.74	0.191
SVR	C = 0.0001, epsilon = 4500	5017.59	3838.26	0.249
Neural Net	Hidden layer = 40,40 activation:relu, alpha = .00001 Tolerance = .000001	3534.08	2608.22	0.151
Grad Boost Tree	Learning_rate = 0.04, max_depth = 2, n_estimators = 800	4218.26	3102.40	0.192
K-NN	n_neighbors = 200 Weights = uniform	4923.98	3685.75	0.229