CIS 9760 Big Data Technologies
Data Engineering Project
Monirul Islam

Proposal

I will utilize the TLC Yellow Taxi Trips datasets from 2022 and 2023, which can be found at https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page. These datasets offer comprehensive information on yellow taxi rides in New York City.

Primary variables include:

- Fare-related features
  - Total_amount
  - Fare_amount
  - Extra
  - MTA_tax
  - **Tip_amount (Target Variable)**
  - Tolls_amount
  - Congestion_Surcharge
  - Airport_fee
- Trip-related features
  - Trip_distance
  - tpep_pickup_datetime
  - tpep_dropoff_datetime
- Payment method
  - Payment_type
- Passenger count
  - Passenger_count
- Location-based features
  - PULocationID
  - DOLocationID

The objective is to predict the 'tip_amount' using a linear regression machine learning model. The majority of the columns in the datasets will serve as input features as they influence the tip value.

Data Acquisition

To efficiently acquire the TLC Yellow Taxi Trips datasets from 2022 and 2023, a nano file was created on a virtual machine instance. A function was defined inside the nano file to download the Parquet files from the TLC website. The function uses a nested loop to iterate over a list of years and months, constructing file names and retrieving the Parquet files with the urlretrieve() command. The nano file was then executed via Python to download the data. A Google Cloud Storage bucket was created to store the data objects. A set of folders was initialized for different project stages, including landing, cleaned, trusted, code, and models. The downloaded Parquet files were placed into the landing folder.

Exploratory Data Analysis (EDA)

Before performing exploratory data analysis (EDA), the monthly Parquet files were aggregated into yearly dataframes, which were then saved back into the landing folder as Parquet files. In some of the monthly files for 2023, the ‘airport_fee’ column name is formatted differently from the rest, so it had to be renamed for consistency. After combining the files, several functions were developed to perform EDA efficiently. These functions differentiated the numeric and categorical features and provided respective summary statistics in a structured manner.

There are 39,656,098 records for 2022 and 38,310,226 records for 2023. Both yearly datasets contain 19 columns, with data types including integers, floats, and datetimes. Columns such as ‘passenger_count’, ‘RatecodeID’, ‘congestion_surcharge’, and ‘airport_fee’ have missing values in both datasets. Attached below are snippets of statistical outputs for numerical and categorical columns for both years:

Taxi Trip Data 2022

	Filename	Column	Minimum	Maximum	Average	Standard Deviation	Missing Values
0	landing/taxi_tripdata_2022.parquet	VendorID	1.00	6.00	1.72	0.48	0
1	landing/taxi_tripdata_2022.parquet	passenger_count	0.00	9.00	1.40	0.96	1368303
2	landing/taxi_tripdata_2022.parquet	trip_distance	0.00	389678.46	5.96	599.19	0
3	landing/taxi_tripdata_2022.parquet	RatecodeID	1.00	99.00	1.42	5.79	1368303
4	landing/taxi_tripdata_2022.parquet	PULocationID	1.00	265.00	164.87	65.31	0
5	landing/taxi_tripdata_2022.parquet	DOLocationID	1.00	265.00	162.58	70.23	0
6	landing/taxi_tripdata_2022.parquet	payment_type	0.00	5.00	1.19	0.52	0
7	landing/taxi_tripdata_2022.parquet	fare_amount	-133391414.00	401092.32	10.36	22328.30	0
8	landing/taxi_tripdata_2022.parquet	extra	-22.18	33.50	1.01	1.26	0
9	landing/taxi_tripdata_2022.parquet	mta_tax	-0.55	25.48	0.49	0.09	0
10	landing/taxi_tripdata_2022.parquet	tip_amount	-410.00	133391363.53	7.23	22328.08	0
11	landing/taxi_tripdata_2022.parquet	tolls_amount	-99.99	911.87	0.54	2.04	0
12	landing/taxi_tripdata_2022.parquet	improvement_surcharge	-1.00	1.00	0.32	0.13	0
13	landing/taxi_tripdata_2022.parquet	total_amount	-2567.80	401095.62	21.67	96.37	0
14	landing/taxi_tripdata_2022.parquet	congestion_surcharge	-2.50	2.75	2.28	0.75	1368303
15	landing/taxi_tripdata_2022.parquet	airport_fee	-1.25	1.25	0.10	0.34	1368303

	Filename	Column	Unique Values	Minimum	Maximum	Missing Values
0	landing/taxi_tripdata_2022.parquet	passenger_count	10	0.00	9.00	1368303
1	landing/taxi_tripdata_2022.parquet	PULocationID	262	1.00	265.00	0
2	landing/taxi_tripdata_2022.parquet	DOLocationID	262	1.00	265.00	0
3	landing/taxi_tripdata_2022.parquet	payment_type	6	0.00	5.00	0

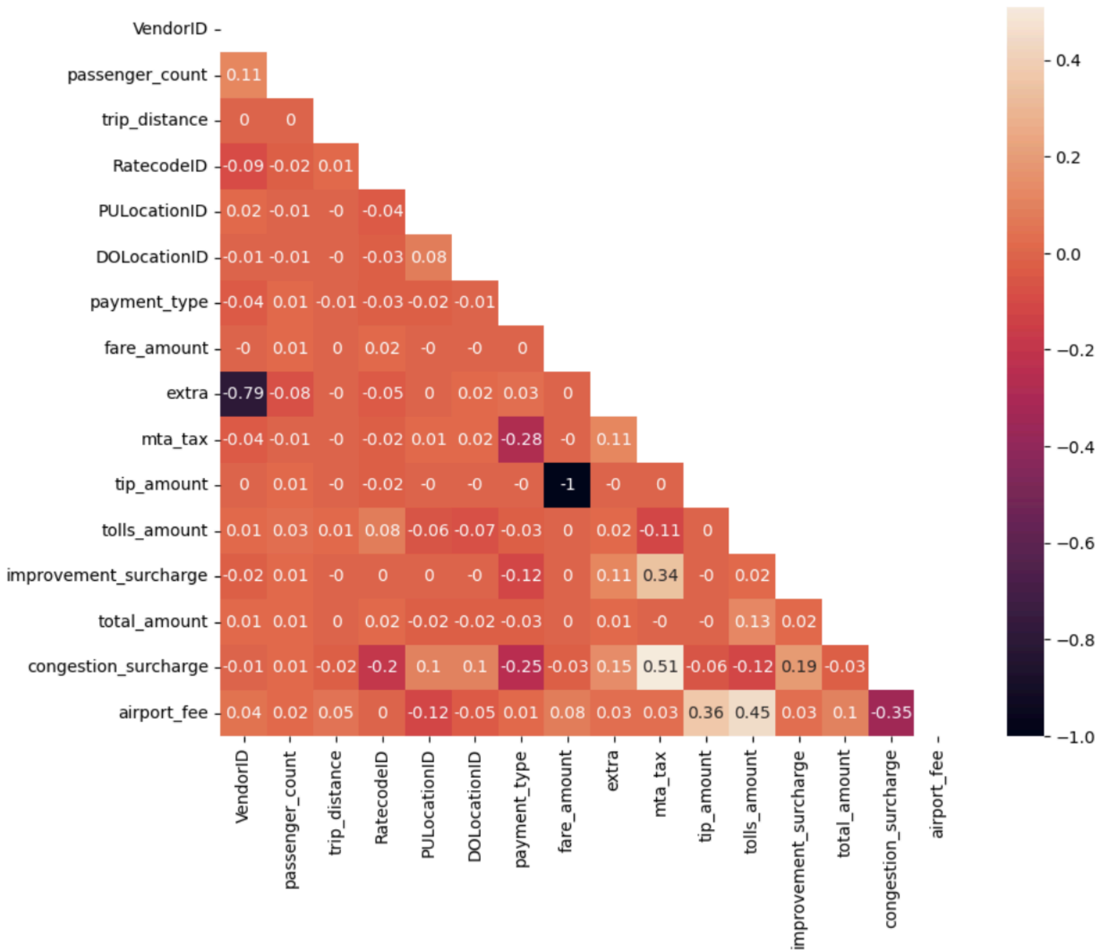
Taxi Trip Data 2023

	Filename	Column	Minimum	Maximum	Average	Standard Deviation	Missing Values
0	landing/taxi_tripdata_2023.parquet	VendorID	1.00	6.00	1.74	0.44	0
1	landing/taxi_tripdata_2023.parquet	passenger_count	0.00	9.00	1.37	0.89	1309356
2	landing/taxi_tripdata_2023.parquet	trip_distance	0.00	345729.44	4.09	241.25	0
3	landing/taxi_tripdata_2023.parquet	RatecodeID	1.00	99.00	1.64	7.43	1309356
4	landing/taxi_tripdata_2023.parquet	PULocationID	1.00	265.00	165.18	64.00	0
5	landing/taxi_tripdata_2023.parquet	DOLocationID	1.00	265.00	163.95	69.86	0
6	landing/taxi_tripdata_2023.parquet	payment_type	0.00	5.00	1.18	0.56	0
7	landing/taxi_tripdata_2023.parquet	fare_amount	-1087.30	386983.63	19.52	75.73	0
8	landing/taxi_tripdata_2023.parquet	extra	-39.17	10002.50	1.56	2.45	0
9	landing/taxi_tripdata_2023.parquet	mta_tax	-0.50	53.16	0.49	0.11	0
10	landing/taxi_tripdata_2023.parquet	tip_amount	-411.00	4174.00	3.52	4.15	0
11	landing/taxi_tripdata_2023.parquet	tolls_amount	-91.30	665.56	0.59	2.20	0
12	landing/taxi_tripdata_2023.parquet	improvement_surcharge	-1.00	1.00	0.98	0.20	0
13	landing/taxi_tripdata_2023.parquet	total_amount	-1094.05	386987.63	28.46	77.13	0
14	landing/taxi_tripdata_2023.parquet	congestion_surcharge	-2.50	2.75	2.26	0.80	1309356
15	landing/taxi_tripdata_2023.parquet	airport_fee	-1.75	1.75	0.14	0.47	1309356

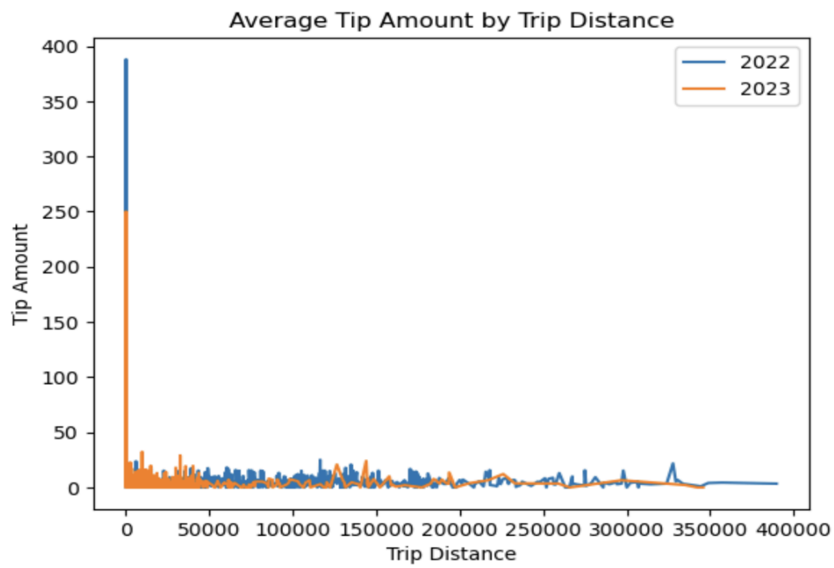
	Filename	Column	Unique Values	Minimum	Maximum	Missing Values
0	landing/taxi_tripdata_2023.parquet	passenger_count	10	0.00	9.00	1309356
1	landing/taxi_tripdata_2023.parquet	PULocationID	263	1.00	265.00	0
2	landing/taxi_tripdata_2023.parquet	DOLocationID	262	1.00	265.00	0
3	landing/taxi_tripdata_2023.parquet	payment_type	6	0.00	5.00	0

Based on the summary statistics, extreme outliers are present across both years. For example, some numerical columns contain negative float values, which would be illogical for modeling in this scenario. The maximum trip distance on record is approximately 390,000 miles, with the total amount nearing $400,000 – both values being unrealistic and could be due to incorrect inputs. Therefore, it is necessary to filter these records out for a better model.

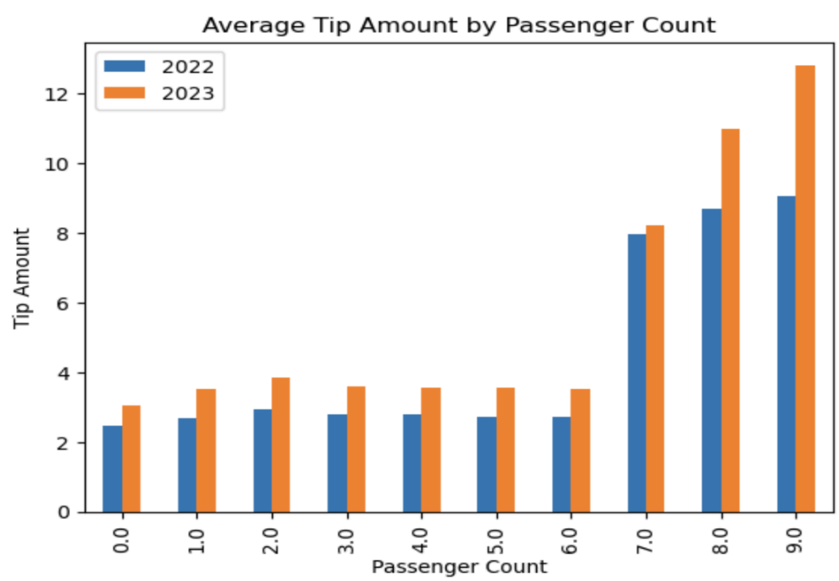
Additionally, visualizations were created during the EDA process to observe the relationships between variables in more depth.



The correlation matrix above reveals that none of the variables has a strong positive relationship. However, ‘mta_tax’ and ‘congestion_surcharge’ show a moderate positive correlation. The columns ‘tip_amount’ and ‘airport_fee’, as well as ‘tolls_amount’ and ‘airport_fee’, both exhibit weak positive correlations. Meanwhile, the columns ‘extra’ and ‘VendorID’ display a strong negative correlation.



The line chart above illustrates the average tip amount by the distance traveled. Based on the range of the distance axis and the concentration of the data points, it is evident that outliers are present and need to be excluded for an optimal model. The chart indicates that tipping amounts do range as distance increases.



The bar chart above shows the average tip amount by passenger count. As the number of passengers increases, the tip amount increases, which can be attributed to an increase in perceived workload, increased risk, and social norms. The average tip amount per passenger count is higher in 2023 compared to 2022.

Data Cleanup

The data cleaning process involves removing unnecessary columns (i.e., ‘store_and_fwd_flag’), dropping duplicates, and filtering out extreme outliers for model optimization and performance. Since the tip amount field of the datasets is only populated for credit card tips and the objective is to use a linear regression to predict the tip amount, the payment method column is filtered to only account for such payments. Other filters applied include at least one passenger, trip distance to be between 0.1 and 50 miles, fare amount of at least $3, which is the base fare in NYC, to $250, tip amount between $0 and $250, and extra charges greater than $0. Ultimately, after cleaning, both datasets had zero null values, zero duplicates, and consisted of records with realistic taxi ride data for model training and testing. The cleaned files were then saved as Parquet files and uploaded to the cleaned folder of the bucket.

Feature Engineering

Prior to the modeling process, feature engineering is required to transform the raw data into meaningful features that will improve the model’s accuracy, reduce overfitting, and enhance interpretability. The cleaned taxi trip data Parquet files were loaded into a PySpark environment and merged for comprehensive feature engineering. The “total_amount” column was dropped to prevent data leakage, as it included the target variable. Several new features were engineered, including:

- “trip_duration”: the trip interval in minutes
- “day_of_week”: the day of the week the trip occurred
- “hour_of_day”: the hour of the day the trip began
- “season”: the season during which the trip took place

These features were created to capture patterns in tipping behavior based on temporal factors such as time of day, week, and seasonality. The columns “VendorID", "passenger_count", "RatecodeID", "PULocationID", "DOLocationID", "payment_type", "day_of_week", "hour_of_day", and "season" were treated as categorical variables. They were transformed using StringIndexer to assign a numeric index to each unique category, and then One-Hot encoded. The continuous variables "trip_distance", "trip_duration", "fare_amount", "mta_tax", "congestion_surcharge", "airport_fee", "tolls_amount", "extra", and "improvement_surcharge" were scaled using MinMaxScaler to normalize their values within the range of 0.0 to 1.0. Finally, all categorical and continuous variables were combined into a single feature vector using VectorAssembler. The transformed dataframe was saved in the trusted folder of the project bucket.

Modeling

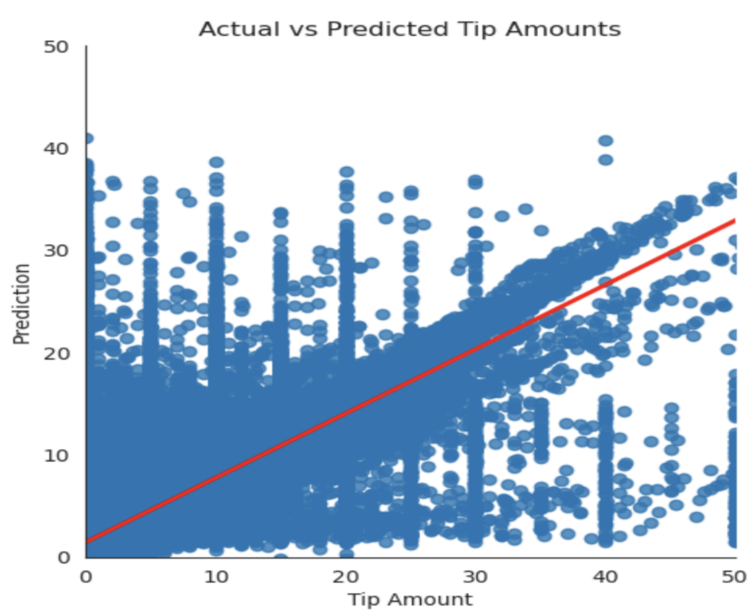
The transformed dataframe was uploaded into a new PySpark environment specifically for modeling and the hyperparameter tuning process. 70 percent of the data was allocated for training, while the remaining 30 percent was reserved for testing. A linear regression model was instantiated and trained on the training set. Subsequently, the model generated predictions utilizing the test set. To evaluate the model's predictions against the actual tip amounts, the root mean squared error (RMSE) and R-squared (R2) metrics were computed. The model achieved an RMSE of 2.18 and an R2 of 0.63, which is decent but indicates room for improvement. Hyperparameters were adjusted using a grid search approach to achieve better evaluation scores. A range of values for the regularization parameters, including regParam (controlling the strength of penalization) and elasticNetParam (balancing between L1 and L2 regularization), was explored. A cross-validation process, combined with the hyperparameter grid, was used to identify the optimal model. Despite training multiple models and selecting the best one, the evaluation metrics remained the same as those of the original model. Some notable feature coefficients from the best model include:

- "trip_duration" (0.92): Longer trip durations are associated with higher tip amounts.
- "hour_of_day" (16.81): Tipping behavior varies by time of day, likely influenced by factors such as rush hours and nighttime activity.
- "fare_amount" (0.69): Higher fares, resulting from longer distances and durations, are correlated with higher tips.
- "congestion_surcharge" (-5.10): Additional charges such as congestion surcharges tend to discourage tipping behavior.
- "passenger_count" (0.01): Passenger count has insignificant predictive value for tip amounts.

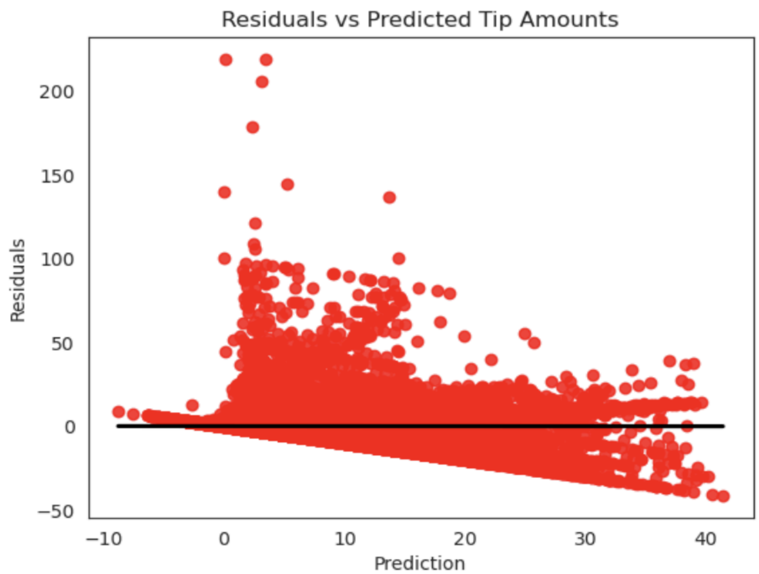
The best linear regression model was saved in the models folder of the project bucket.

Data Visualization

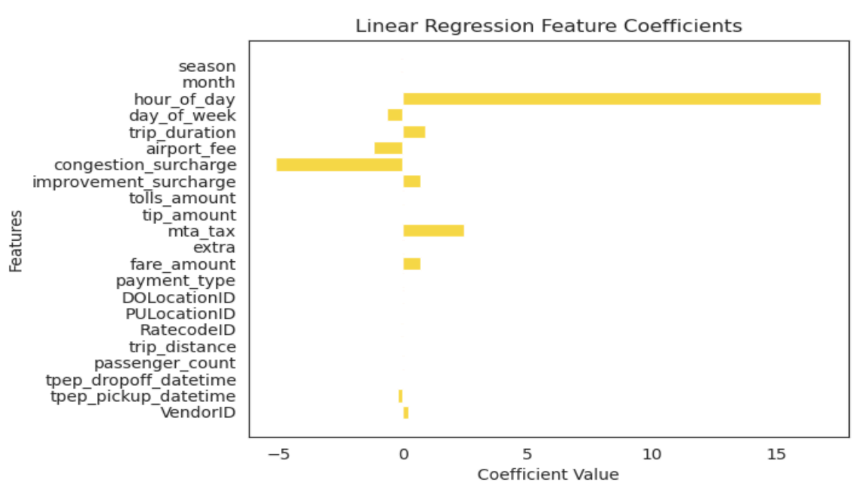
Several visualizations were created below to highlight the results of the best linear regression model and provide supplementary analysis.



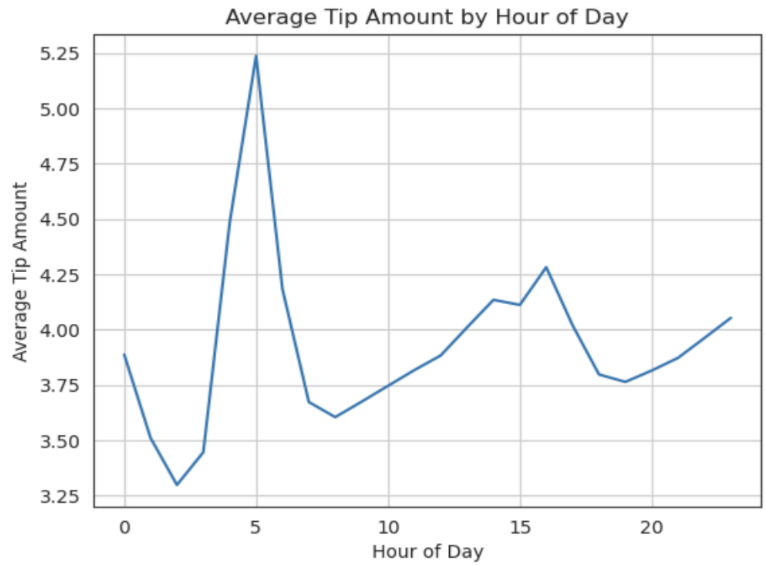
The visualization above illustrates tip amounts by plotting the predicted and actual values. It is important to note that the plot randomly samples ten percent of the data without replacement for faster rendering and to reduce overplotting. Both axes are limited for consistent comparison and to focus on the most clustered area. The red line indicates the best-fit regression line. The data points are scattered around the line, indicating average prediction performance.



The next visualization above depicts the residuals, which are the errors, versus the predicted tip amounts. Again, a ten percent sample without replacement was incorporated for faster rendering, given a substantial number of data points. In an ideal scenario, the residuals would be centered and evenly spread around zero. But in this case, while the residuals are roughly centered, they are not consistently spread. This could stem from several factors, including greater variability in high-tipping circumstances, skewed tip distributions, or even limitations of the current model.



The third visualization highlights the impact of the feature coefficients on the predicted tip amount. To reiterate some of the information from the modeling section, the most influential feature is the "hour_of_day," suggesting that tipping behavior increases for certain periods of the day, such as nighttime. In contrast, "congestion_surcharge" adversely affects tipping behavior as consumers could be dissatisfied with congested traffic.



To further analyze the leading coefficient, "hour_of_day," a line plot was constructed to show the average tip amounts across the hours of the day. The peak average tip amount occurs around 5 AM, possibly due to early runs to the airport or smoother rides with less traffic. There is a sharp decline and low tipping behavior observed between 6 AM and 9 AM, which coincides with the normal morning commute to work and school. Tip amounts gradually increase throughout midday hours. Similarly, tipping behavior increases during the evening as people engage in nightlife activities and tend to feel more generous afterwards.

Conclusion

This project demonstrates a data engineering pipeline and predictive model to estimate taxi tip amounts utilizing the TLC Yellow Taxi Trip datasets from 2022 and 2023. The workflow begins with data acquisition, automating the downloading of Parquet files from the TLC website and storing them in a Google Cloud Storage bucket. Next, exploratory data analysis was performed to understand the data and generate descriptive statistics of the variables. Extensive data cleaning was conducted to remove outliers and to ensure the dataset mostly comprised realistic taxi rides. The cleaned data was further processed during feature engineering in a PySpark environment. New temporal features were created to enhance the dataset for achieving better model evaluation scores. Categorical variables were indexed and one-hot encoded, while continuous variables were normalized. The prepared data was trained on a linear regression model, which was evaluated on metrics of RMSE and R-squared. The model achieved an RMSE of 2.18 and an R-squared of 0.63, indicating moderate performance. During hyperparameter tuning, grid search and cross-validation were applied to optimize results, though performance remained unchanged.

According to the best model, the most important features were “hour_of_day,” which had a substantial positive impact on tip amounts, whereas “congestion_surcharge” had a negative influence. Several visualizations were produced, highlighting insights into prediction accuracy, residual patterns, feature importance, and temporal trends. To further improve the project analysis in predicting taxi tip amounts, other advanced models could be explored, like non-linear ones. Another method is to engineer additional relevant features to provide the model with more context.

Appendix A (Code for Data Acquisition)

```
# Create a nano file
nano download_yellow.py

# Create a function to download "TLC Yellow Taxi Trips" parquet files within the nano file
from urllib.request import urlretrieve
years_list = ['2022', '2023']
months_list = ['01', '02', '03', '04', '05', '06', '07', '08', '09', '10', '11', '12']

for year in years_list:
    for month in months_list:
        filename = f"yellow_tripdata_{year}-{month}.parquet"
        url = f"https://d37ci6vzurychx.cloudfront.net/trip-data/{filename}"
        print(f"Working on file: {filename}")
        urlretrieve(url, filename)

# Save the nano file and exit shell
Ctrl + o
Enter
Ctrl + x

# Execute the nano file with Python
python3 download_yellow.py

# Create a bucket on Google Cloud Storage (GCS)
gcloud storage buckets create gs://project-bucket --project=your-gcp-project-id
--default-storage-class=STANDARD --location=us-central1 --uniform-bucket-level-access

# Copy the parquet files to GCS bucket in the /landing folder
gcloud storage cp yellow_tripdata_*.parquet gs://project-bucket/landing

# Create empty folders
gcloud storage cp /dev/null gs://project-bucket/cleaned
gcloud storage cp /dev/null gs://project-bucket/trusted
gcloud storage cp /dev/null gs://project-bucket/code
gcloud storage cp /dev/null gs://project-bucket/models
```

Appendix B (Code for Exploratory Data Analysis)

```
# Create a single node cluster
gcloud dataproc clusters create cluster-1234 --enable-component-gateway --region us-central1
--single-node --master-machine-type n2-standard-8 --master-boot-disk-type pd-balanced
--master-boot-disk-size 100 --image-version 2.2-debian12 --optional-components JUPYTER
--max-idle 7200s --project your-gcp-project-id
```

Jupyter Notebook 1 - Aggregating Monthly Parquet Files Into Yearly Files

```
# Import libraries and modules
from google.cloud import storage
from io import StringIO, BytesIO
import pyarrow
import fastparquet
import pandas as pd
import dask.dataframe as dd
pd.set_option('display.float_format', '{:.2f}'.format)
pd.set_option('display.width', 5000)

# Create a client object that points to GCS
storage_client = storage.Client()
# Point to the bucket
bucket_name = 'project-bucket'
bucket = storage_client.get_bucket(bucket_name)

# Aggregate year 2022 Parquet files into one DataFrame
# Get a list of the 'blobs' (objects or files) in the bucket
blobs = storage_client.list_blobs(bucket_name, prefix="landing/")
# Iterate through the list and print out their names
parquet_blobs = [blob for blob in blobs if blob.name.endswith('.parquet') and
"yellow_tripdata_2022" in blob.name]
# Initialize an empty list to collect dataframes
df_list = []
# Loop through each Parquet blob and read it into a DataFrame
for blob in parquet_blobs:
    # Note the use of BytesIO and .download_as_bytes() function
    df = pd.read_parquet(BytesIO(blob.download_as_bytes()))
    # Append to list
    df_list.append(df)
```

```
# Concatenate all DataFrames into one
taxi_tripdata_2022 = pd.concat(df_list, ignore_index=True)
# Confirm the result
print(f"Combined DataFrame shape: {taxi_tripdata_2022.shape}")

# Uploading aggregated file back to the ‘landing/’ folder
# Create new filename
new_filename = "landing/taxi_tripdata_2022.parquet"
# Convert the DataFrame to a Parquet byte string
filedata = taxi_tripdata_2022.to_parquet(index=False)
# Create a new blob and upload the file
new_blob = bucket.blob(new_filename)
new_blob.upload_from_string(filedata, content_type='application/octet-stream')

# Aggregate year 2023 Parquet files into one DataFrame
# Get a list of the 'blobs' (objects or files) in the bucket
blobs = storage_client.list_blobs(bucket_name, prefix="landing/")
# Iterate through the list and print out their names
parquet_blobs = [blob for blob in blobs if blob.name.endswith('.parquet') and
"yellow_tripdata_2023" in blob.name]
# Initialize an empty list to collect dataframes
df_list = []
# Loop through each Parquet blob and read it into a DataFrame
for blob in parquet_blobs:
    # Note the use of BytesIO and .download_as_bytes() function
    df = pd.read_parquet(BytesIO(blob.download_as_bytes()))
    # Renaming the ‘Airport_fee’ column to ‘airport_fee’
    if "Airport_fee" in df.columns:
        df = df.rename(columns={"Airport_fee": "airport_fee"})
    # Append to list
    df_list.append(df)
# Concatenate all DataFrames into one
taxi_tripdata_2023 = pd.concat(df_list, ignore_index=True)
# Confirm the result
print(f"Combined DataFrame shape: {taxi_tripdata_2023.shape}")

# Uploading aggregated file back to the ‘landing/’ folder
# Create new filename
new_filename = "landing/taxi_tripdata_2023.parquet"
# Convert the DataFrame to a Parquet byte string
```

```
filedata = taxi_tripdata_2023.to_parquet(index=False)
# Create a new blob and upload the file
new_blob = bucket.blob(new_filename)
new_blob.upload_from_string(filedata, content_type='application/octet-stream')
```

Jupyter Notebook 2 - EDA

```
# Import libraries and modules
from google.cloud import storage
from io import StringIO, BytesIO
import pyarrow
import fastparquet
import pandas as pd
import numpy as np
import dask.dataframe as dd
pd.set_option('display.float_format', '{:.2f}'.format)
pd.set_option('display.width', 5000)
import matplotlib.pyplot as plt
import seaborn as sns

# Create a client object that points to GCS
storage_client = storage.Client()
# Point to the bucket
bucket_name = 'project-bucket'
bucket = storage_client.get_bucket(bucket_name)

# Create functions to perform EDA
def perform_EDA(df : pd.DataFrame, filename : str):
    """
    perform_EDA(df : pd.DataFrame, filename : str)
    Accepts a dataframe and a text filename as inputs.
    Runs some basic statistics on the data and outputs to console.

    :param df: The Pandas dataframe to explore
    :param filename: The name of the data file
    :returns:
    """
    print(f"{filename} Number of records:")
    print(df.count())
    number_of_duplicate_records = df.duplicated().sum()
```

```python
    print(f"{filename} Number of duplicate records: {number_of_duplicate_records}" )
    print(f"{filename} Info")
    print(df.info())
    print(f"{filename} Describe")
    print(df.describe())
    print(f"{filename} Columns with null values")
    print(df.columns[df.isnull().any()].tolist())
    rows_with_null_values = df.isnull().any(axis=1).sum()
    print(f"{filename} Number of Rows with null values: {rows_with_null_values}" )
    integer_column_list = df.select_dtypes(include='int64').columns
    print(f"{filename} Integer data type columns: {integer_column_list}")
    float_column_list = df.select_dtypes(include='float64').columns
    print(f"{filename} Float data type columns: {float_column_list}")

def perform_EDA_numeric(df : pd.DataFrame, filename : str):
    """
    perform_EDA_numeric(df : pd.DataFrame, filename : str)
    Accepts a dataframe and a text filename as inputs.
    Runs some basic statistics on numeric columns and saves the output in a dataframe.

    :param df: The Pandas dataframe to explore
    :param filename: The name of the data file
    :returns:
    :   pd.DataFrame: A new dataframe with summary statistics
    """
    # Initialize a list to collect summary data
    summary_data = []
    # Gather summary statistics on numeric columns
    for col in df.select_dtypes(include=['int64', 'float64']).columns:
        summary_data.append({
            'Filename': filename,        'Column': col,
            'Minimum': df[col].min(),     'Maximum': df[col].max(),
            'Average': df[col].mean(),    'Standard Deviation': df[col].std(),
            'Missing Values': df[col].isnull().sum()
        })
    # Convert the summary data list into a DataFrame
    return pd.DataFrame(summary_data)
```

```
def perform_EDA_categorical(df : pd.DataFrame, filename : str, categorical_columns):
    """
    perform_EDA_categorical(df : pd.DataFrame, filename : str, categorical_columns)
    Accepts a dataframe and a text filename as inputs.
    Collects statistics on Categorical columns

    :param df: The Pandas dataframe to explore
    :param filename: The name of the data file
    :param categorical_columns: A list of column names for categorical columns
    :returns:
    :   pd.DataFrame: A new dataframe with summary statistics
    """
    # Initialize a list to collect summary data
    summary_data = []
    # Gather summary statistics on numeric columns
    for col in categorical_columns:
      summary_data.append({
          'Filename': filename,
          'Column': col,
          'Unique Values': df[col].apply(lambda x: tuple(x) if isinstance(x, list) else x).nunique(),
          'Minimum': df[col].min(),
          'Maximum': df[col].max(),
          'Missing Values': df[col].isnull().sum()
      })
    # Convert the summary data list into a DataFrame
    return pd.DataFrame(summary_data)

# Create a function to apply EDA functions
def main_taxi():
    categorical_columns_list = ["passenger_count", "PULocationID", "DOLocationID",
"payment_type"]

    # Get a list of the 'blobs' (objects or files) in the bucket
    blobs = storage_client.list_blobs(bucket_name, prefix="landing/")

    # Iterate through the list and process 'taxi_tripdata_2022.parquet' and
'taxi_tripdata_2023.parquet'
    parquet_blobs = [blob for blob in blobs if blob.name.endswith('.parquet') and "taxi_tripdata_"
in blob.name]
    for blob in parquet_blobs:
```

```python
        # Read in the Parquet file from the blob
        df = pd.read_parquet(BytesIO(blob.download_as_bytes()))
        perform_EDA(df, blob.name)

        # Gather the statistics on numeric columns
        numeric_summary_df = perform_EDA_numeric(df, blob.name)
        print(numeric_summary_df.head(24))

        # Gather statistics on the categorical columns
        categorical_summary_df = perform_EDA_categorical(df, blob.name,
categorical_columns_list)
        print(categorical_summary_df.head(24))

if __name__ == "__main__":
    main_taxi()
```

Jupyter Notebook 2 Continued - EDA Visualizations

```python
# Get a list of the 'blobs' (objects or files) in the bucket
blobs = storage_client.list_blobs(bucket_name, prefix="landing/")
# Iterate and load each file into a DataFrame
for blob in blobs:
    if blob.name.endswith('.parquet') and "taxi_tripdata_" in blob.name:
        if "2022" in blob.name:
            df_2022 = pd.read_parquet(BytesIO(blob.download_as_bytes()))
        elif "2023" in blob.name:
            df_2023 = pd.read_parquet(BytesIO(blob.download_as_bytes()))

# Correlation matrix for variables in df_2022
plt.figure(figsize=(10,8))
correlation_matrix = df_2022.corr(numeric_only=True).round(2)
mask = np.zeros_like(correlation_matrix, dtype=bool)
mask[np.triu_indices_from(mask)] = True
sns.heatmap(correlation_matrix, annot = True, mask=mask)
plt.show()

# Visualization of the average tip amount by trip distance
avg_tip_2022 = df_2022.groupby('trip_distance')['tip_amount'].mean()
avg_tip_2023 = df_2023.groupby('trip_distance')['tip_amount'].mean()
```

```python
avg_tip_2022.plot(kind='line', label='2022')
avg_tip_2023.plot(kind='line', label='2023')
plt.xlabel('Trip Distance')
plt.ylabel('Tip Amount')
plt.title('Average Tip Amount by Trip Distance')
plt.legend()
plt.show()

# Visualization of the average tip amount by passenger count
pd.concat([
   df_2022.groupby('passenger_count')['tip_amount'].mean().rename('2022'),
   df_2023.groupby('passenger_count')['tip_amount'].mean().rename('2023')],
axis=1).plot(kind='bar')
plt.xlabel('Passenger Count')
plt.ylabel('Tip Amount')
plt.title('Average Tip Amount by Passenger Count')
plt.legend()
plt.show()
```

Appendix C (Code for Data Cleanup)

Jupyter Notebook 3 - Data Cleanup

```
# Import libraries and modules
from google.cloud import storage
from io import StringIO, BytesIO
import pyarrow
import fastparquet
import pandas as pd
import dask.dataframe as dd
pd.set_option('display.float_format', '{:.2f}'.format)
pd.set_option('display.width', 5000)

# Create a client object that points to GCS
storage_client = storage.Client()
# Point to the bucket
bucket_name = 'project-bucket'
bucket = storage_client.get_bucket(bucket_name)

# Get a list of the 'blobs' (objects or files) in the bucket
blobs = storage_client.list_blobs(bucket_name, prefix="landing/")
# Iterate and load each file into a DataFrame
for blob in blobs:
    if blob.name.endswith('.parquet') and "taxi_tripdata_" in blob.name:
        if "2022" in blob.name:
            df_2022 = pd.read_parquet(BytesIO(blob.download_as_bytes()))
        elif "2023" in blob.name:
            df_2023 = pd.read_parquet(BytesIO(blob.download_as_bytes()))

# Cleanup taxi trip data 2022
# Drop the unnecessary column
df_2022.drop('store_and_fwd_flag', axis=1, inplace=True)
# Filter process
df_2022 = df_2022[
    (df_2022['payment_type'] == 1) &
    (df_2022['passenger_count'] > 0) &
    (df_2022['trip_distance'] >= 0.1) & (df_2022['trip_distance'] <= 50) &
    (df_2022['fare_amount'] >= 3) & (df_2022['fare_amount'] <= 250) &
    (df_2022['tip_amount'] >= 0) & (df_2022['tip_amount'] <= 250) &
```

```
    (df_2022['extra'] >= 0)
]
# Drop duplicates
df_2022.drop_duplicates(inplace=True)
# EDA of cleaned-up DataFrame
df_2022.shape
df_2022.isna().sum()
df_2022.info()
df_2022.describe()

# Uploading cleaned file to the ‘cleaned/’ folder
# Create new filename
new_filename = "cleaned/taxi_tripdata_2022_clean.parquet"
# Convert the DataFrame to a Parquet byte string
filedata = df_2022.to_parquet(index=False)
# Create a new blob and upload the file
new_blob = bucket.blob(new_filename)
new_blob.upload_from_string(filedata, content_type='application/octet-stream')

# Cleanup taxi trip data 2023
# Drop the unnecessary column
df_2023.drop('store_and_fwd_flag', axis=1, inplace=True)
# Filter process
df_2023 = df_2023[
    (df_2023['payment_type'] == 1) &
    (df_2023['passenger_count'] > 0) &
    (df_2023['trip_distance'] >= 0.1) & (df_2023['trip_distance'] <= 50) &
    (df_2023['fare_amount'] >= 3) & (df_2023['fare_amount'] <= 250) &
    (df_2023['tip_amount'] >= 0) & (df_2023['tip_amount'] <= 250) &
    (df_2023['extra'] >= 0)
]
# Drop duplicates
df_2023.drop_duplicates(inplace=True)
# EDA of cleaned-up DataFrame
df_2023.shape
df_2023.isna().sum()
df_2023.info()
df_2023.describe()

# Uploading cleaned file to the ‘cleaned/’ folder
# Create new filename
```

```
new_filename = "cleaned/taxi_tripdata_2023_clean.parquet"
# Convert the DataFrame to a Parquet byte string
filedata = df_2023.to_parquet(index=False)
# Create a new blob and upload the file
new_blob = bucket.blob(new_filename)
new_blob.upload_from_string(filedata, content_type='application/octet-stream')
```

Appendix D (Code for Feature Engineering and Modeling)

PySpark Jupyter Notebook 1 - Feature Engineering

```
# Create a 5-node cluster
gcloud dataproc clusters create cluster-a8fe --enable-component-gateway --region us-central1
--master-machine-type n2-standard-4 --master-boot-disk-type pd-balanced
--master-boot-disk-size 100 --num-workers 4 --worker-machine-type n2-standard-4
--worker-boot-disk-type pd-balanced --worker-boot-disk-size 100 --image-version 2.2-debian12
--optional-components JUPYTER --max-idle 7200s --project your-gcp-project-id

# Import libraries
import pandas as pd
from pyspark.sql.functions import col, hour, dayofweek, month, when, expr
from pyspark.ml.feature import StringIndexer, OneHotEncoder, VectorAssembler,
MinMaxScaler
from pyspark.ml import Pipeline

# Set up the path to a file
bucket = 'project-bucket'
cleaned_folder = f"gs://{bucket}/cleaned"

taxi_2022 = f"{cleaned_folder}/taxi_tripdata_2022_clean.parquet"
taxi_2023 = f"{cleaned_folder}/taxi_tripdata_2023_clean.parquet"

# Read the parquet files into a Spark DataFrame
df = spark.read.parquet(taxi_2022, taxi_2023)

# Get the number of records in the dataframe
df.count()

# Check the schema
df.printSchema()

# Drop target-leaking column
df = df.drop("total_amount")

# Derive trip duration (in minutes)
df = df.withColumn("trip_duration", expr("timestampdiff(MINUTE, tpep_pickup_datetime,
tpep_dropoff_datetime)"))
```

```
# Derive the day of the week
df = df.withColumn("day_of_week", dayofweek("tpep_pickup_datetime"))

# Derive the hour of the day
df = df.withColumn("hour_of_day", hour("tpep_pickup_datetime"))

# Derive seasons
df = df.withColumn("month", month("tpep_pickup_datetime"))
df = df.withColumn("season", when(col("month").isin(12, 1, 2), "winter")
                                        .when(col("month").isin(3, 4, 5), "spring")
                                        .when(col("month").isin(6, 7, 8), "summer")
                                        .when(col("month").isin(9, 10, 11), "fall"))

# Check the schema
df.printSchema()

# Create an indexer for categorical features
indexer = StringIndexer(inputCols=["VendorID", "passenger_count", "RatecodeID",
"PULocationID", "DOLocationID", "payment_type", "day_of_week", "hour_of_day", "season"],
outputCols=["VendorID_idx", "passenger_count_idx", "RatecodeID_idx", "PULocationID_idx",
"DOLocationID_idx", "payment_type_idx", "day_of_week_idx", "hour_of_day_idx",
"season_idx"], handleInvalid="keep")
# Create an encoder for the indexes
encoder = OneHotEncoder(inputCols=["VendorID_idx", "passenger_count_idx",
"RatecodeID_idx", "PULocationID_idx", "DOLocationID_idx", "payment_type_idx",
"day_of_week_idx", "hour_of_day_idx", "season_idx"],
 outputCols=["VendorID_vec", "passenger_count_vec", "RatecodeID_vec",
"PULocationID_vec", "DOLocationID_vec", "payment_type_vec", "day_of_week_vec",
"hour_of_day_vec", "season_vec"], dropLast=True, handleInvalid="keep")

# Scale numerical variables using MinMax
columns_to_scale = ["trip_distance", "trip_duration", "fare_amount", "mta_tax",
"congestion_surcharge", "airport_fee", "tolls_amount", "extra", "improvement_surcharge"]
assembler = [VectorAssembler(inputCols=[col], outputCol=col + "_vec") for col in
columns_to_scale]
scalers = [MinMaxScaler(inputCol=col + "_vec", outputCol=col + "_scaled") for col in
columns_to_scale]

# Assemble all of the vectors together into one large vector
```

```
combined_assembler = VectorAssembler(inputCols=["VendorID_vec", "passenger_count_vec",
"RatecodeID_vec", "PULocationID_vec", "DOLocationID_vec", "payment_type_vec",
"day_of_week_vec", "hour_of_day_vec", "season_vec", "trip_distance_scaled",
"trip_duration_scaled", "fare_amount_scaled", "mta_tax_scaled",
"congestion_surcharge_scaled", "airport_fee_scaled", "tolls_amount_scaled", "extra_scaled",
"improvement_surcharge_scaled"], outputCol="features")

# Build the pipeline with all of the stages
pipeline = Pipeline(stages=[indexer, encoder] + assembler + scalers + [combined_assembler])

# Call .fit to transform the data
transformed_df = pipeline.fit(df).transform(df)
transformed_df.show()

# Save features to 'trusted/' folder
trusted_folder = f"gs://{bucket}/trusted"
transformed_df.write.mode("overwrite").parquet(trusted_folder)
```

PySpark Jupyter Notebook 2 - Modeling

```
# Import libraries
from pyspark.sql.functions import *
from pyspark.ml import Pipeline
from pyspark.ml.regression import LinearRegression, GeneralizedLinearRegression
from pyspark.ml.evaluation import BinaryClassificationEvaluator, RegressionEvaluator
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
import pandas as pd
import numpy as np

# Set up the path to a file
bucket = "project-bucket"
trusted_folder = f"gs://{bucket}/trusted/"

# Load the Parquet dataset
data = spark.read.parquet(trusted_folder)
data.printSchema()
data.count()

# Split the data into training and test sets
training_data, test_data = data.randomSplit([0.70, 0.3], seed=42)
```

```
# Create a Linear Regression Estimator
lr = LinearRegression(featuresCol='features', labelCol='tip_amount')
# Create a regression evaluator (to get RMSE, R^2, RME, etc.)
evaluator = RegressionEvaluator(labelCol='tip_amount')
# Train model
lr_model = lr.fit(training_data)
# Make predictions
test_results = lr_model.transform(test_data)
test_results.select("tip_amount", "prediction").show(truncate=False)
# Calculate RMSE and R^2
rmse = evaluator.evaluate(test_results, {evaluator.metricName:'rmse'})
r2 =evaluator.evaluate(test_results,{evaluator.metricName:'r2'})
print(f"RMSE: {rmse:.2f}  R-squared:{r2:.2f}")

# Experimentation with different hyperparameters, regularization techniques, and cross-folds
# Create the pipeline
regression_pipe = Pipeline(stages=[lr])
# Create a grid to hold hyperparameters
grid = ParamGridBuilder()
grid = grid.addGrid(lr.regParam, [0.0, 0.5, 1.0])
grid = grid.addGrid(lr.elasticNetParam, [0, 1])
# Build the grid
grid = grid.build()
print('Number of models to be tested: ', len(grid))
# Create the CrossValidator using the hyperparameter grid
cv = CrossValidator(estimator=regression_pipe, estimatorParamMaps=grid, evaluator=evaluator,
numFolds=3)
# Train the models
all_models = cv.fit(training_data)
# Show the average performance over the three folds for each grid combination
print(f"Average metric {all_models.avgMetrics}")
# Get the best model from all of the models trained
bestModel = all_models.bestModel
# Use the model 'bestModel' to predict the test set
test_results = bestModel.transform(test_data)
test_results.select("tip_amount", "prediction").show(truncate=False)
# Calculate RMSE and R2
rmse = evaluator.evaluate(test_results, {evaluator.metricName:'rmse'})
r2 =evaluator.evaluate(test_results,{evaluator.metricName:'r2'})
```

```
print(f"RMSE: {rmse:.2f}  R-squared:{r2:.2f}")
# Obtain coefficients from the best model
coefficients = bestModel.stages[-1].coefficients
for i in range(len(test_data.columns)-1):
    print(test_data.columns[i],coefficients[i])

# Save the best model to 'models/' folder
model_folder = f"gs://{bucket}/model/lr_best_model"
bestModel.write().overwrite().save(model_folder)
```

Appendix E (Code for Data Visualization)

PySpark Jupyter Notebook 2 Continued

```
# Create a 5-node cluster
gcloud dataproc clusters create cluster-a8fe --enable-component-gateway --region us-central1
--master-machine-type n2-standard-4 --master-boot-disk-type pd-balanced
--master-boot-disk-size 100 --num-workers 4 --worker-machine-type n2-standard-4
--worker-boot-disk-type pd-balanced --worker-boot-disk-size 100 --image-version 2.2-debian12
--optional-components JUPYTER --max-idle 7200s --project your-gcp-project-id

# Import libraries
import matplotlib.pyplot as plt
import seaborn as sns

# Visual 1 - Actual vs Predicted Tip Amounts
# Select and convert to a Pandas dataframe
df = test_results.select('tip_amount','prediction').sample(False, 0.1).toPandas()
# Set the style for Seaborn plots
sns.set_style("white")
# Create a relationship plot between tip and prediction
plot = sns.lmplot(x='tip_amount', y='prediction', data=df, line_kws={'color': 'red'})
plt.title("Actual vs Predicted Tip Amounts")
plt.xlabel('Tip Amount')
plt.ylabel('Prediction')
plt.xlim(0, 50)
plt.ylim(0, 50)
plt.show

# Visual 2 - Residuals vs Predicted Tip Amounts
df = test_results.select('tip_amount','prediction').sample(False, 0.1).toPandas()
df['residuals'] = df['tip_amount'] - df['prediction']
# Set the style for Seaborn plots
sns.set_style("white")
# Create a relationship plot between residuals and predictions
sns.regplot(x = 'prediction', y = 'residuals', data = df, scatter = True, color = 'red',
line_kws={'color': 'black'})
plt.title('Residuals vs Predicted Tip Amounts')
plt.xlabel('Prediction')
plt.ylabel('Residuals')
```

```python
plt.show()

# Visual 3 - Feature Coefficient Bar Chart
# Feature names and their corresponding coefficients
features = [
    'VendorID', 'tpep_pickup_datetime', 'tpep_dropoff_datetime',
    'passenger_count', 'trip_distance', 'RatecodeID', 'PULocationID',
    'DOLocationID', 'payment_type', 'fare_amount', 'extra', 'mta_tax',
    'tip_amount', 'tolls_amount', 'improvement_surcharge', 'congestion_surcharge',
    'airport_fee', 'trip_duration', 'day_of_week', 'hour_of_day', 'month', 'season'
]
coefficients = [
    0.2009109535913344, -0.20091093275230473, 0.0,
    0.01170130794148016, -0.005276810476561724, -0.04640964252238373,
    -0.031651806117211544, 0.02831197747443519, 0.0357961671942197,
    0.6910186539643141, 0.010668454235577194, 2.4505831043804487,
    0.0, 0.05018010741297196, 0.6871002558106704, -5.10459065979098,
    -1.2002454043094202, 0.9198714677503868, -0.6265799959165672,
    16.8059146690989, 0.0, -0.08172811369509314
]
# Plot
plt.figure()
plt.barh(features, coefficients, color='gold')
plt.title('Linear Regression Feature Coefficients')
plt.xlabel('Coefficient Value')
plt.ylabel('Features')
plt.show()

# Visual 4 - Line Plot of Mean Tip by Hour
# Group by hour and compute the mean
hourly_avg = test_results.groupBy('hour_of_day').avg('tip_amount').toPandas()
# Rename column
hourly_avg.columns = ['hour_of_day', 'avg_tip_amount']
# Plot
sns.lineplot(x='hour_of_day', y='avg_tip_amount', data=hourly_avg)
plt.title('Average Tip Amount by Hour of Day')
plt.xlabel('Hour of Day')
plt.ylabel('Average Tip Amount')
plt.grid(True)
plt.show()
```