Taxi and Weather Data Project

CIS 4130 Big Data Technologies

Jose Garcia

jose.mauricio0226@gmail.com

Table of Contents:

Project Overview

Description

The project analyzes and predicts the tipping behavior of New York City taxi riders using historical trip data and weather conditions. The goal is to explore how external factors, such as weather, time of day, and trip characteristics, influence this tipping behavior. This analysis will help identify patterns in customer behavior and provide actionable insights for stakeholders like the Taxi and Limousine Commission (TLC).

Data Sources:

1) New York City Taxi and Limousine Commission (TLC) Yellow Taxi Trip Records
    a) https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page
2) Weather Data from Visual Crossing
    a) https://www.visualcrossing.com/weather/weather-data-services

Data Description:

The NYC Yellow Taxi Trip Data provides general information about taxi trips in New York City. The dataset contains location identifiers for pickup and dropoff points. It also includes fare components, including fare amount, payment type, taxes, surcharges, and tolls.

The Weather Data from Visual Crossing offers detailed weather data for each day. It includes temperature data like the daily average, min, max, and feels like temperature. There is also data on precipitation, snow, winds, and humidity.

Models:

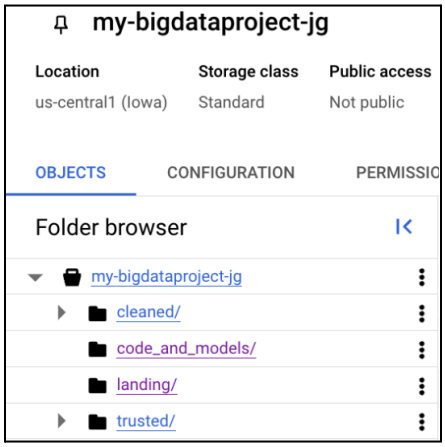
This project develops a Logistic Regression model to predict whether a tip will be "good" (≥15%) or "bad" (<15%) based on various input features. The model uses Logistic

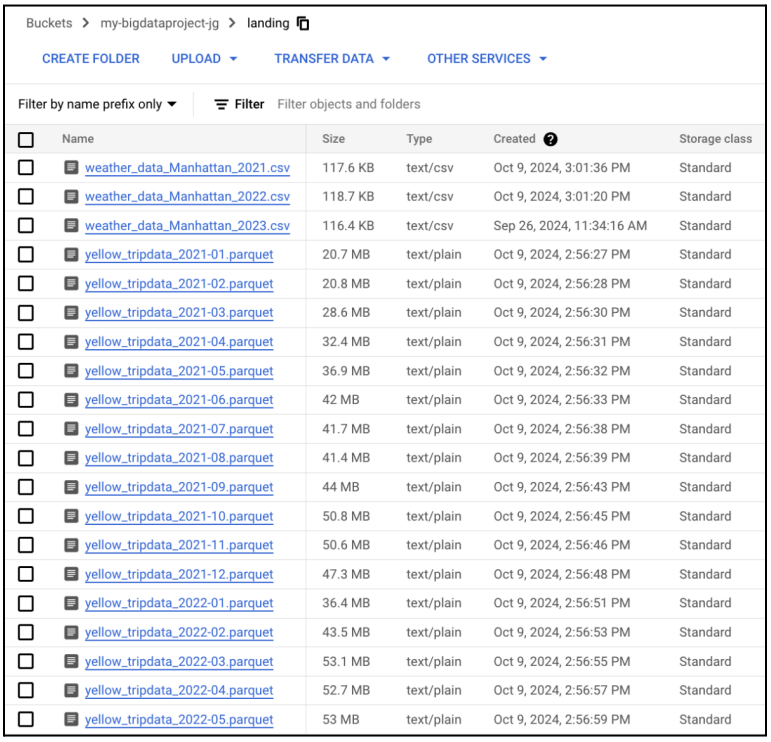
Regression, and the evaluation metrics include AUC (Area Under Curve), accuracy, precision, recall, and F1 score.

Data Acquisition

Google Cloud Storage:

For cloud storage, the following bucket and folder format was implemented:





VM and Python Set Up:

To see which VM commands are needed for this project, refer to the *set_up_vm.sh* script in Appendix A.

Extracting Taxi Data:

The complete *extracting_taxi_data.py* code is in Appendix A.

To get the taxi data, we used the NYC Taxi & Limousine Commission's data repository (click here). The data is stored in parquet files, each representing one month of data. We extract data for the years 2021 to 2023.

The script uses the requests library to download the parquet files. It iterates through each year (2021-2023) and month (01-12) and constructs the full URL for the file. It extracts the file "yellow_tripdata_YYYY-MM.parquet " for each iteration and stores it in GCS.

Extracting Taxi Zones Data:

The complete *extracting_taxi_zones_data.py* code is in Appendix A.

To get the taxi zone data, we will use data from NYC Open Data(click here). This data is needed to The data is stored in a single CSV file. The program retrieves the data from the given URL and uploads it to GCS.

Extracting Weather Data:

The complete *extracting_weather_data.py* code is in Appendix A.

To retrieve the weather data, we will use Visual Crossing. You must create an account and obtain an API Key (click here). We will retrieve daily weather data for the five boroughs from 2021 to 2023. We must extract 5,475 records; free accounts are limited to 1,000 daily records. This script is structured to extract weather data for a single borough and a specific year.

The program asks you to choose a borough and year from a list. The Visual Crossing API will retrieve weather data based on the location and date range provided(Timeline Weather API). In the parameters, we specify that daily data should be returned in CSV format. Finally, we process the CSV and save it to GCS.

Exploratory Data Analysis and Data Cleaning

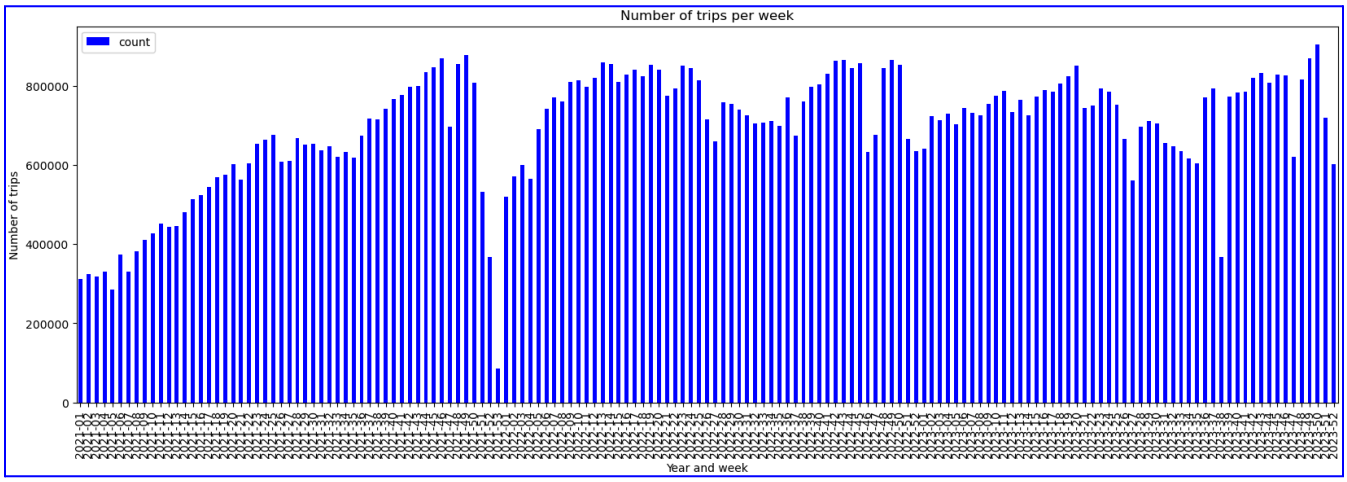
Exploratory Data Analysis

Weather Data EDA

Each file from the weather data has 365 records, as expected. There are 33 columns, and most are categorized in the correct data type. There are a lot of records with missing *preciptype* and *severerisk* attributes. This is because some of these files have columns that have a null value by default. Overall, there is

Taxi Data EDA

There are 108,870,632 taxi records from 2021 to 2023.

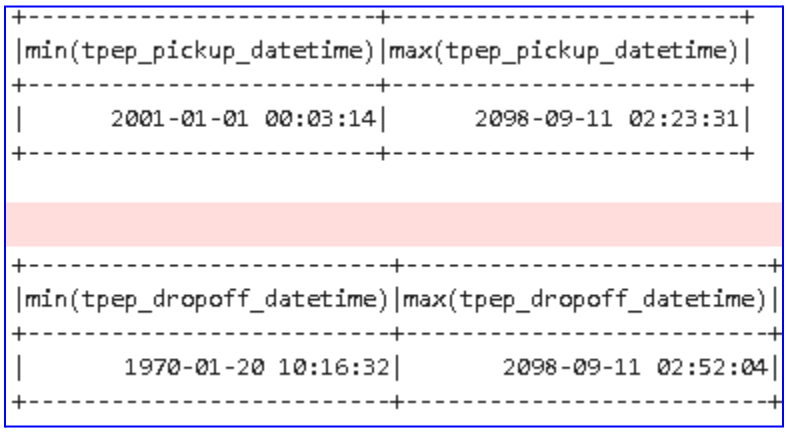


Taxi Data Frame:

Column Name	Data Type	New Type	Empty Records
VendorID	int		
tpep_pickup_datetime	datetime		
tpep_dropoff_datetime	datetime		
passenger_count	double	int	4156354
trip_distance	double		

RatecodeID	double	int	4156354
store_and_fwd_flag	string		4156354
PULocationID	long		
DOLocationID	long		
payment_type	long		
fare_amount	double		
extra	double		
mta_tax	double		
tip_amount	double		
tolls_amount	double		
improvement_surcharge	double		4156354
total_amount	double		8319077
congestion_surcharge	double		
airport_fee	double		

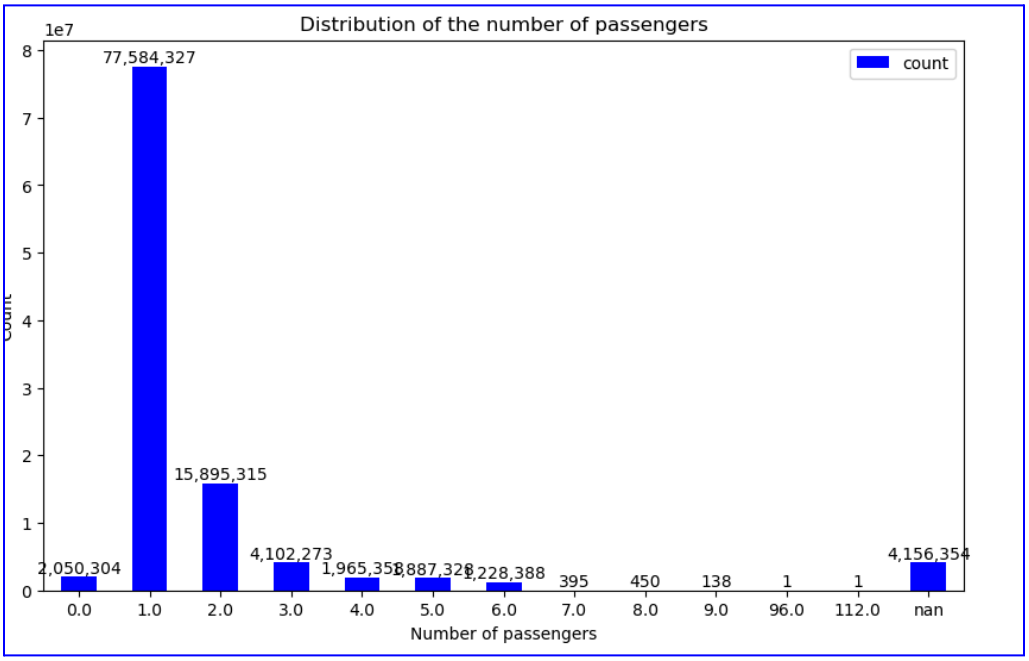
The *pickup* and *dropoff datetime* columns have some extreme outliers. Both columns' expected min and max should be similar to 2021-1-1 and 2024-1-1.



```
+------------------------+------------------------+
|min(tpep_pickup_datetime)|max(tpep_pickup_datetime)|
+------------------------+------------------------+
|      2001-01-01 00:03:14|      2098-09-11 02:23:31|
+------------------------+------------------------+


+-------------------------+-------------------------+
|min(tpep_dropoff_datetime)|max(tpep_dropoff_datetime)|
+-------------------------+-------------------------+
|       1970-01-20 10:16:32|       2098-09-11 02:52:04|
+-------------------------+-------------------------+
```

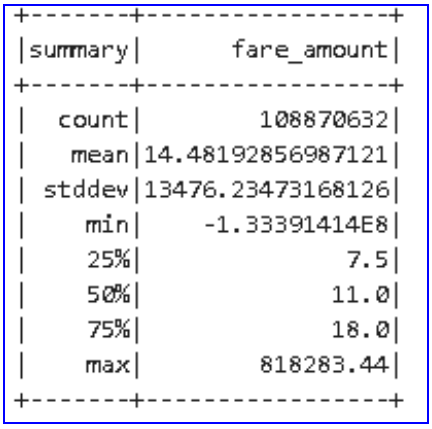
The passenger count contains records with zero values: 2,050,304. Most passenger counts are between 1 and 6.



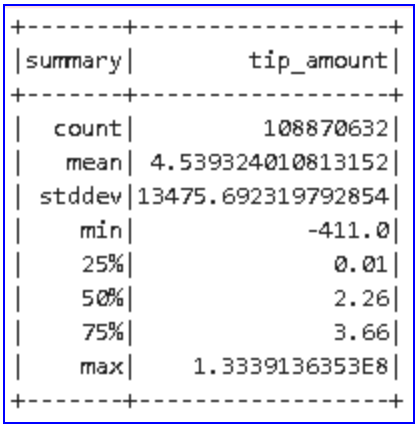
In the *trip distance*, there are 1,755,328 with a zero value. The mean trip distance is 5.57 miles, but there are many outliers with large distances.

summary	trip_distance
count	108870632
mean	5.574651813539291
stddev	538.2452378045958
min	0.0
25%	1.09
50%	1.82
75%	3.43
max	389678.46

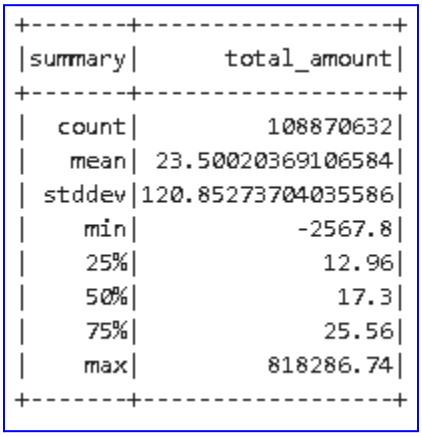
The *fare amount*, *total amount*, and *tip amount* columns contain records with negative values. This is because some are disputed trips, meaning there are duplicate trip records with positive and negative amounts.



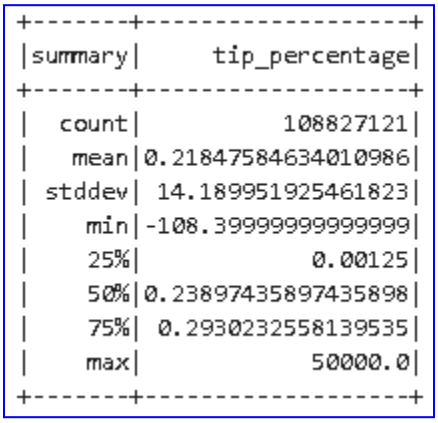
summary	fare_amount
count	108870632
mean	14.48192856987121
stddev	13476.23473168126
min	-1.33391414E8
25%	7.5
50%	11.0
75%	18.0
max	818283.44



summary	tip_amount
count	108870632
mean	4.539324010813152
stddev	13475.692319792854
min	-411.0
25%	0.01
50%	2.26
75%	3.66
max	1.3339136353E8



summary	total_amount
count	108870632
mean	23.50020369106584
stddev	120.85273704035586
min	-2567.8
25%	12.96
50%	17.3
75%	25.56
max	818286.74



summary	tip_percentage
count	108827121
mean	0.21847584634010986
stddev	14.189951925461823
min	-108.39999999999999
25%	0.00125
50%	0.23897435897435898
75%	0.2930232558139535
max	50000.0

Data Cleaning

Weather Data Cleaning

The majority of the weather data has already been formatted and cleaned. We start by keeping the columns we need. The default value for *preciptype* is null, so we filled it to “None”. Then, we changed the column data types to the appropriate types. Finally, we drop any empty records and upload the data to the clean folder in the storage bucket.

Taxi Data Cleaning

Column Name	New Type	Field Limitations
pickup_datetime	datetime	Min: 2021-1-1 Max: 2024-1-1

dropoff_datetime	datetime	Min: 2021-1-1 Max: 2024-1-1
passenger_count	int	1-7
trip_distance	double	0 < x <= 60
RatecodeID	int	1, 2, 6,99
PULocationID	int	
DOLocationID	int	
payment_type (dropped)	int	!= 6
fare_amount	double	3 < x < 250
tip_amount	double	0 <= x < 150
tip_percentage	double	tip_amount / total_amount * 100 < 1.5
total_amount	double	3 < x < 400

For the *datetime* columns, we must remove any records that are not within the year range of 2021 to 2023. We remove any voided taxi records by excluding the *payment types* equal to 6. For the *RatecodeID*, we fill empty values to 99 (“Unknown”) and only keep the IDs equal to 1, 2, 6, and 99. We fill the empty records for *passenger count* to 0 and keep values less than or equal to 7.

We filter *fare amount* to the range [3, 250), *tip amount* to [0, 150), and *total amount* to (3, 400). We Create a new column, *tip percentage,* and set it as a double. We filter *trip distance* to (0, 60].

Finally, we export the spark taxi data frame as a parquet file to the GCS bucket.

Taxi Zones Cleaning

For the taxi zone data, we remove unused columns and set the other columns to the proper data types. We drop any empty records and upload the file to the GCS bucket.

Future Challenges

For the next milestone, we still need to replace the "*LocationIDs*" with the corresponding boroughs in the taxi data. In addition, we will need to decide how to filter the "*trip_distance*" column. We must also decide how to feature engineer the "conditions" attribute in the weather data.

Feature Engineering and Modeling

Feature Engineering

For the feature engineering and modeling, we will be utilizing Pyspark. We start by reading all the taxi, weather, and taxi zone files.

Weather Dataframe

For the weather data, we will have to rename some columns: *borough* to *weather_borough* and *snow* to *snow_precip*. Then, we drop the columns that are not needed.

```
datetime: string (nullable = true)
temp: double (nullable = true)
feelslike: double (nullable = true)
precip: double (nullable = true)
snow_precip: double (nullable = true)
snowdepth: double (nullable = true)
windspeed: double (nullable = true)
uvindex: long (nullable = true)
conditions: string (nullable = true)
weather_borough: string (nullable = true)
```

Taxi Dataframe

For the taxi data frame, we will have to do some filtering and column changes to the cleaned taxi data frame:

```
pickup_datetime: timestamp_ntz (nullable = true)
dropoff_datetime: timestamp_ntz (nullable = true)
passenger_count: integer (nullable = true)
trip_distance: double (nullable = true)
RatecodeID: integer (nullable = true)
PULocationID: long (nullable = true)
DOLocationID: long (nullable = true)
fare_amount: double (nullable = true)
tip_amount: double (nullable = true)
total_amount: double (nullable = true)
tip_percentage: double (nullable = true)
```

```
passenger_count: integer (nullable = true)
trip_distance: double (nullable = true)
fare_amount: double (nullable = true)
tip_percentage: double (nullable = true)
pickup_date: date (nullable = true)
time_of_day: string (nullable = false)
group_size: integer (nullable = false)
PUBorough: string (nullable = true)
DOBorough: string (nullable = true)
```

Original vs. Combined data frame

We replace the pickup_datetime with a *pickup_date* and the *time_of_day* (morning, afternoon, night) columns. The *passenger_count* will be replaced by group_size, which

is 1 when passenger_count is 1, 2 when passenger_count is 2, and 3 when the passenger_count is 3 or greater.

Next, we will further filter the data to minimize the amount of outliers. First, we remove rows with incomplete data using *RatecodeID* and *passenger_count*. For a standard taxi fare, there is an initial charge of $3, plus $0.70 for every 0.2 miles and a $0.50 MTA State Surcharge. So, we remove records with the following conditions:

1. *RatecodeID* = 99
2. *passenger_count* = 0
3. *fare_amount* < 3.70.
4. *total_amount* <= 4.20
5. *trip_distance* < 0.2 miles

After filtering the data, we combine the taxi and taxi_zone data using the *PULocationID* and *DOLocationID*. We replace these columns with the *PUBorough* and *DOBorough*. We remove any records where either borough is “EWR”.

Finally, we combine the data frames and end up with the following columns:

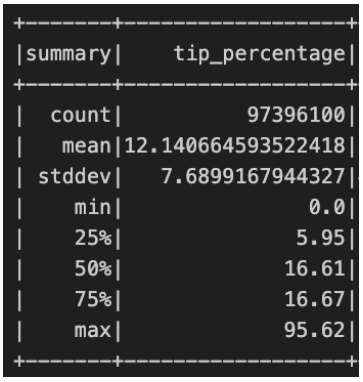
```
passenger_count: integer (nullable = true)
trip_distance: double (nullable = true)
fare_amount: double (nullable = true)
tip_percentage: double (nullable = true)
pickup_date: date (nullable = true)
time_of_day: string (nullable = false)
group_size: integer (nullable = false)
PUBorough: string (nullable = true)
DOBorough: string (nullable = true)
temp: double (nullable = true)
feelslike: double (nullable = true)
precip: double (nullable = true)
snow_precip: double (nullable = true)
snowdepth: double (nullable = true)
windspeed: double (nullable = true)
uvindex: long (nullable = true)
conditions: string (nullable = true)
```

Feature engineering chart

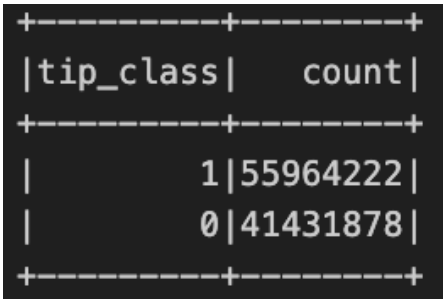
	Original Column	Format	Type	Features	Type	Method
Taxi Data	pickup_datetime	YYYY-MM-DD H:MM:SS	date time	month	Int: 1-12	Encode Directly
				dayofweek	Int: 1-7	Encode Directly
				weekend	Int: 0 or 1	Encode Directly
				holiday	Int: 0 or 1	Encode Directly
				time_of_day	String	StringIndexer
	PULocationID	NYC Taxi Zone data: Location ID Zone (neighborhood) Borough	int: 8 Astoria Park Queens	PUBorough	Int: 1-5 Queens - 2	StringIndexer
	DOLocationID			DUBorough		StringIndexer
	passenger_count	Range: 1 - 7	int	group_size	int: 1-3	Encode Directly
	trip_distance			trip_distance		Encode Directly
	fare_amount			fare_amount		Encode Directly
	tip_percentage		double	tip_class		TARGET VARIABLE
Weather	name	Borough		borough	Int: 1-5 Queens - 2	StringIndexer
	datetime	MM/DD/YYYY	date time	Datetime Features (listed above)		
	temp	Mean Temp in F	double	temp		MinMaxScaler
	feelslike	Mean Feels Like Temp in F	double	feelslike		MinMaxScaler

Data	precip	Precipitation in inches	double	precip		MinMaxScaler
	snow	Snow in inches	double	snow_precip		MinMaxScaler
	snowdepth	Snow depth in inches	double	snowdepth		MinMaxScaler
	windspeed	wind speed in mph	double	windspeed		MinMaxScaler
	uvindex	UV Index	int	uvindex		MinMaxScaler
	conditions	Short text about the weather: Snow, Rain, Partially cloudy	string	column for each condition	Int: 0 or 1	UDF

Feature engineering



summary	tip_percentage
count	97396100
mean	12.140664593522418
stddev	7.6899167944327
min	0.0
25%	5.95
50%	16.61
75%	16.67
max	95.62



tip_class	count
1	55964222
0	41431878

First, we create a *tip_class*, which is 1 when the tip percentage is 15 or greater and otherwise 0. Then we turn the pickup_date into a month, *dayofweek*, weekend, and holiday features. Next, we create binary features *Rain*, *Snow*, *Clear*, and *Ice* using the conditions column. We use String Indexer and One Hot Encoding for *PUBorough*, *DOBorough,* and *time_of_day*. For the rest of the weather conditions, we Vector Assemble them and then use Min Max Scaler.

Modeling

After Vector Assembling the input columns, we split the data into 70% training and 30% testing. We use Logistic Regression with *maxIter*=10, *regParam*=0.1, and *elasticNetParam*=0.8. Our Logistic Regression pipeline stages are the following:

1. *indexer,*
2. *encoder,*
3. *assembler_to_scale,*
4. *scaler,*
5. *assembler,*
6. *lr* (LogisticRegression)

We build a ParamGridBuilder with the following:

1. *regParam* = [0.0, 0.2, 0.4, 0.6, 0.8, 1.0]
2. *elasticNetParam* = [0, 0.5, 1]

In total, we will be testing 18 models. We will be using the Binary Classification Evaluator and areaUnderROC as the metric. Then, we create the Cross-Validator and fit all the models.

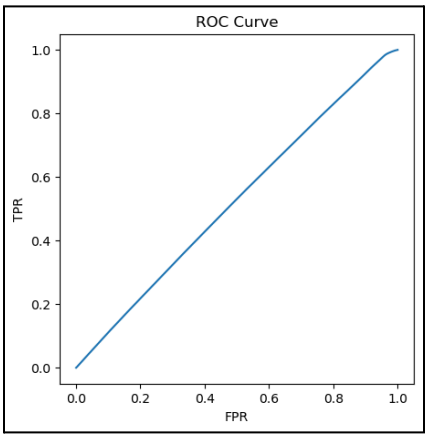
The biggest challenge of this model is the processing requirement for this model. The run time of this program was about 2-3 hours. We need four worker nodes with a total of ~100 GB in YARD Memory to test the model.

Data Visualizing

Results

The AUC indicates that the model's ability to distinguish between classes is slightly better than random chance. In most instances, the model predicts that the tip class would be a good tip. This led to many false positives and affected the precision to 41.9%. Overall, there is room for improvement with the model. For future steps, I will continue to filter out outliers and remove features with small coefficients.

AUC: 0.5239737326719903



```
+---------+----------+--------+
|tip_class|prediction|   count|
+---------+----------+--------+
|        1|       1.0|16432056|
|        0|       0.0|  563152|
|        0|       1.0|11863179|
|        1|       0.0|  361321|
+---------+----------+--------+
```

Accuracy: 0.4183648926265793

Precision: 0.41926419766437706

Recall: 0.9546807500942958

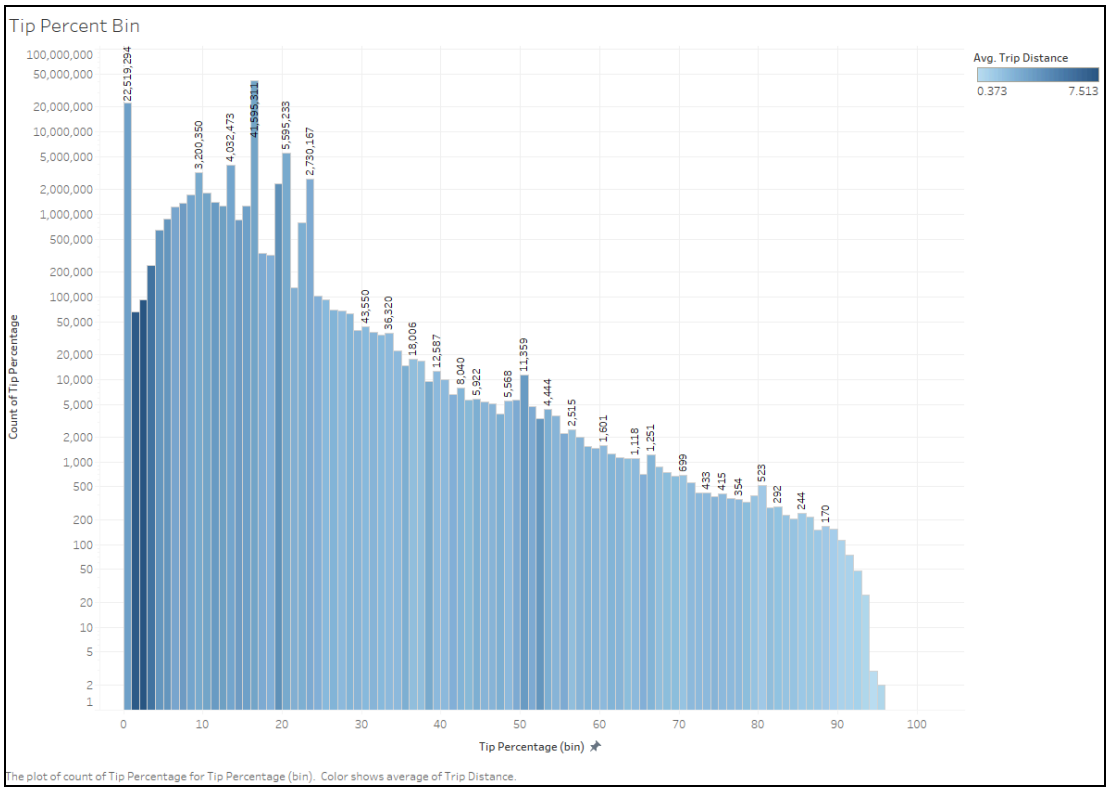
F1 Score: 0.5826484669081734

Features	Coefficients
PUBorough_encoded	0.1026303781
DOBorough_encoded	-0.08122394549

time_of_day_encoded	-0.1408812442
month	-1.314957822
dayofweek	0.1790463238
weekend	-0.190075341
holiday	0.11234117
trip_distance	-1.134378806
group_size	-0.07739401525
fare_amount	-0.1181883706
Rain	0.005569633259
Snow	0.0009186153853
Clear	-0.01235878256
Ice	-0.04407484658
scaled_features	0.01215823529

Tip Percentage

Frequency of Tip Percentages



There are 22.5 million records with 0% tip, and the average trip distance is 3.41 miles. A 0% tip can also signify that the rider paid a cash tip. The most popular tip percentage is 16.67%, with 41.6 million records, followed by 20%, 13%, 9%, and 23%. The graph also shows that records with higher tip percentages have smaller average trip distances.

Borough Columns

Features	Coefficients
PUBorough_encoded	0.1026303781
DOBorough_encoded	-0.08122394549

Combinations of Pickup and Dropoff locations

PU / DO Borough Count

DO Borough	Bronx	Brooklyn	Manhattan	Queens	Staten Island	Grand Total
Bronx	23,816	2,157	343,654	197,472	154	567,253
Brooklyn	1,323	194,057	2,079,507	1,328,405	844	3,604,136
Manhattan	19,492	162,458	83,524,757	4,885,006	424	88,592,137
Queens	3,084	27,299	2,932,832	1,646,819	563	4,610,597
Staten Island	124	1,098	8,680	11,342	733	21,977
Grand Total	47,839	387,069	88,889,430	8,069,044	2,718	97,396,100

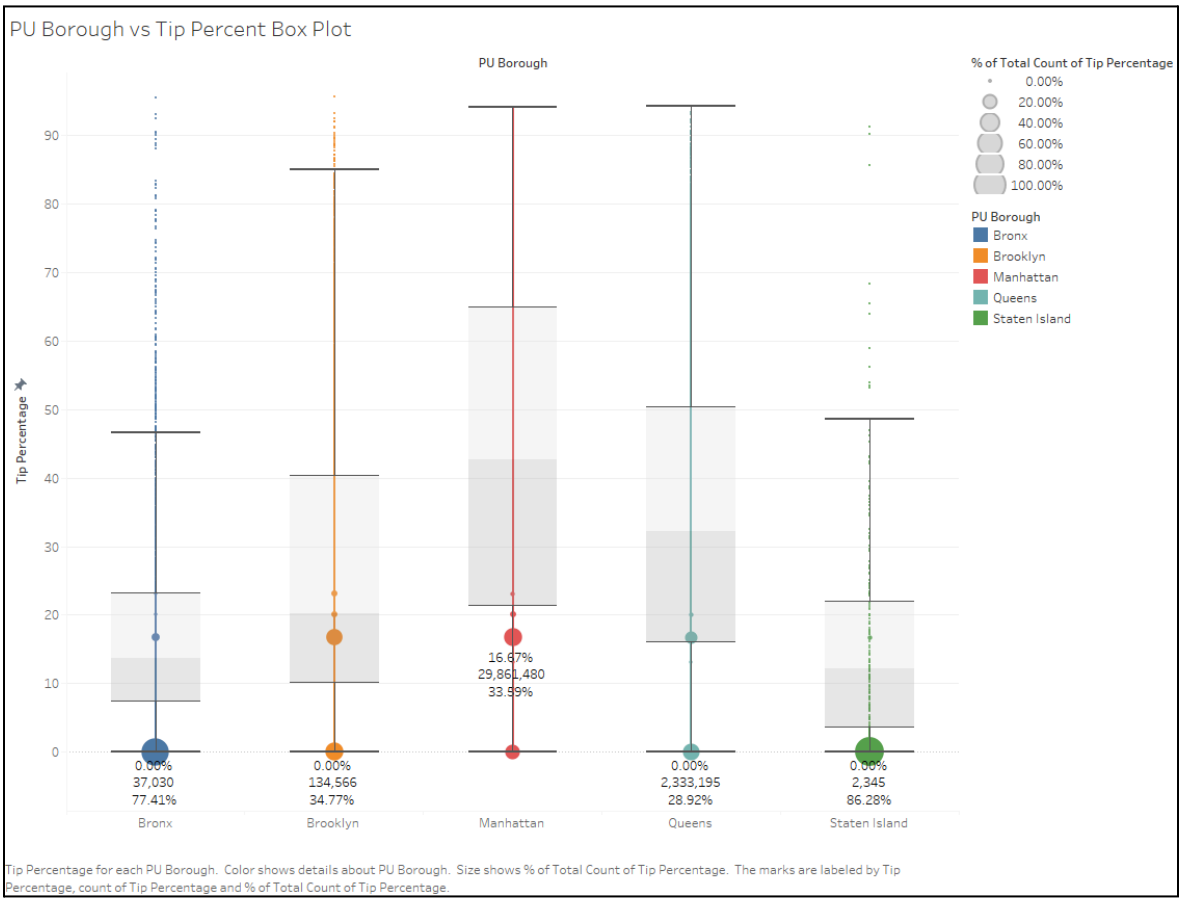
PU Borough

Count of PU Borough
124 10,000,000

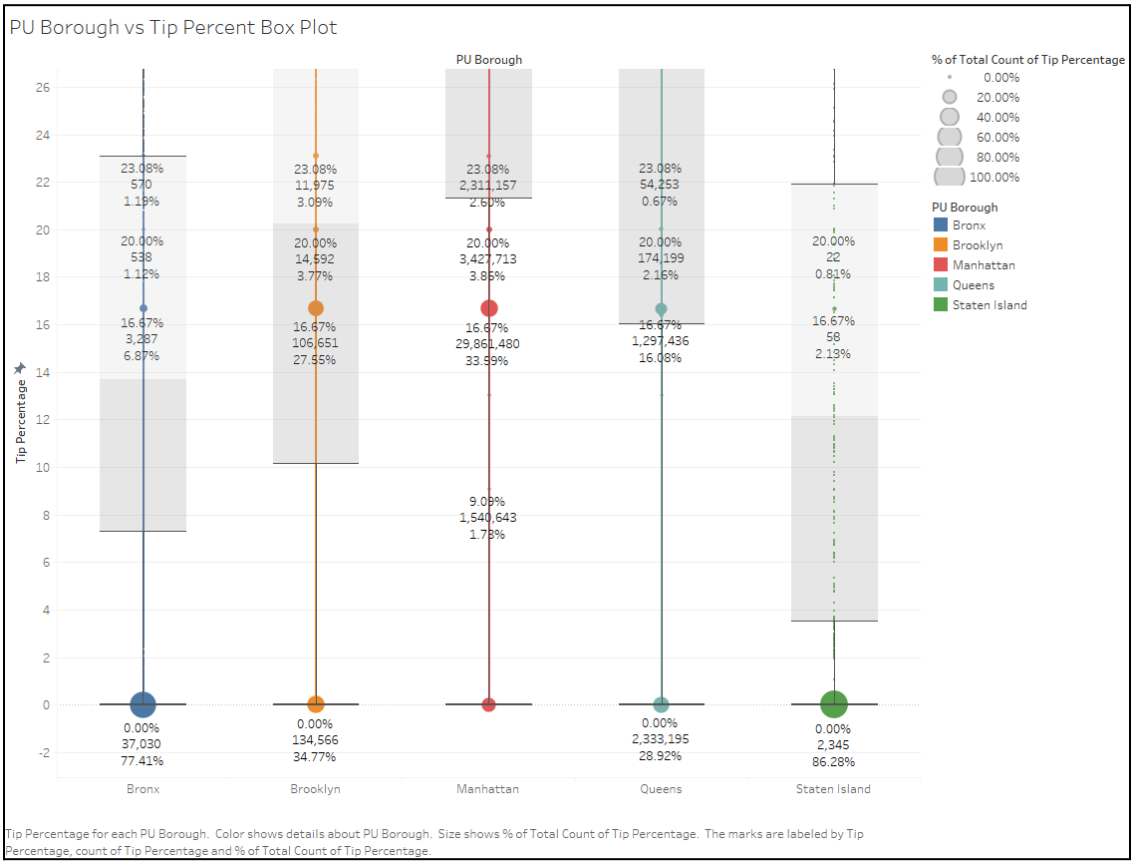
Count of PU Borough broken down by PU Borough vs. DO Borough. Color shows count of PU Borough. The marks are labeled by count of PU Borough.

Most taxi trip records indicated pick-ups and drop-offs in Manhattan or Queens.

Tip Percent Box Plot by Pickup Borough

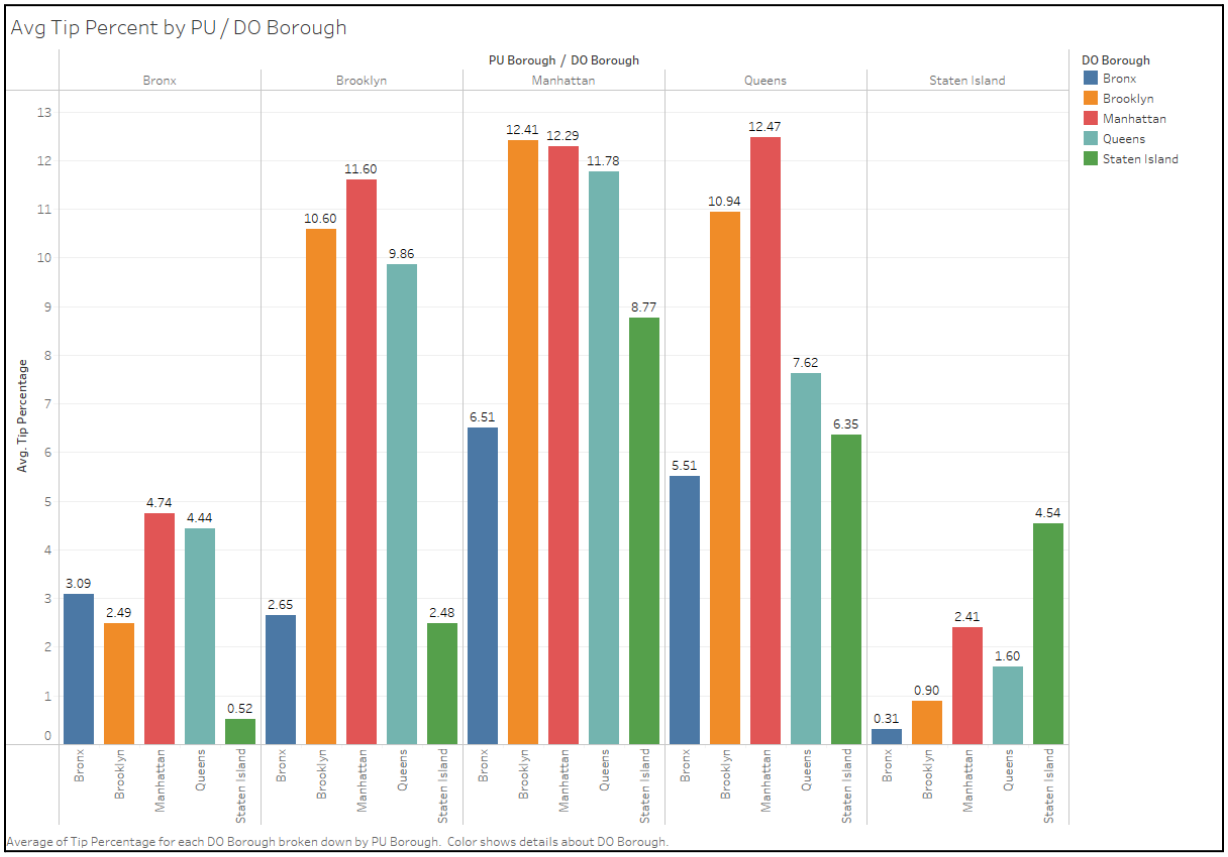


Brooklyn has the largest IQR Range, but 34% of the records have a 0 tip percentage. The Bronx and Staten Island have similar tip percentage distributions. Their IQR range is relatively lower, and about 80% of their records have a 0 tip percentage. Manhattan has the largest median of 42, and 33.6% of its records show a 16.67 tip percentage.



Looking further into the Manhattan records, we see that many records leave either a 23%, 20%, 16.67%, or 9% tip. Other boroughs follow this pattern, which could indicate that they are default options in taxis.

Average Tip Percent by Pickup and Dropoff Combination

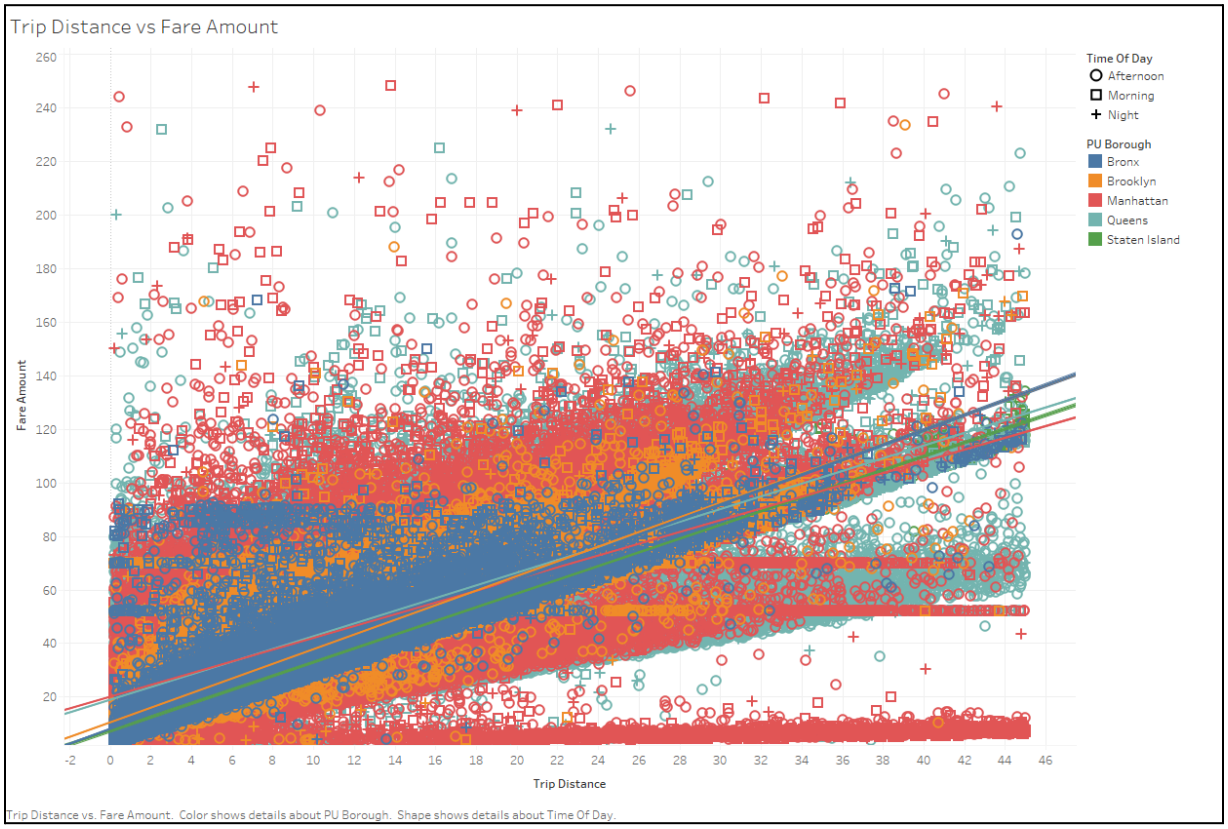


On average, pick-ups in the Bronx and Staten Island yield the lowest tip percentage, regardless of the drop-off location. In contrast, pick-ups from Manhattan and Queens have the highest average tip percentage. The tip percentage is usually higher when the drop-off location is in Manhattan.

Trip Distance & Fare Amount

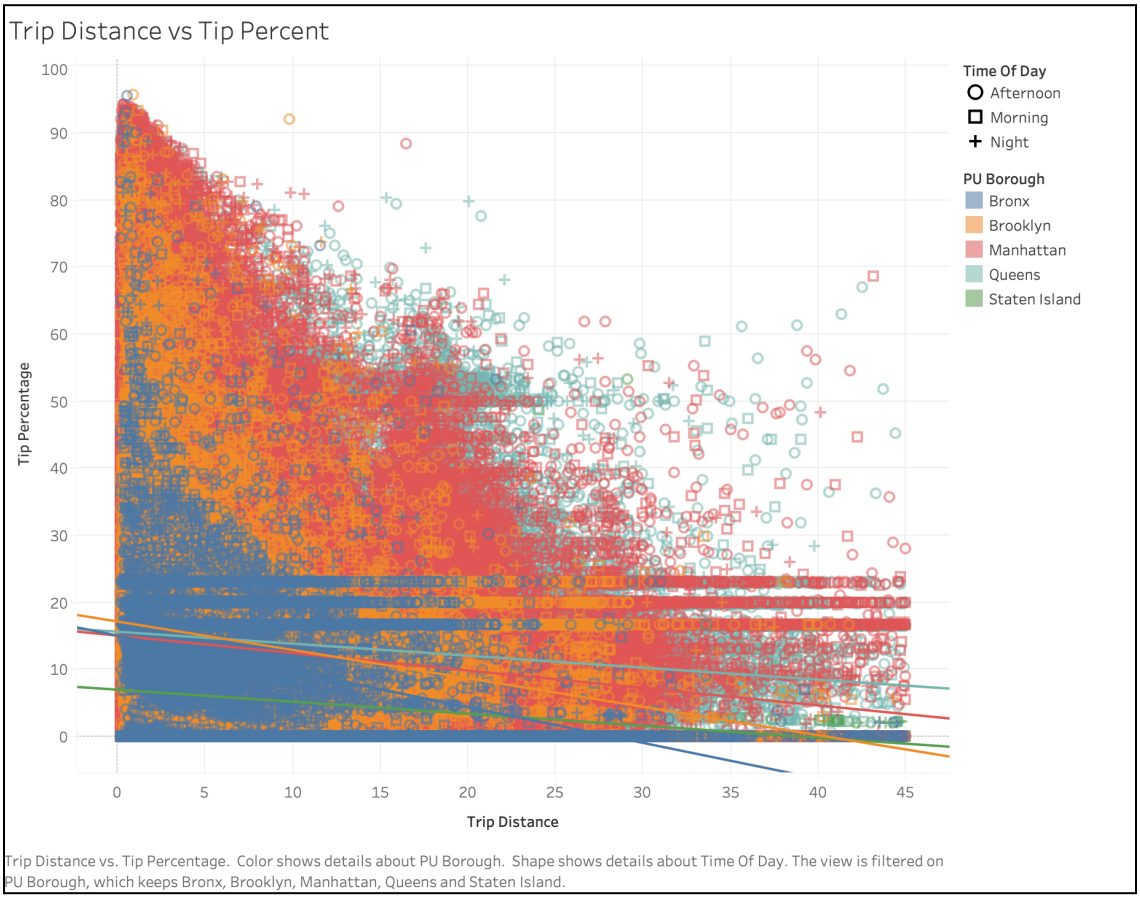
Features	Coefficients
trip_distance	-1.134378806
fare_amount	-0.1181883706

Trip Distance vs. Fare Amount



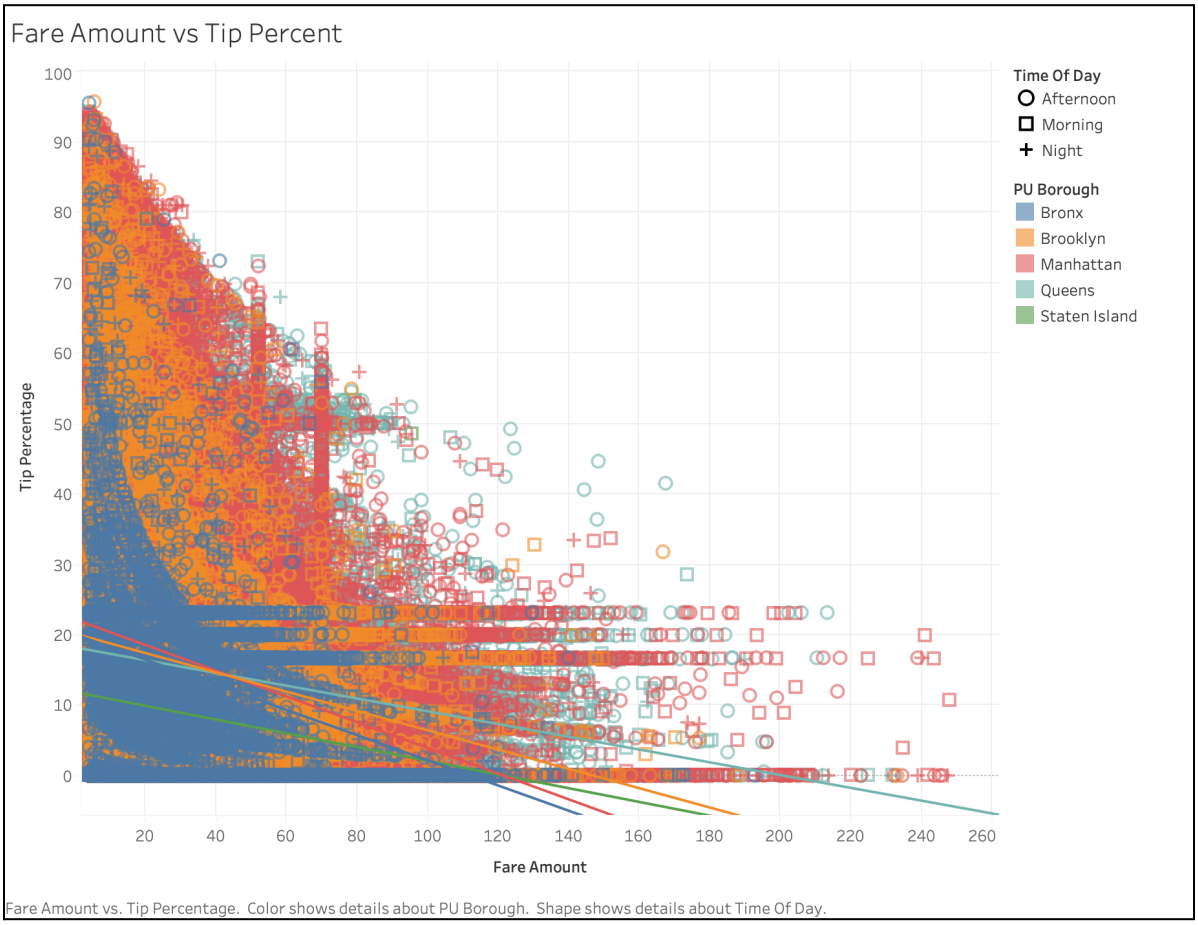
This graph shows that the relationship between trip distance and fare amount, as expected, is a positive correlation. We can see that there are still many outliers within the data set, such as trips with a short distance but unusually large fare amount or trips with a long distance but low fare amount. There are records with a $50 or $70 fare, even with increasing trip distance.

Trip Distance vs. Tip Percent



Trip Distance vs. Tip Percentage. Color shows details about PU Borough. Shape shows details about Time Of Day. The view is filtered on PU Borough, which keeps Bronx, Brooklyn, Manhattan, Queens and Staten Island.

Fare Amount vs. Tip Percent

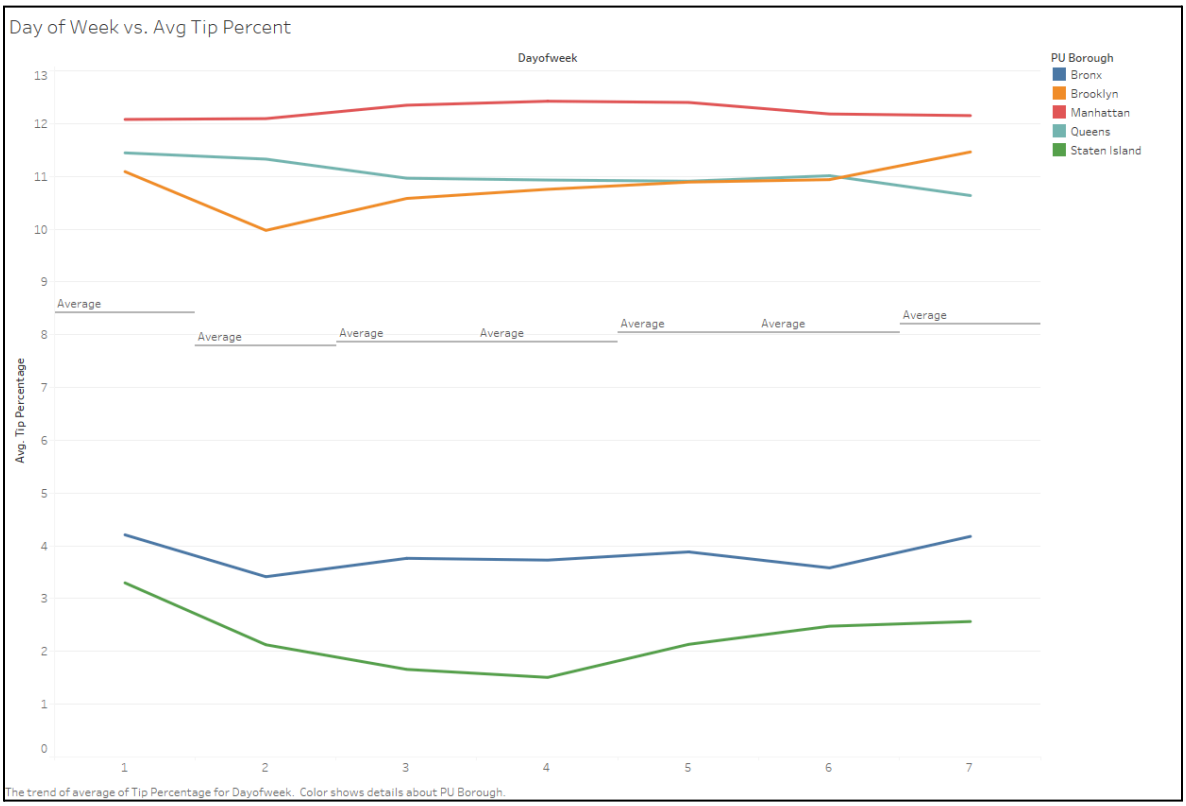


The trip distance and fare amount negatively correlate with the tip percentage. Since fare amount and trip distance are correlated, we must remove one from our model.

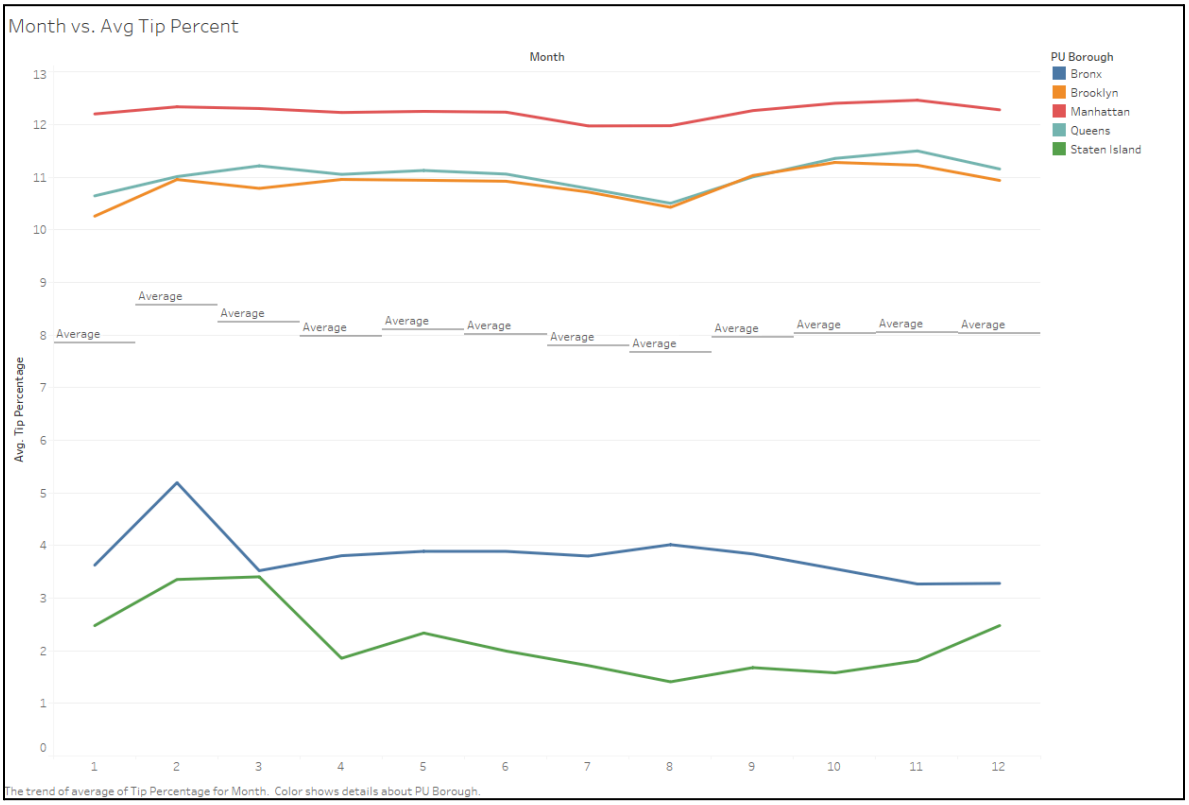
Data Features

Features	Coefficients
month	-1.314957822
dayofweek	0.1790463238
weekend	-0.190075341
holiday	0.11234117

Day of Week vs. Avg Tip Percent

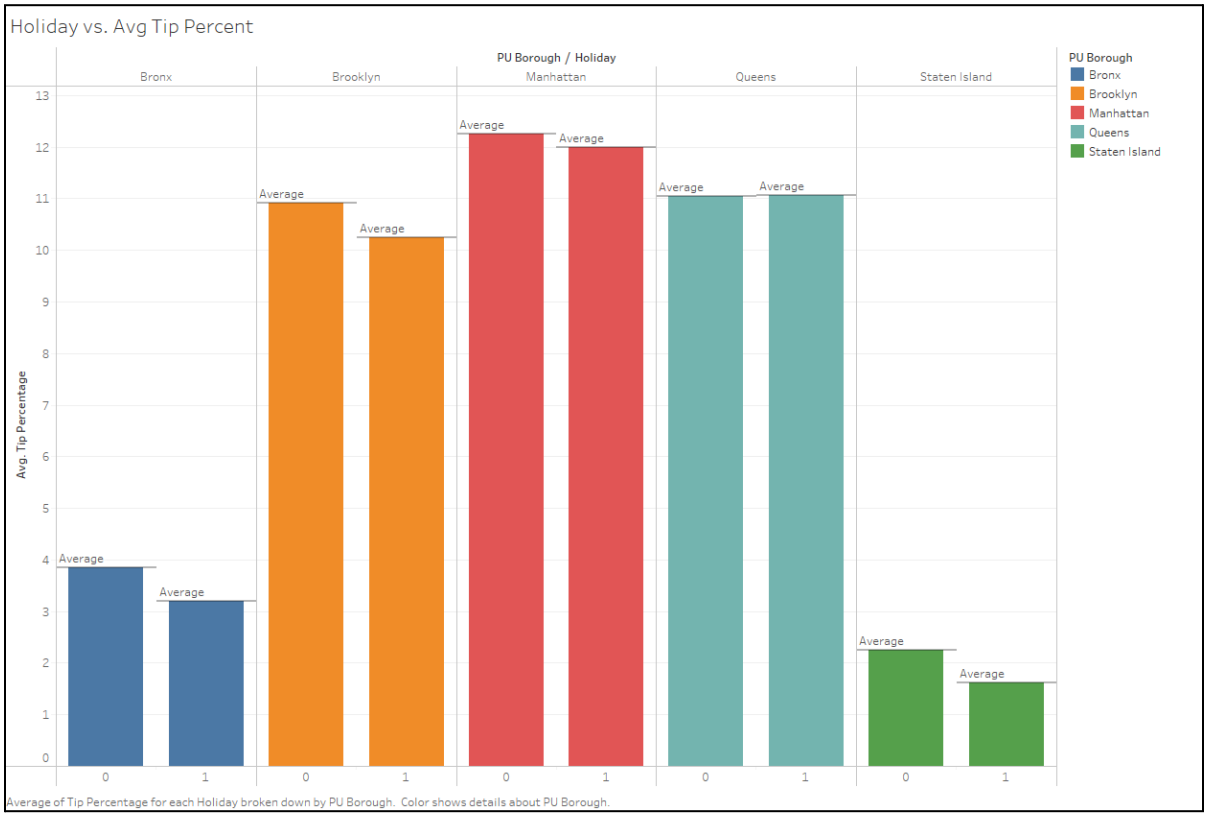


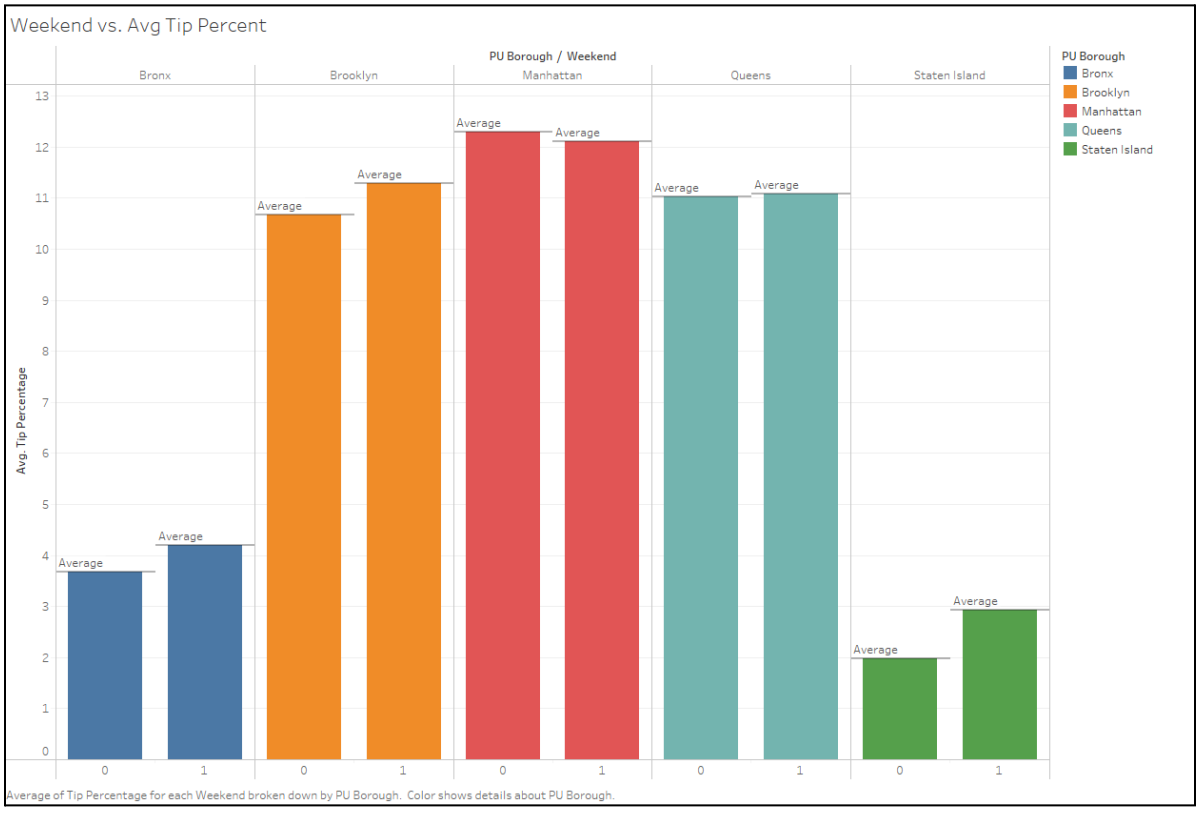
Month vs. Avg Tip Percent



The data shows that Saturday and Sunday have the highest average tip percentages, while Monday has the lowest. August has the lowest average tip percentage, and February has the highest.

Holiday vs. Avg Tip Percent

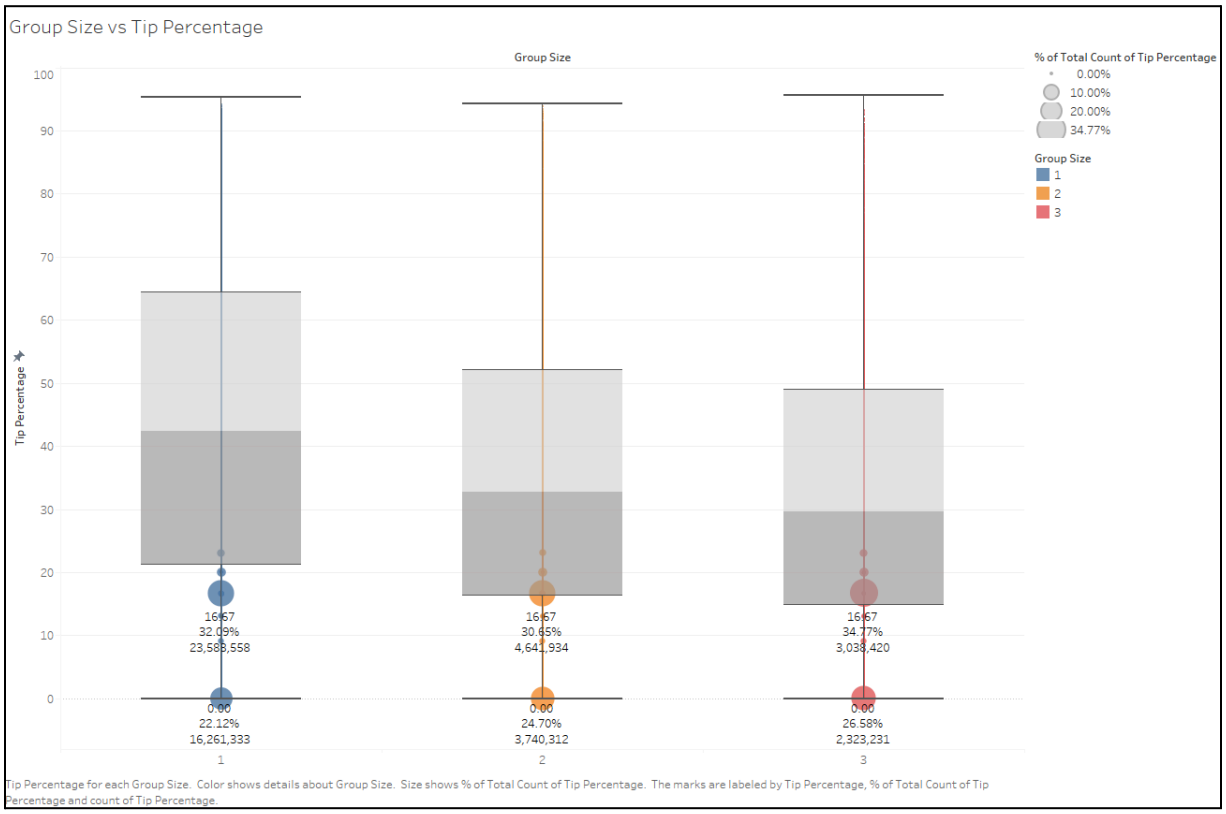




Group Size

Features	Coefficients
group_size	-0.07739401525

Group Size vs. Tip Percent

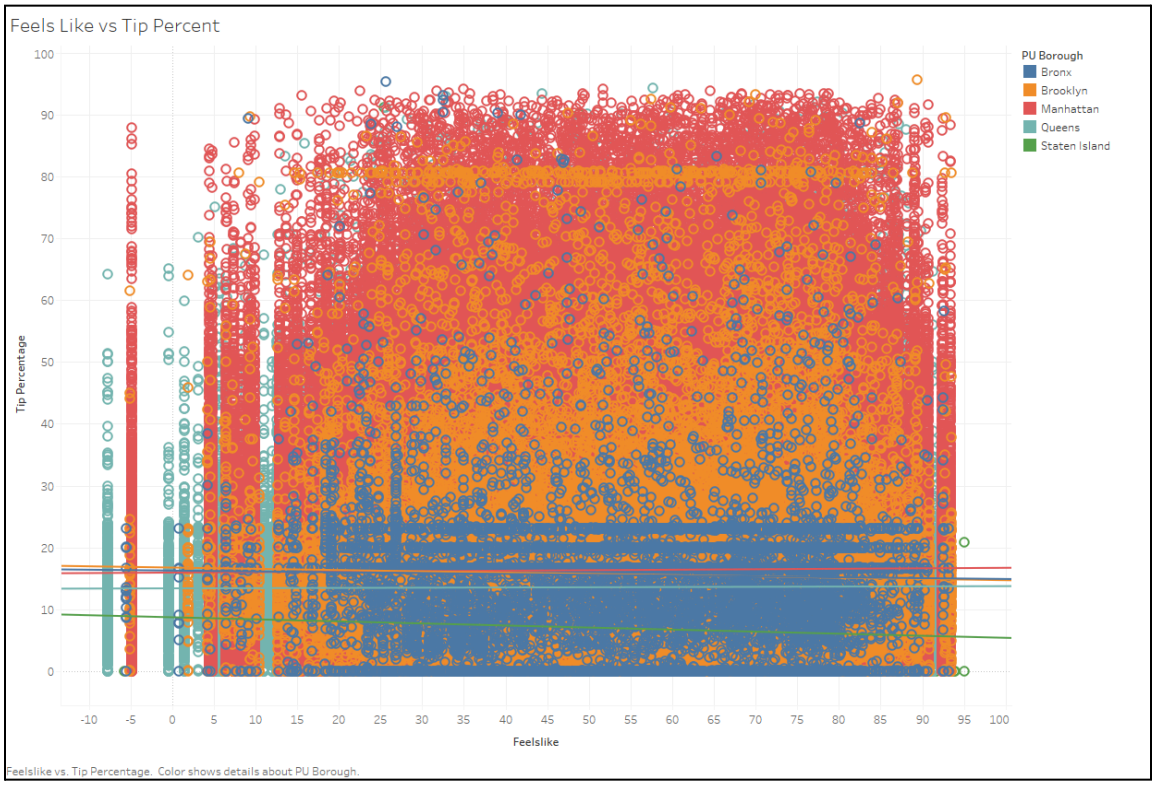


Weather Features

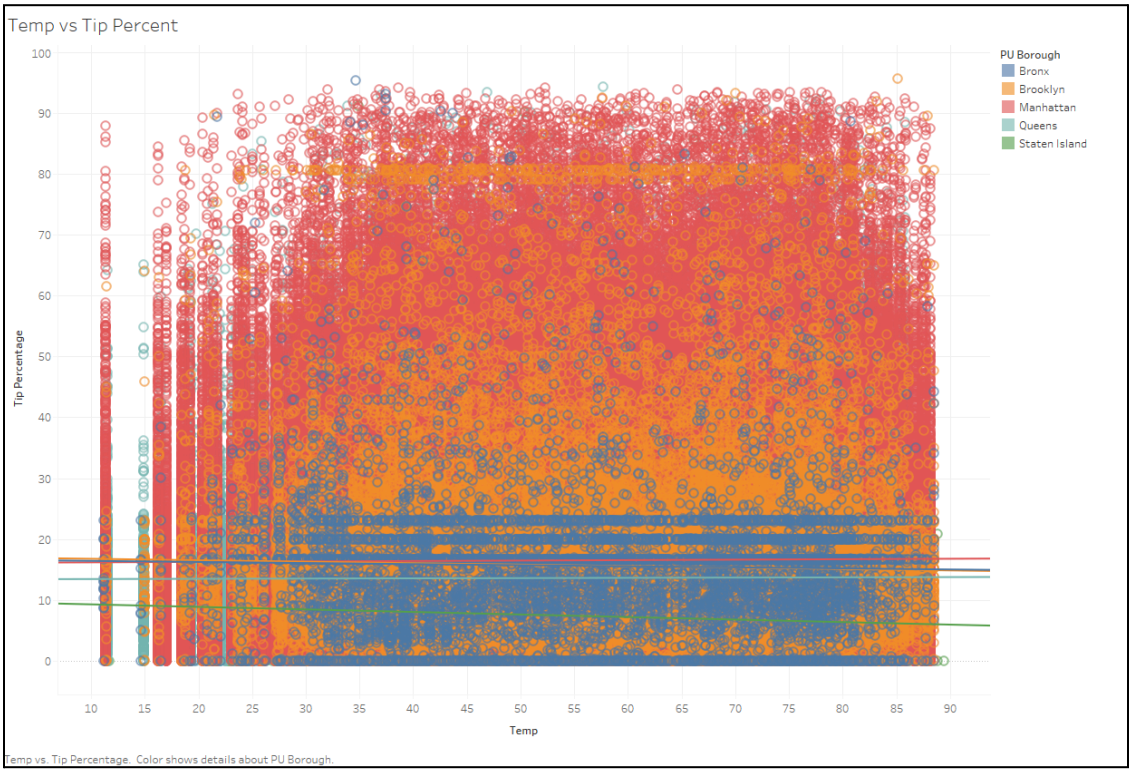
Features	Coefficients
scaled_features	0.01215823529
temp	
feelslike	
precip	

snow_precip	
snowdepth	
windspeed	
uvindex	

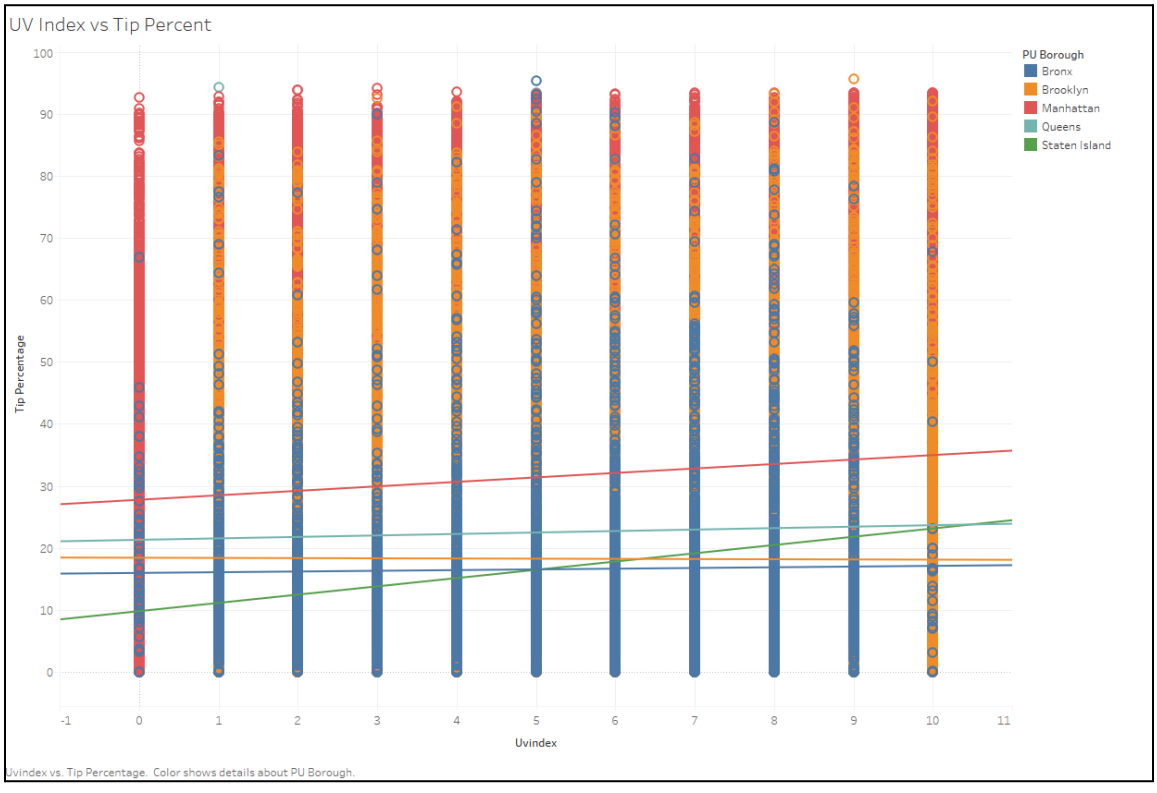
Feels Like



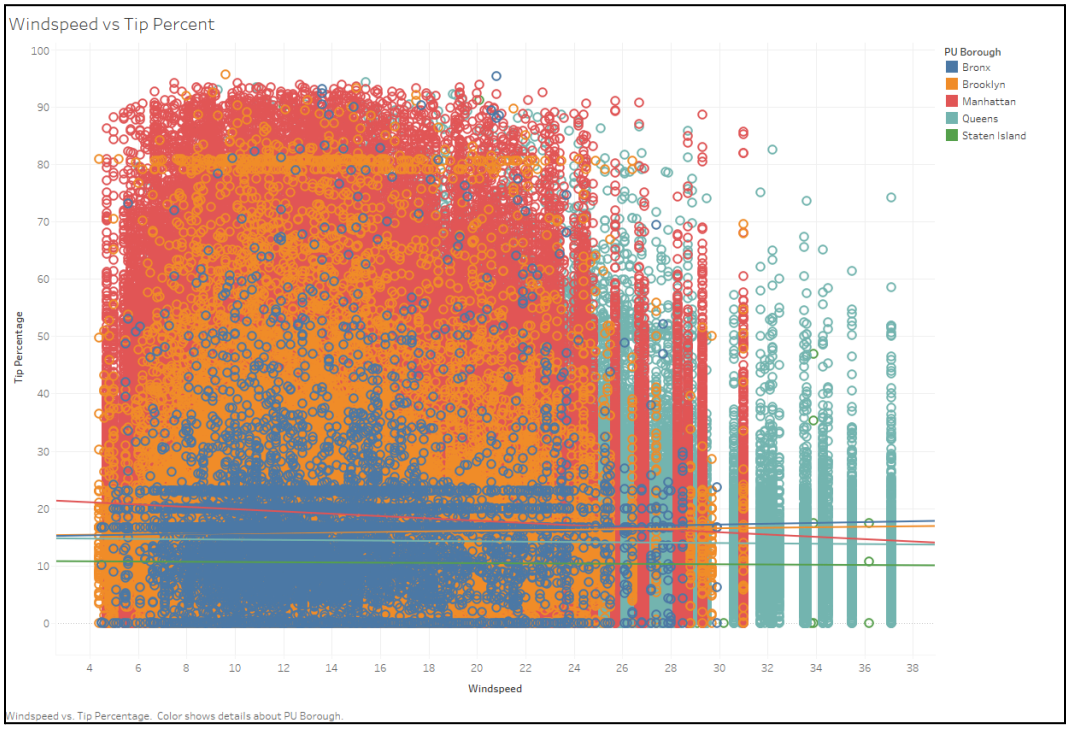
Temp



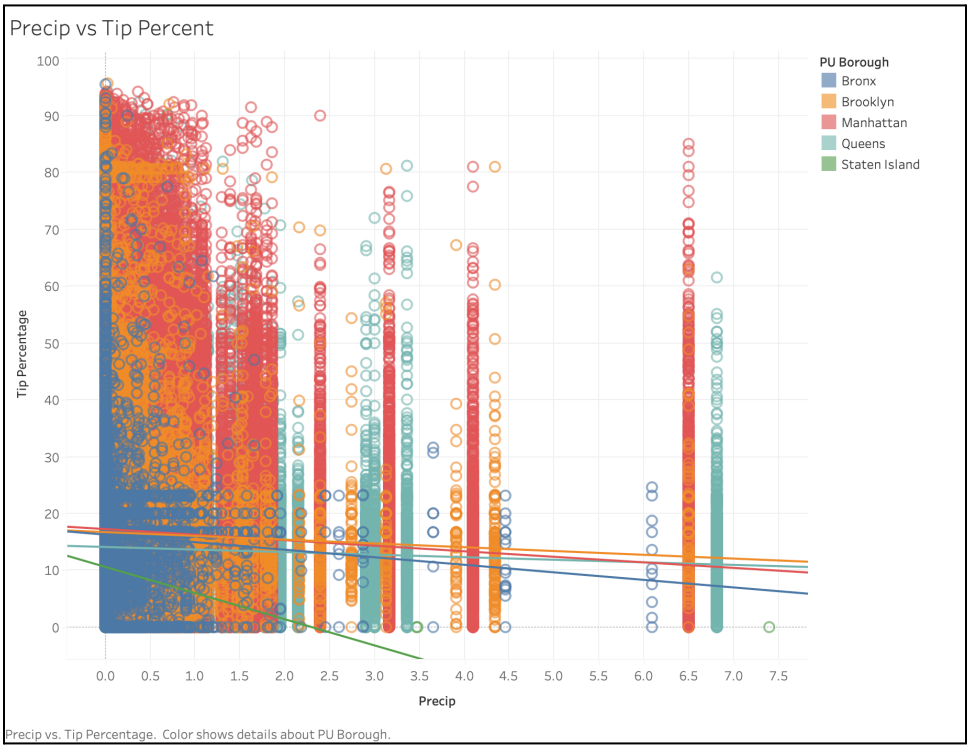
UV Index



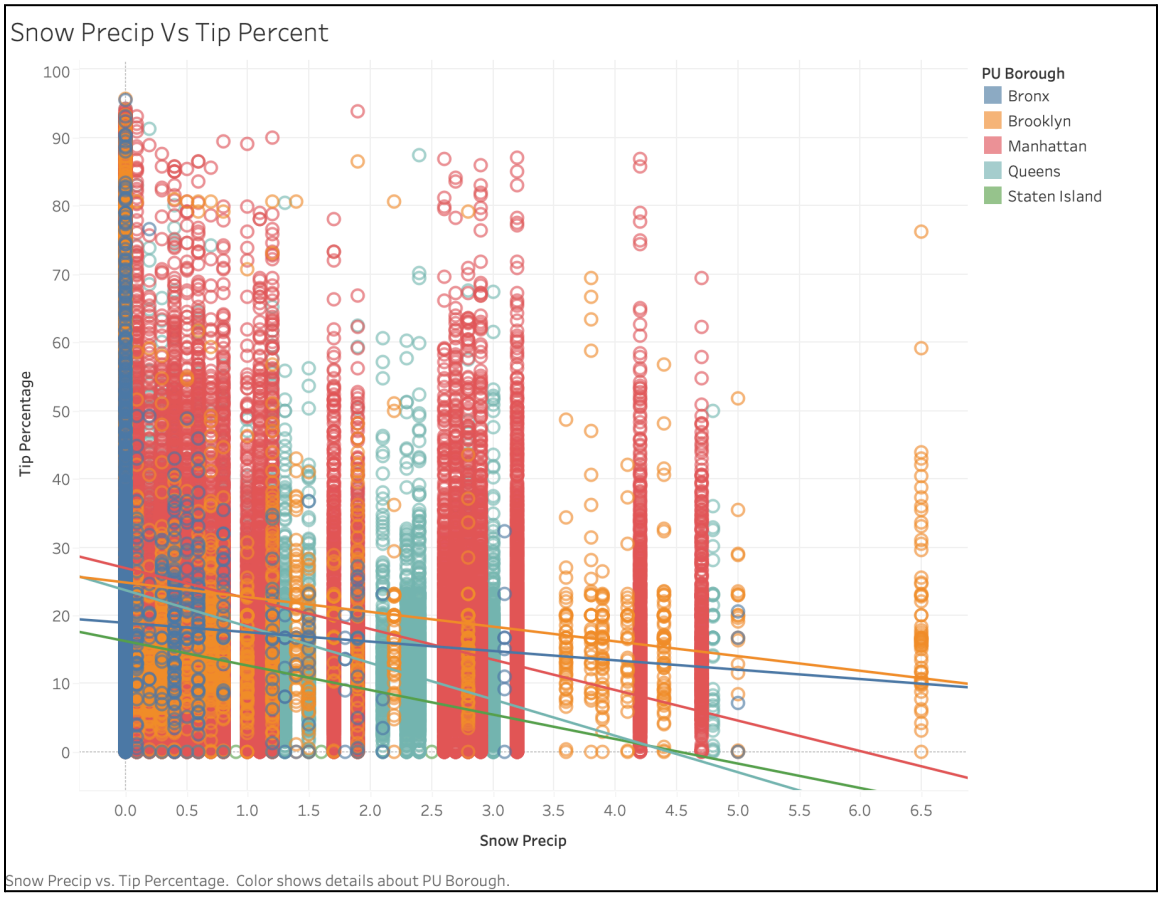
Windspeed



Precip vs. Tip Percent



Snow Precip vs. Tip Percent



Snow Precip vs. Tip Percentage. Color shows details about PU Borough.

Future Work

Overall, there are a lot of changes that I need to consider to improve this project further. For future tests, I want to continue to filter out taxi trips with unusual fare amounts relative to their trip distance. Then, I will remove either the fare amount or trip distance feature. Next, I would remove some features with significantly small coefficients. I would remove the binary condition features (Rain, Snow, Clear, Ice) and consider removing the temp and feelslike columns, as there is no correlation with tip percentage. The most significant features in this model should be the pick-up and drop-off borough, trip distance, month, day of week.

I will also consider using SMOTE to handle the imbalanced data set and adjusting class weights to improve precision and recall. I also want to learn more advanced machine learning techniques, such as Random Forest or Gradient Boosting, and apply them to this project.

Appendix

Appendix A

set_up_vm.sh

```
# Install pip3
sudo apt -y install python3-pip python3.11-venv

# Create a Python Virtual Environment
python3 -m venv pythondev

# Activate the Python Virtual Environment
cd pythondev; source bin/activate

# Install the required Python packages
pip3 install --upgrade pip
pip3 install jupyter pandas numpy requests sodapy
pip3 install google-cloud-storage

# Set up credentials for Google user account
gcloud auth application-default login

# Create extracting taxi data script
nano extracting_taxi_data.py

# Excute python script
python3 extracting_taxi_data.py

# Create extracting weather data script
nano extracting_weather_data.py

# Excute python script
python3 extracting_weather_data.py
```

extracting_taxi_data.py

```python
# Imports
from google.cloud import storage
import requests


# Create a client object that points to GCS
storage_client = storage.Client()

# Define the bucket
bucket_name = 'my-bigdataproject-jg'
bucket = storage_client.get_bucket(bucket_name)

# Destination folder
destination_folder = 'landing/'

# Taxi Data URL information
taxi_data_base_url = "https://d37ci6vzurychx.cloudfront.net/trip-data/"
data_years = [2021, 2022, 2023]
months = ['01', '02', '03', '04', '05', '06', '07', '08', '09', '10', '11', '12']

Headers = {'user-agent': 'Mozilla/4.0 (compatible; MSIE 5.5; Windows NT)'}

session = requests.Session()


def extract_taxi_data(file_name):
    print(f"Extracting file: \t{file_name}")
    full_url = taxi_data_base_url + file_name
    response = session.get(full_url, headers=Headers)
    print(f"Found file: \t{file_name}")
    data = response.content
    print(f"Uploading to GCS: \t{file_name}")
    blob = bucket.blob(destination_folder + file_name)
    blob.upload_from_string(data)
    print(f"Upload complete: \t{file_name}")

if __name__ == "__main__":
    # Loop through the taxi data ids keys
    print("Downloading taxi data...")
```

```
    for year in data_years:
        for month in months:
            file_name = f"yellow_tripdata_{year}-{month}.parquet"
            extract_taxi_data(file_name)
    print("Download complete.")
```

extracting_weather_data.py

```
from google.cloud import storage
import pandas as pd
from io import StringIO
from datetime import datetime
import requests

# Create a client object that points to GCS
storage_client = storage.Client()

# Define the bucket
bucket_name = 'my-bigdataproject-jg'
bucket = storage_client.get_bucket(bucket_name)

# Destination folder
destination_folder = 'landing/'

# Weather API Request URL example:
#
https://weather.visualcrossing.com/VisualCrossingWebServices/rest/services/timeline/[l
ocation]/[date1]/[date2]?key=YOUR_API_KEY

base_url =
"https://weather.visualcrossing.com/VisualCrossingWebServices/rest/services/timeline/"

boroughs = ['Manhattan', 'Brooklyn', 'Queens', 'Bronx', 'Staten Island']
years = [2021, 2022, 2023]
key = "" # Replace with your own API key
params = {
    "key" : key,
    "contentType" : "csv",
    "include" : "days"
```

```
}

def extract_weather_data(borough, year):
    start_date = f"{year}-01-01"
    end_date = f"{year}-12-31"
    location = f"{borough}, NY"

    url = f"{base_url}{location}/{start_date}/{end_date}"
    response = requests.get(url, params=params)

    if response.status_code == 200:
        df = pd.read_csv(StringIO(response.text))
        file_name = f"weather_data_{borough}_{year}.csv"
        print(f"Saving to CSV: \t{file_name}")
        csv_buffer = df.to_csv(index=False)
        blob = bucket.blob(destination_folder + file_name)
        blob.upload_from_string(csv_buffer, content_type='text/csv')
        print(f"Upload complete: \t{file_name}")
    else:
        print(f"Error: {response.status_code}")


if __name__ == "__main__":
    print("Choose a borough:")
    for i, borough in enumerate(boroughs):
        print(f"{i+1}. {borough}")
    borough = int(input("Select a borough:\t")) - 1

    print("Choose a year:")
    for i, year in enumerate(years):
        print(f"{i+1}. {year}")
    year = int(input("Select the year:\t")) - 1

    borough = boroughs[borough]
    year = years[year]

    print(f"Extracting weather data for {borough} in {year}")

    extract_weather_data(borough, year)
```

Appendix B

taxi_data_analysis.ipynb

```
# %%
spark

# %% [markdown]
# # Imports

# %%
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from google.cloud import storage
from io import BytesIO
from datetime import datetime, date

import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import StringType, BooleanType, IntegerType , DateType, FloatType,
StructType, StructField
from pyspark.sql.functions import col, isnan, when, count, udf, to_date, year, month,
date_format, size, split, dayofweek
from pyspark.sql import functions as F

# %% [markdown]
# # Taxi Data Frame

# %%
bucket_name = 'my-bigdataproject-jg'
gs_path  = f'gs://{bucket_name}/'
landing_folder = 'landing/'
cleaned_folder = 'cleaned/'
destination_folder = 'code_and_models/'

storage_client = storage.Client()
bucket = storage_client.get_bucket(bucket_name)

# %%
data_years = [2021, 2022, 2023]
months = ['01', '02', '03', '04', '05', '06', '07', '08', '09', '10', '11', '12']

taxi_file_path = gs_path + landing_folder
```

```
taxi_file_list = [taxi_file_path + f'yellow_tripdata_{year}-{month}.parquet' for year in
data_years for month in months]

taxi_df = None
for file in taxi_file_list:
    df = spark.read.parquet(file)
    df = df.withColumn('VendorID', df['VendorID'].cast(IntegerType()))
    df = df.withColumn('passenger_count', df['passenger_count'].cast(IntegerType()))
    if taxi_df is None:
        taxi_df = df
    else:
        taxi_df = taxi_df.union(df)

taxi_df.show(5)

# %% [markdown]
# # Shape

# %%
print(f'Number of records in the dataframe: {taxi_df.count():,}')
print(f'Number of columns in the dataframe: {len(taxi_df.columns)}')

taxi_df.printSchema()

# %% [markdown]
# # Empty Records

# %%
taxi_df.select([count(when(col(c).isNull(), c)).alias(c) for c in taxi_df.columns]).show()

# %% [markdown]
# # RatecodeID

# %%
taxi_df.groupBy('RatecodeID').count().show()

# %% [markdown]
# # passenger_count

# %%
taxi_df.select('passenger_count').summary().show()

# %%
taxi_df.groupBy('passenger_count').count().show()
```

```
# %%
# Create a histogram of the passenger_count column to show the distribution of the number of passengers
passenger_count_df = taxi_df.groupBy('passenger_count').count().toPandas()
passenger_count_df = passenger_count_df.sort_values(by='passenger_count')
passenger_count_df.plot(kind='bar', x='passenger_count', y='count', color='blue', figsize=(10, 6))
plt.title('Distribution of the number of passengers')
plt.xlabel('Number of passengers')
plt.ylabel('Count')
plt.xticks(rotation=0)
for i, count in enumerate(passenger_count_df['count']):
    plt.text(i, count, f'{count:,}', ha='center', va='bottom')

plt.show()


# %% [markdown]
# # trip_distance

# %%
taxi_df.select('trip_distance').summary().show()

# %%
long_trips_df = taxi_df.filter(taxi_df['trip_distance'] >= 50)
long_trips_df.select('trip_distance').summary().show()

# %%
short_trips_df = taxi_df.filter(taxi_df['trip_distance'] <= 3)
short_trips_df.select('trip_distance').summary().show()

# %%
print(f'Number of trips with a distance of 0 miles: {taxi_df.filter(taxi_df["trip_distance"] == 0).count()}')
print(f'Number of trips with a distance greater than 45 miles: {taxi_df.filter(taxi_df["trip_distance"] >= 45).count()}')
print(f'Number of trips with a distance greater than 60 miles: {taxi_df.filter(taxi_df["trip_distance"] >= 60).count()}')
print(f'Number of trips with a distance greater than 75 miles: {taxi_df.filter(taxi_df["trip_distance"] >= 75).count()}')
print(f'Number of trips with a distance greater than 100 miles: {taxi_df.filter(taxi_df["trip_distance"] >= 100).count()}')

# %% [markdown]
# # fare_amount
```

```
# %%
taxi_df.select('fare_amount').summary().show()

# %%
# Create a fare amount data frame with only positive fare amounts
positive_fare_amount_df = taxi_df.filter(taxi_df['fare_amount'] >= 3)
positive_fare_amount_df.select('fare_amount', 'tip_amount').summary().show()

# %%
# Large fare amount data frame
large_fare_amount_df = taxi_df.filter(taxi_df['fare_amount'] >= 250)
large_fare_amount_df.select('fare_amount', 'tip_amount').summary().show()

# %% [markdown]
# # tip_amount

# %%
taxi_df.select('tip_amount').summary().show()

# %%
positive_tip_amount_df = taxi_df.filter(taxi_df['tip_amount'] >= 0)
positive_tip_amount_df.select('tip_amount').summary().show()

# %%
# large tip amount data frame
large_tip_amount_df = taxi_df.filter(taxi_df['tip_amount'] >= 150)
large_tip_amount_df.select('tip_amount').summary().show()


# %% [markdown]
# # tip_percent

# %%
taxi_df = taxi_df.withColumn('tip_percentage', taxi_df['tip_amount'] / taxi_df['fare_amount'])
taxi_df.select('tip_percentage').summary().show()

# %%
# get the positive tip percentage data frame
positive_tip_percentage_df = taxi_df.filter(taxi_df['tip_percentage'] > 0)
positive_tip_percentage_df.select('tip_percentage').summary().show()

# %% [markdown]
# # total_amount
```

```
# %%
taxi_df.select('total_amount').summary().show()

# %%
positive_total_amount_df = taxi_df.filter(taxi_df['total_amount'] > 3)
positive_total_amount_df.select('total_amount').summary().show()


# %%
# get the large total amount data frame
large_total_amount_df = taxi_df.filter(taxi_df['total_amount'] >= 400)
large_total_amount_df.select('total_amount', 'fare_amount', 'tip_amount').summary().show()

# %% [markdown]
# # Date Time

# %%
# Check the min and max dates for the two columns: tpep_pickup_datetime and
tpep_dropoff_datetime
taxi_df.select(F.min('tpep_pickup_datetime'), F.max('tpep_pickup_datetime')).show()
taxi_df.select(F.min('tpep_dropoff_datetime'), F.max('tpep_dropoff_datetime')).show()

# Only keep the records within the date range: 2021-01-01 to 2024-01-01

taxi_df = taxi_df.filter((col('tpep_pickup_datetime') >= '2021-01-01') &
(col('tpep_pickup_datetime') < '2024-01-01'))
taxi_df = taxi_df.filter((col('tpep_dropoff_datetime') >= '2021-01-01') &
(col('tpep_dropoff_datetime') < '2024-01-01'))

# Check the min and max dates for the two columns: tpep_pickup_datetime and
tpep_dropoff_datetime
taxi_df.select(F.min('tpep_pickup_datetime'), F.max('tpep_pickup_datetime')).show()
taxi_df.select(F.min('tpep_dropoff_datetime'), F.max('tpep_dropoff_datetime')).show()


# %% [markdown]
# # Number of trips

# %%
# Create a new date column
# Using the tpep_pickup_datetime column, create a new column week_of_year which is the week of
the year
# Then create a new column year_week which is the year and week combined
# It should be the week number and year combined, e.g. 2023-01
taxi_df = taxi_df.withColumn('week_of_year', F.weekofyear('tpep_pickup_datetime'))
```

```
taxi_df = taxi_df.withColumn('year_week', F.concat(F.year('tpep_pickup_datetime'), F.lit('-'),
F.lpad('week_of_year', 2, '0')))

# Create a graph showing the number of trips per week
trips_per_week_df = taxi_df.groupBy('year_week').count().toPandas()
trips_per_week_df = trips_per_week_df.sort_values(by='year_week')
trips_per_week_df.plot(kind='bar', x='year_week', y='count', color='blue', figsize=(20, 6))
plt.title('Number of trips per week')
plt.xlabel('Year and week')
plt.ylabel('Number of trips')
plt.xticks(rotation=90)

plt.show()



# %%
```

weather_data_analysis.ipynb

```
# %%
import pandas as pd
import matplotlib.pyplot as plt

# %%
# ex url: gs://my-bigdataproject-jg/landing/weather_data_Manhattan_2021.csv
bucket_name = 'my-bigdataproject-jg'
gs_path  = f'gs://{bucket_name}/landing/'

boroughs = ['Manhattan', 'Brooklyn']
years = [2021, 2022, 2023]

# %%
def get_weather_data(borough, year):
    file_name = f'weather_data_{borough}_{year}.csv'
    full_path = gs_path + file_name
    return pd.read_csv(full_path)
```

```
# %%
def analyze_data(df):
   print(f"Number of records: {len(df)}")
   print(f"Number of columns: {len(df.columns)}\n")
   print(df.info())

   print("\nMissing values:")
   print(df.isnull().sum())

   print(df.describe())

   # Create a histogram of tempreatures throughout the year
   df['datedatetime'] = pd.to_datetime(df['datetime'])
   df.set_index('datetime', inplace=True)
   df['temp'].plot()
   plt.show()

   # we want to check for outliers in temp, tempmax, tempmin, feelslike, feelslikemax,
feelslikemin
   columns = ['temp', 'tempmax', 'tempmin', 'feelslike', 'feelslikemax',
'feelslikemin']
   # all in one chart
   df[columns].plot(kind='box')
   plt.show()

# %%
if __name__ == "__main__":
   print("Starting data processing")
   for borough in boroughs:
       for year in years:
           print(f'Processing data for {borough}, {year}')
           df = get_weather_data(borough, year)
           analyze_data(df)
   print("Data processing finished")
```

Appendix C

cleaning_taxi_data.ipynb

```
# %%
spark

# %%
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from google.cloud import storage
from io import BytesIO
from datetime import datetime, date

import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import StringType, BooleanType, IntegerType , DateType,
FloatType, StructType, StructField
from pyspark.sql.functions import col, isnan, when, count, udf, to_date, year, month,
date_format, size, split, dayofweek
from pyspark.sql import functions as F

# %% [markdown]
# # Taxi Data Frame

# %%
bucket_name = 'my-bigdataproject-jg'
gs_path  = f'gs://{bucket_name}/'
landing_folder = 'landing/'
cleaned_folder = 'cleaned/'
destination_folder = 'code_and_models/'

storage_client = storage.Client()
bucket = storage_client.get_bucket(bucket_name)

# %%
data_years = [2021, 2022, 2023]
months = ['01', '02', '03', '04', '05', '06', '07', '08', '09', '10', '11', '12']

taxi_file_path = gs_path + landing_folder
taxi_file_list = [taxi_file_path + f'yellow_tripdata_{year}-{month}.parquet' for year
in data_years for month in months]
```

```
taxi_df = None
for file in taxi_file_list:
   df = spark.read.parquet(file)
   df = df.withColumn('VendorID', df['VendorID'].cast(IntegerType()))
   df = df.withColumn('passenger_count', df['passenger_count'].cast(IntegerType()))
   if taxi_df is None:
       taxi_df = df
   else:
       taxi_df = taxi_df.union(df)

taxi_df.show(5)

# %%
print(f'Number of records in the dataframe: {taxi_df.count():,}')

# %% [markdown]
# # Date Range

# %%
taxi_df = taxi_df.filter((col('tpep_pickup_datetime') >= '2021-01-01') &
(col('tpep_pickup_datetime') < '2024-01-01'))
taxi_df = taxi_df.filter((col('tpep_dropoff_datetime') >= '2021-01-01') &
(col('tpep_dropoff_datetime') < '2024-01-02'))

taxi_df.select(F.min('tpep_pickup_datetime'), F.max('tpep_pickup_datetime')).show()
taxi_df.select(F.min('tpep_dropoff_datetime'), F.max('tpep_dropoff_datetime')).show()

# rename tpep_pickup_datetime and tpep_dropoff_datetime columns to pickup_datetime and
dropoff_datetime
taxi_df = taxi_df.withColumnRenamed('tpep_pickup_datetime', 'pickup_datetime')
taxi_df = taxi_df.withColumnRenamed('tpep_dropoff_datetime', 'dropoff_datetime')


# %% [markdown]
# # payment_type

# %%
# payment_type = 6 are voided trips
taxi_df = taxi_df.filter(col('payment_type') != 6)

# %% [markdown]
```

```
# # RatecodeID

# %%
# 1 is standard rate, 2 is JFK, 6 is group ride, 99 is unknown
allowed_rate_codes = [1, 2, 6, 99]
# fill the empty RatecodeID with 99
taxi_df = taxi_df.withColumn('RatecodeID', when(col('RatecodeID').isNull(),
99).otherwise(col('RatecodeID')))
# Change RatecodeID to an integer
taxi_df = taxi_df.withColumn('RatecodeID', taxi_df['RatecodeID'].cast(IntegerType()))
# filter the RatecodeID to only include the allowed values
taxi_df = taxi_df.filter(col('RatecodeID').isin(allowed_rate_codes))

# %% [markdown]
# # passenger_count

# %%
taxi_df = taxi_df.fillna({'passenger_count': 0})
taxi_df = taxi_df.filter(col('passenger_count') <= 7)



# %%
print(f'Number of records in the dataframe: {taxi_df.count():,}')

# %% [markdown]
# # fare_amount

# %%

taxi_df = taxi_df.filter((col('fare_amount') >= 3) & (col('fare_amount') <= 250))



# %% [markdown]
# # total_amount

# %%

taxi_df = taxi_df.filter((col('total_amount') > 3) & (col('total_amount') < 400))

# %% [markdown]
# # tip_amount
```

```
# %%
# tip amount should be less than 150 and greater than 0
taxi_df = taxi_df.filter((col('tip_amount') >= 0) & (col('tip_amount') < 150))

# %% [markdown]
# # tip_percentage

# %%
# Create a tip percentage column
taxi_df = taxi_df.withColumn('tip_percentage', col('tip_amount') / col('total_amount')
* 100)

# %% [markdown]
# # trip_distance

# %%
# trip_distance should be greater than 0 and less than 60
taxi_df = taxi_df.filter((col('trip_distance') > 0) & (col('trip_distance') < 60))

# %% [markdown]
# # Saving the cleaned taxi data

# %%
# Summarize the fare_amount, tip_amount, total_amount, tip_percentage, trip_distance
taxi_df.select('fare_amount', 'tip_amount', 'total_amount', 'tip_percentage',
'trip_distance').summary().show()

# %%
# Schema for the cleaned data
taxi_df = taxi_df.select('pickup_datetime', 'dropoff_datetime', 'passenger_count',
'trip_distance', 'RatecodeID', 'PULocationID', 'DOLocationID', 'fare_amount',
'tip_amount', 'total_amount', 'tip_percentage')
taxi_df.printSchema()

# %%
# save the cleaned data
print('Saving the cleaned data')
taxi_df.write.mode('overwrite').parquet(gs_path + cleaned_folder + 'taxi_data/')
print('Data saved successfully')
```

cleaning_weather_data.ipynb

```
# %%
import pandas as pd
from google.cloud import storage

# %%
boroughs = ['Manhattan', 'Brooklyn']
years = [2021, 2022, 2023]

# %%
bucket_name = 'my-bigdataproject-jg'
gs_path  = f'gs://{bucket_name}/'
storage_client = storage.Client()
bucket = storage_client.get_bucket(bucket_name)
cleaned_folder = 'cleaned/'
landing_folder = 'landing/'

# %%
def get_data(borough, year):
   print(f'Getting data for {borough} in {year}')
   file_name = f'weather_data_{borough}_{year}.csv'
   file_path = f'{landing_folder}{file_name}'
   blob = bucket.blob(file_path)
   blob.download_to_filename(file_name)
   df = pd.read_csv(file_name)
   return df


# %%
def clean_data(df, borough, year):
   print('Cleaning data')
   # Keep only the columns we need
   df = df[['name', 'datetime', 'tempmax', 'tempmin', 'temp', 'feelslikemax',
'feelslikemin', 'feelslike', 'humidity', 'precip', 'preciptype', 'snow', 'snowdepth',
'windspeed', 'cloudcover', 'visibility', 'uvindex', 'sunrise', 'sunset',
'conditions']].copy()
```

```
# Fill in the rows with missing precip type to 'None'
df['preciptype'] = df['preciptype'].fillna('None')

# Set the data types
# Set to datetime
df['datetime'] = pd.to_datetime(df['datetime'])
df['sunrise'] = pd.to_datetime(df['sunrise'])
df['sunset'] = pd.to_datetime(df['sunset'])

# Set to string
df['name'] = df['name'].astype('string')
df['preciptype'] = df['preciptype'].astype('string')
df['conditions'] = df['conditions'].astype('string')

# Set to double
df['tempmax'] = df['tempmax'].astype('float64')
df['tempmin'] = df['tempmin'].astype('float64')
df['temp'] = df['temp'].astype('float64')
df['feelslikemax'] = df['feelslikemax'].astype('float64')
df['feelslikemin'] = df['feelslikemin'].astype('float64')
df['feelslike'] = df['feelslike'].astype('float64')
df['humidity'] = df['humidity'].astype('float64')
df['precip'] = df['precip'].astype('float64')
df['snow'] = df['snow'].astype('float64')
df['snowdepth'] = df['snowdepth'].astype('float64')
df['windspeed'] = df['windspeed'].astype('float64')
df['cloudcover'] = df['cloudcover'].astype('float64')
df['visibility'] = df['visibility'].astype('float64')

# Set to int
df['uvindex'] = df['uvindex'].astype('Int64')


# Drop rows with missing values
df = df.dropna()
print('Data cleaned')

# Upload cleaned data to GCS as a Parquet file
file_name = f'weather_data_{borough}_{year}.parquet'
file_path = f'{cleaned_folder}{file_name}'
print(f'Uploading:\t {file_path}')
```

```
    df.to_parquet(file_name)
    blob = bucket.blob(file_path)
    blob.upload_from_filename(file_name)
    print('Data uploaded')


# %%
if __name__ == "__main__":
    print('Starting data cleaning')
    for borough in boroughs:
        for year in years:
            df = get_data(borough, year)
            clean_data(df, borough, year)
    print('Data cleaning complete')
```

cleaning_taxi_zone_data.ipynb

```
# %%
import pandas as pd
from google.cloud import storage

# %%
bucket_name = 'my-bigdataproject-jg'
gs_path  = f'gs://{bucket_name}/'
storage_client = storage.Client()
bucket = storage_client.get_bucket(bucket_name)
cleaned_folder = 'cleaned/'
landing_folder = 'landing/'

# %%
def get_data():
    print('Getting data')
    file_name = 'taxi_zones_data.csv'
    file_path = f'{landing_folder}{file_name}'
    blob = bucket.blob(file_path)
    blob.download_to_filename(file_name)
    df = pd.read_csv(file_name)
    return df

# %%
```

```
def clean_data(df):
    print('Cleaning data')
    # Keep zone, location_id, borough
    df = df[['zone', 'LocationID', 'borough']].copy()
    # Set column types
    df['zone'] = df['zone'].astype(str)
    df['LocationID'] = df['LocationID'].astype(int)
    df['borough'] = df['borough'].astype(str)

    # Drop empty rows
    df = df.dropna()
    print('Data Cleaned')
    print(df.head())

    # Upload cleaned data to GCS as a Parquet file
    file_name = 'taxi_zones_data.parquet'
    file_path = f'{cleaned_folder}{file_name}'
    print(f'Uploading to: \t{file_path}')
    df.to_parquet(file_name)
    blob = bucket.blob(file_path)
    blob.upload_from_filename(file_name)
    print('Data Uploaded')




# %%
if __name__ == "__main__":
    df = get_data()
    clean_data(df)
```

Appendix D

features_and_models.ipynb

```
# %%
!pip install holidays

# %%
```

```
spark

# %% [markdown]
# # Set Up

# %%
import pandas as pd
from google.cloud import storage
from io import BytesIO
from datetime import datetime, date
import matplotlib.pyplot as plt
import holidays

import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, isnan, when, count, udf, to_date, year, month,
date_format, size, split, dayofweek
from pyspark.sql import functions as F
from pyspark.ml.feature import StringIndexer, OneHotEncoder, VectorAssembler,
StandardScaler, MinMaxScaler
from pyspark.ml.regression import LinearRegression
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.mllib.evaluation import MulticlassMetrics
from pyspark.ml import Pipeline




# %%
bucket_name = 'my-bigdataproject-jg'
gs_path  = f'gs://{bucket_name}/'
cleaned_folder = 'cleaned/'
destination_folder = 'code_and_models/'

storage_client = storage.Client()
bucket = storage_client.get_bucket(bucket_name)

# %% [markdown]
# # Weather DF
```

```
# %%
weather_files = gs_path + cleaned_folder + 'weather_data_*.parquet'
weather_df = spark.read.parquet(weather_files)


weather_df = weather_df.drop('tempmax', 'tempmin', 'feelslikemax', 'feelslikemin',
'preciptype', 'cloudcover', 'visibility', 'humidity')

weather_df = weather_df.withColumnRenamed('borough', 'weather_borough')
weather_df = weather_df.withColumnRenamed('snow', 'snow_precip')

weather_df.printSchema()

# %%
# Show me summary statistics for temp, feelslike, precip, snow_precip, snowdepth,
windspeed, uvindex
weather_df.select('temp', 'feelslike', 'precip', 'snow_precip', 'snowdepth',
'windspeed', 'uvindex').summary().show()

# %% [markdown]
# # Taxi Trips DF

# %%
taxi_files = gs_path + cleaned_folder + "taxi_data/*.parquet"
taxi_df = spark.read.parquet(taxi_files)
taxi_df.printSchema()

# %%
# create a new column pickup_date, which the date using the pickup_datetime column
taxi_df = taxi_df.withColumn('pickup_date', to_date(col('pickup_datetime')))


# Create hour_of_day column from pickup_datetime
taxi_df = taxi_df.withColumn('hour_of_day', F.hour(col('pickup_datetime')))
# Create a time_of_day column based on hour_of_day
taxi_df = taxi_df.withColumn('time_of_day', when((col('hour_of_day') >= 5) &
(col('hour_of_day') < 12), 'Morning').when((col('hour_of_day') >= 12) &
(col('hour_of_day') < 21), 'Afternoon').otherwise('Night'))
# Show the number of records for each unique time_of_day
taxi_df.groupBy('time_of_day').count().show()
```

```
# %%
# Create a new column, group_size, based on the number of passengers
# group_size is 1 if num_passengers is 1
# group_size is 2 if num_passengers is 2
# group_size is 3 if num_passengers is greater than 3
taxi_df = taxi_df.withColumn('group_size', when(col('passenger_count') == 1,
1).when(col('passenger_count') == 2, 2).otherwise(3))

# Show the number of records for each unique group_size
taxi_df.groupBy('group_size').count().show()



# %%
# Remove the records with RatecodeID = 99
taxi_df = taxi_df.filter(col('RatecodeID') != 99)

# Remove the records passenger_count = 0
taxi_df = taxi_df.filter(col('passenger_count') != 0)

# Remove the records with fare_amount < 3.70 (minimum fare amount)
taxi_df = taxi_df.filter(col('fare_amount') >= 3.70)

# Remove the records with total_amount <= 4.20 (minimum total amount)
taxi_df = taxi_df.filter(col('total_amount') > 4.20)

# Remove the records with trip_distance < 1/5 mile
taxi_df = taxi_df.filter((col('trip_distance') >= 0.2) & (col('trip_distance') <= 45))

# Round the tip_percentage to 2 decimal places
taxi_df = taxi_df.withColumn('tip_percentage', F.round(col('tip_percentage'), 2))



taxi_df = taxi_df.drop('dropoff_datetime', 'RatecodeID', 'payment_type',
'total_amount', 'pickup_datetime', 'tip_amount', 'hour_of_day')



# %% [markdown]
# # Taxi Zone DF

# %%
```

```
taxi_zone_file = gs_path + cleaned_folder + 'taxi_zones_data.parquet'
taxi_zone_df = spark.read.parquet(taxi_zone_file)
taxi_zone_df = taxi_zone_df.drop('zone')
taxi_zone_df.printSchema()


# %% [markdown]
# ## Taxi data frames combined

# %%
# PU Location join
taxi_df = taxi_df.join(taxi_zone_df, taxi_df.PULocationID == taxi_zone_df.LocationID,
how='left')


taxi_df = taxi_df.withColumnRenamed('Borough', 'PUBorough')
taxi_df = taxi_df.drop('LocationID')

# DO Location join
taxi_df = taxi_df.join(taxi_zone_df, taxi_df.DOLocationID == taxi_zone_df.LocationID,
how='left')

taxi_df = taxi_df.withColumnRenamed('Borough', 'DOBorough')
taxi_df = taxi_df.drop('LocationID')

# Drop the PULocationID and DOLocationID columns
taxi_df = taxi_df.drop('PULocationID', 'DOLocationID')

# Drop the records where the PUBorough or DOBorough is 'EWR'
taxi_df = taxi_df.filter((taxi_df.PUBorough != 'EWR'))
taxi_df = taxi_df.filter((taxi_df.DOBorough != 'EWR'))

taxi_df.printSchema()

# %% [markdown]
# # Combined Data Frame

# %%
combined_df = taxi_df.join(weather_df, [taxi_df.pickup_date == weather_df.datetime,
taxi_df.PUBorough == weather_df.weather_borough])

combined_df = combined_df.drop('datetime')
```

```
combined_df = combined_df.drop('weather_borough')

combined_df.printSchema()



# %%
combined_df.select('tip_percentage', 'trip_distance', 'fare_amount',
'passenger_count').summary().show()

# %% [markdown]
# # Tip Label

# %%
# Tip Class column
# When tip_percentage >= 15, tip_class is 1, otherwise 0
combined_df = combined_df.withColumn('tip_class', when(col('tip_percentage') >= 15,
1).otherwise(0))

# Show the number of records for each unqiue tip_class
combined_df.groupBy('tip_class').count().show()



# %% [markdown]
# # Datetime features

# %%
# datetime

# month
combined_df = combined_df.withColumn('month', month(col('pickup_date')))
# dayofweek
combined_df = combined_df.withColumn('dayofweek', dayofweek(col('pickup_date')))

# weekend
combined_df = combined_df.withColumn('weekend', when(col('dayofweek') == 1,
1.0).when(col('dayofweek') == 7, 1.0).otherwise(0))

# holiday
combined_df = combined_df.withColumn('pickup_date', to_date(col('pickup_date')))

# Get the min and max date in the datetime column
min_date = combined_df.agg({"pickup_date": "min"}).collect()[0][0]
```

```
max_date = combined_df.agg({"pickup_date": "max"}).collect()[0][0]


# Get the holidays observed in New York
us_holidays = holidays.UnitedStates(years=[min_date.year, max_date.year],
observed=True, subdiv='NY')

#print(us_holidays)

# Keep only the dates of the holidays
us_holidays = list(us_holidays.keys())

# Create a new column holiday and set it to 1 if the date is a holiday, 0 otherwise
combined_df = combined_df.withColumn('holiday',
when(col('pickup_date').isin(us_holidays), 1).otherwise(0))

# %% [markdown]
# # UDF Condition Features

# %%

# Or we can use the list ['Rain', 'Snow', 'Clear', 'Ice']
conditions = ['Rain', 'Snow', 'Clear', 'Ice']

# Create a new column for each condition
for condition in conditions:
   combined_df = combined_df.withColumn(condition,
when(col('conditions').contains(condition), 1).otherwise(0))


# %% [markdown]
# # Pipeline

# %% [markdown]
# ## String Indexer

# %%
# String Index columns
indexer_input = ['PUBorough', 'DOBorough', 'time_of_day']
indexer_output = [x + '_index' for x in indexer_input]
indexer = StringIndexer(inputCols=indexer_input, outputCols=indexer_output)
```

```
# %% [markdown]
# ## One Hot Encoder

# %%
encoder_output = [x + '_encoded' for x in indexer_input]
encoder = OneHotEncoder(inputCols=indexer_output, outputCols=encoder_output)

# %% [markdown]
# # Min Max Scaler

# %%
columns_to_scale =[
   'temp',
   'feelslike',
   'precip',
   'snow_precip',
   'snowdepth',
   'windspeed',
   'uvindex'
]

# assembler for scaling
assembler_to_scale = VectorAssembler(inputCols=columns_to_scale,
outputCol='features_to_scale')

# MinMaxScaler
scaler = MinMaxScaler(inputCol='features_to_scale', outputCol='scaled_features')

# %% [markdown]
# ## Final Vector Assembler

# %%


assemble_directly = [
   'month',
   'dayofweek',
   'weekend',
   'holiday',
   'trip_distance',
   'group_size',
```

```
    'fare_amount'
]


input_cols =  encoder_output + assemble_directly + conditions + ['scaled_features']
print(input_cols)

# %%
assembler = VectorAssembler(inputCols=input_cols, outputCol='features')

# %% [markdown]
# # Save the pipeline with features

# %%
print('Saving the transformed data...')
# Create a new frame with the transformed data
pipeline = Pipeline(stages=[indexer, encoder, assembler_to_scale, scaler, assembler])
model = pipeline.fit(combined_df)
transformed_df = model.transform(combined_df)

# Save the transformed data
transformed_df.write.parquet(gs_path + destination_folder + 'features',
mode='overwrite')
print('Transformed data saved!')

# %% [markdown]
# # Train / Test

# %%
train_df, test_df = combined_df.randomSplit([0.7, 0.3], seed=42)

# %% [markdown]
# # Pipeline

# %%
lr = LogisticRegression(featuresCol='features', labelCol='tip_class', maxIter=10,
regParam=0.1, elasticNetParam=0.8)

# %%
lr_pipeline = Pipeline(stages=[
    indexer,
    encoder,
```

```
    assembler_to_scale,
    scaler,
    assembler,
    lr
])

# %% [markdown]
# # Cross validating

# %%
grid = ParamGridBuilder()
grid = grid.addGrid(lr.regParam, [0.0, 0.2, 0.4, 0.6, 0.8, 1.0])
grid = grid.addGrid(lr.elasticNetParam, [0, 0.5, 1])
grid = grid.build()

print('Number of models to be tested: ', len(grid))

# Binary classification evaluator with area under ROC as the metric
evaluator = BinaryClassificationEvaluator(labelCol='tip_class',
metricName='areaUnderROC')


cv = CrossValidator(
    estimator=lr_pipeline,
    estimatorParamMaps=grid,
    evaluator=evaluator,
    numFolds=3
)

all_models = cv.fit(train_df)



# %%
# predictions
predictions = all_models.transform(test_df)

# Calculate the AUC
auc = evaluator.evaluate(predictions)

print(f"AUC: {auc}")
```

```
# Create a confusion matrix
predictions.groupby('tip_class', 'prediction').count().show()
cm = predictions.groupby('tip_class').pivot('prediction').count().fillna(0).collect()

def calculate_metrics(cm):
   tn = cm[0][1]                # True Negative
   fp = cm[0][2]                # False Positive
   fn = cm[1][1]                # False Negative
   tp = cm[1][2]                # True Positive
   accuracy = (tp + tn) / (tp + fp + tn + fn)
   precision = tp / (tp + fp)
   recall = tp / (tp + fn)
   f1 = 2 * (precision * recall) / (precision + recall)
   return accuracy, precision, recall, f1

accuracy, precision, recall, f1 = calculate_metrics(cm)
print(f'Accuracy: {accuracy}')
print(f'Precision: {precision}')
print(f'Recall: {recall}')
print(f'F1 Score: {f1}')

# %%
# Best model
best_model = all_models.bestModel

print(f"Best Model Stages: \n{best_model.stages}")

# Parameters of the best model
best_model.stages[-1].extractParamMap()

# Create a ROC curve
trainingSummary = best_model.stages[-1].summary

plt.figure(figsize=(5, 5))
plt.plot(trainingSummary.roc.select('FPR').collect(),
        trainingSummary.roc.select('TPR').collect())
plt.xlabel('FPR')
plt.ylabel('TPR')
plt.title('ROC Curve')
plt.show()
```

```
# %%
coefficients = best_model.stages[-1].coefficients
# Print the name of the features and their corresponding coefficients
for i, col in enumerate(input_cols):
   print(f"{col}: {coefficients[i]}")


# %% [markdown]
# # Save the Model

# %%
# Save the model
print('Saving the model')
model_path = gs_path + destination_folder + 'model'
best_model.write().overwrite().save(model_path)
print('Model saved')

# %%


# %%


# %%


# %%
```