NYC Taxi Duration Prediction

December 12, 2023

Authors: Jeff Hansen, Dylan Skinner, Jason Vasquez, Dallin Stewart

1 Introduction

We aim to tackle the challenge of predicting taxi ride durations in New York City based on starting and stopping coordinates. A 'taxi ride duration' refers to how long, in minutes, a taxi ride takes. This research question is relevant for urban commuters and transportation systems, and it incorporates the myriad factors influencing taxi ride times. These factors include traffic patterns, congestion, time of day, and external variables such as local events or weather conditions. The strengths of machine learning methods align well with the intricacies of this problem, and allow us to uncover more nuanced relationships within the data. Beyond the potental convenience for individual riders, the ability to predict taxi ride times carries practical implications for optimizing taxi fleet management, city resource allocation, and enhancing overall traffic flow in New York.

In preparation for our analysis, we conducted initial research about travel time prediction. We quickly learned about the importance of removing outlying data and unnecessary attributes [RR22] (see Data Cleaning). Additionally, we learned which models typically perform best with travel time prediction [BLM18] and concluded that tree based ensambles are typically best for this type of task [HPMP20].

Kaggle published the dataset we are exploring for a coding competition, so over one thousand other groups have investigated this research question. Successful teams performed feature engineering to create fields such as month, day, hour, day of the week, and used models such as Random Forest Regression, Extra Trees Regression, PCA, XGBoost, linear regression, and Light GBM.

The original dataset came fully complete, does not contain missing data, and includes fields such as: taxi driver, the number of passengers, and whether the trip time was recorded in real time. The second dataset contains weather information with timestamps, temperature, precipitation, cloud cover, and wind information at every hour. The NYC taxi cab dataset was published by NYC Taxi and Limousine Commission (TLC) in Big Query on Google Cloud Platform and is well documented and densely populated with over one million data points. The weather dataset is much more sparse and has a much larger time range than needed to match the NYC Taxi data. These datasets provide a good source for addressing our research questions because they extensively cover NYC taxi travel for a significant time period. In short, the data allows us to effectively identify significant features impacting taxi trip duration and develop a robust predictive model for accurate estimations.

While we focus our primary investigation on determining taxi cab trip duration, we are also interested in several other questions. What days of the week and times of day are most busy? Where are the most popular destinations? Furthermore, the multifaceted exploration of temporal, spatial, and environmental influences on travel durations in NYC presents a well-rounded analysis to reveal comprehensive insights into travel behaviors. This poses questions such as how do environmental factors such as rain impact taxi popularity? Our data is well suited for our research question in exploring and predicting taxi trip duration, and has a single, clear answer. A process of machine learning model development will help us go one step further and develop a robust predictive model to estimate and understand trip durations accurately. Our approach, seen below, is comprehensive and unique in several ways. We also use K-means clustering to group the pick-up and drop-off locations together, and use a grid search to find optimal hyperparameters of each model.

2 Feature Engineering

In pursuit of a model with higher predictive power, we included several additional features in our dataset that fall in one of three distinct feature groups: datetime, distance, and weather.

2.1 Data Cleaning

The taxi cab duraction dataset from Kaggle contained several outliers that required removal. For example, some trips lasted 1 second, and others lasted over 980 days. To prevent erroneous data, we removed all rows where trip_duration was in the .005 quantile, or less than 60 seconds. The dataset also contained outliers in the pick up and drop off locations that fell far outside New York City. We fixed these rows by removing any point outside of city limits. For the full plot visualization, see section 8. Data Cleaning in the Appendix.

2.2 Datetime

One of the most important features in our dataset is passenger pickup time, originally represented in a string in the format `YYYY-MM-DD HH:MM:SS`. We created multiple time features from this column including `pickup_month`, one-hot encoded `pickup_day` columns, `pickup_hour`, and `pickup_minute`. We also added other versions of these data points including one-hot encoded `pickup_period`, `pickup_hour_sin`, `pickup_hour_cos`, and `pickup_datetime_norm`.

2.2.1 Pickup Period

The feature `pickup_period` captures the time of day when passengers were picked up in one of four periods: morning (6:00 AM to 12:00 PM), afternoon (12:00 PM to 6:00 PM), evening (6:00 PM to 12:00 AM), and night (12:00 AM to 6:00 AM). These divisions align intuitively with significant periods of the day for taxi services, such as morning rush hours and evening nightlife.

2.2.2 Pickup Period Sine/Cosine

We applied a circular encoding to the hour of the day to account for the cyclical nature of the hours of the day. We created `pickup_hour_sin` and `pickup_hour_cos` features using sine and cosine transformations to avoid discontinuities such as the start and end of a day.

$$\text{hour\_sin} = \sin\left(\frac{2\pi \cdot \text{pickup\_hour}}{24}\right) \quad \text{hour\_cos} = \cos\left(\frac{2\pi \cdot \text{pickup\_hour}}{24}\right). \tag{1}$$

2.2.3 Pickup Datetime Norm

The final feature we created in the datetime feature grouping was `pickup_datetime_norm` to represent the normalized pickup datetime. This feature converts the pickup datetime from nanoseconds to seconds, then scales the value by the maximum to place all the values between 0 and 1.

2.3 Distance

We created two features that estimate distance between pickup and drop off locations: the Manhattan distance and the average distances between local coordinate clusters.

2.3.1 Manhattan Distance

We include the Manhattan Distance feature because of its grid based metric. Many of the streets of New York are laid out in a grid-like fashion, so this metric can better approximate road distances than the Euclidean distance. The Manhattan distance is also more simple and interpretable, since it computes the sum of the absolute values of the differences between the x and y coordinates of the two points. We calculated the Manhattan distance by first converting the pickup and dropoff coordinate points into radians and using the following formula:

$$\text{Manhattan Distance} = R\cdot(|\text{pickup\_latitude} - \text{dropoff\_latitude}| + |\text{pickup\_longitude} - \text{dropoff\_longitude}|) \tag{2}$$

where R is the radius of the Earth in kilometers (6371 km).

2.3.2 KMeans Clustering Average Duration

Along with incorporating distance metrics and weather information, we also added a duration feature that acts as an initial estimate for the model's actual trip duration prediction. To do this, we fit a pickup and dropoff KMeans clustering model with 200 clusters as shown in section `4. Data Visualization`. We then labeled each pickup location and drop off location in the data with its respective cluster label. By grouping the data by these cluster pairs, we computed the average trip_duration between each cluster pair and merged this onto the original dataframe. See section `6. KMeans Clustering` in the Appendix for the full code implementation. This feature is helps by giving the model a baseline for what duration to expect based on location. It also allows us to segment the data in a way that can group outliers, or large distances locations, together and improve accuracy by grouping like locations together.

2.4 Weather

When considering potential features to add to our dataset, accounting for the effect of local weather on taxi ride tme was one of the most obvious additions to include. For example, if it is raining or snowing, we may assume there will be more traffic on the roads, and more people who would normally walk would prefer a taxi. A dataset created by Kaggle user @Aadam contains a miriad of weather data for New York City between 2016 and 2022 on an hourly basis. This dataset includes features such as temperature (in Celcius), precipition (in mm), cloud cover (low, mid, high, and total), wind speed (in km/h), and wind direction. We decided to use temperature, precipitation, and total cloud cover as features in our dataset with a simple join on the pickup datetime rounded to the nearest whole hour.

3 Feature Selection

As seen is section 2, we created several new features in addition to those already in the dataset. We also consider which features are unnecessary or unhelpful.

3.1 L^1 Regularization

To determine the most important features, we utilized L^1 regularization since it functions by setting unneeded feature coefficents to 0 and typically out-performs step-wise feature removal.

[ ]:
```python
# lasso_feature_selection performs feature selection using the LassoLarsIC method
lasso_feature_selection(X, y)
```

4 Data Exploration and Visualization

4.1 Data Exploration

Before visualizing the data, we will first perform basic data exploration to show the effects of adding the features mentioned above.

We built a function `get_X_y()` that performs basic feature engineering and returns two Pandas dataframes: X and y. We also built the function `generate_features()` that performs more advanced feature engineering and also returns two Pandas dataframes: X and y. The table below shows a single example of the data anfe features from the final feature_X dataframe.

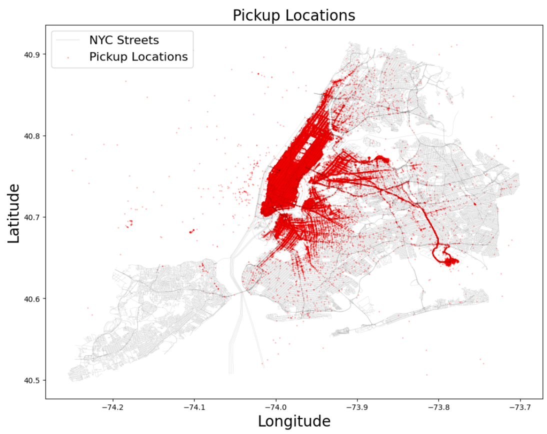
[20]:
```python
X, y = get_X_y(force_clean=True)    # All feature engineering is done in the function get_X_y, found in appendix
feature_X = generate_features(X, y) # generates features adds more features to the data, including weather
feature_X.shape, y.shape
```

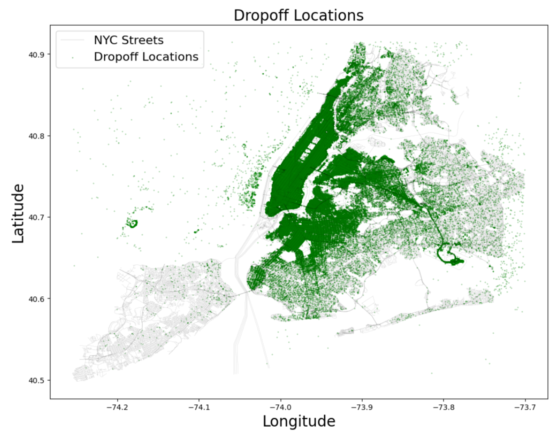
[20]:
```
((1441615, 27), (1441615,))
```

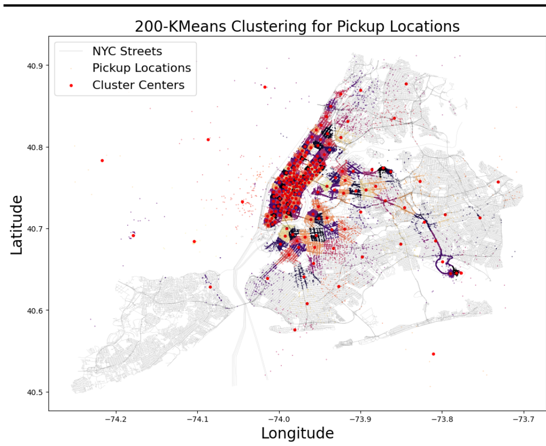
Feature	Value	Feature	Value	Feature	Value
vendor_id	1	pickup_datetime	2016-03-14 17:24:55	passenger_count	1
pickup_longitude	-73.98	pickup_latitude	40.77	dropoff_longitude	-73.96
dropoff_latitude	40.77	pickup_month	3	pickup_day_Monday	1
pickup_day_Saturday	0	pickup_day_Sunday	0	pickup_day_Thursday	0
pickup_day_Tuesday	0	pickup_day_Wednesday	0	pickup_hour	17
pickup_minute	24	pickup_period_morning	0	pickup_period_afternoon	1
pickup_period_evening	0	pickup_hour_sin	-0.97	pickup_hour_cos	-0.26
pickup_datetime_norm	0.41	distance_km	2.21	temperature_2m (°C)	6.40
precipitation (mm)	0.20	cloudcover (%)	100.00	avg_cluster_duration	814.73

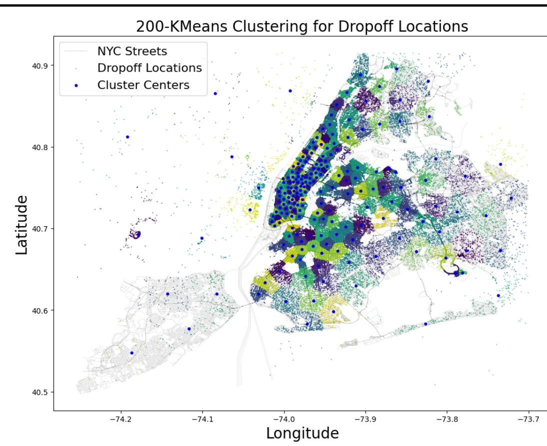
4.2 Data Visualization

In the following figure, the top two graphs visualize the pickup and dropoff locations overlaid over a map of NYC. The bottom two graphs display the pickup and dropoff locations clustered into groups using K-means clustering. These charts reveal that pickup locations are more heavily clustered around downtown (Manhattan), while the dropoff locations are more evenly distributed throughout the city. This distribution indicates that Manhattan, where more people go to work, is the most popular place to hail a taxi. The most popular destination is still in Manhattan, but the destinations are much more distributed across New York. While it is difficult to definitely conclude the cause for this difference, one possible explanation is that people get rides home more frequently. For the full plot implementation, see section 7. Visualization in the Appendix.

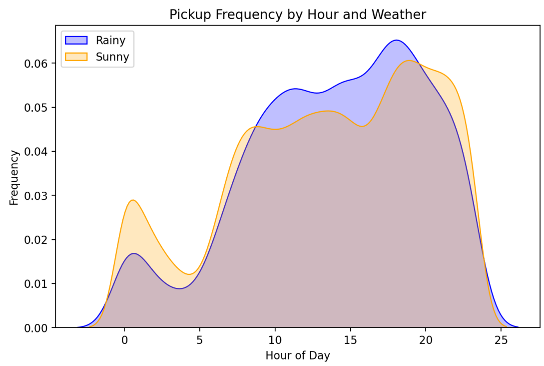


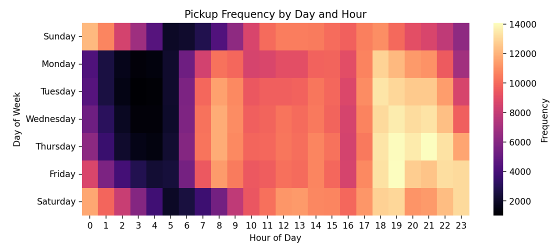






We can also learn about the factors that influence a New Yorker's decision to take a taxi from the data. For example, in the figure below, the graph on the left displays the most popular times of day to hail a taxi, which peeks around 6:00 PM and drops the lowest around 4 am. We also see that poor weather encourages more people to take a taxi during the day when people are more likely to be returning from their daily activities, but less likely to choose to go out at night in the first place. The graph on the right shows the most popular days of the week for taxis, which peeks on Friday and Saturday and during the evenings of the weekdays. In the graph on the right, brighter colors indicate more taxi rides, and darker colors indicate fewer taxi rides. For the full plot implementation, see section 7. Visualization in the Appendix.





5 Modeling and Results

Our primary research question is to predict the duration of a taxi ride in New York City. To achieve this goal, we implemented a variety of machine learning models including Lasso regression, Random Forest regression, XGBoost, and LightGBM. We explain our implementation, training, optimization, and results for each model below.

5.1 Model Selection

After iteratively designing features, we performed hyperparameter grid searches for the each of these models. The table below shows the training time for each model in one columna and references the function call made in the other.

Model	Time	Function Call
Light GBM	1.5 hr	`lightgbm_hyperparameter_search(X, y)`
Lasso Regression	<1 min	`lasso_regression_model(optimal_alpha=1.0806)`
XGBoost	9 hr	`xgboost_hyperparameter_search(X, y)`
Random Forest	6 hr	`random_forest_gridsearch()`

These following table contains the optimal hyperparameter choices and the best RMSE for each model:

Parameter	Lasso Params	LightGBM Params	LightGBM_large Params	RF Params	XGBoost Params
boosting_type	–	gbdt	gbdt	–	gbtree
learning_rate	–	0.01	0.01	–	0.01
max_depth	–	20	50	3	10
n_estimators	–	100	100000	200	100
num_leaves	–	30	500	–	–

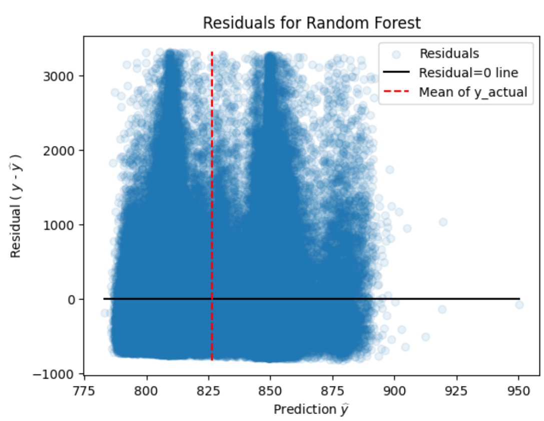
Parameter	Lasso Params	LightGBM Params	LightGBM_large Params	RF Params	XGBoost Params
reg_alpha	1.0806	0.1	0.1	–	0.1
reg_lambda	–	0.5	0.5	–	–
max_features	–	–	–	log2	–
min_samples_leaf	–	–	–	2	–
Best RMSE	606.9680	606.9699	622.4705	605.1684	603.7398

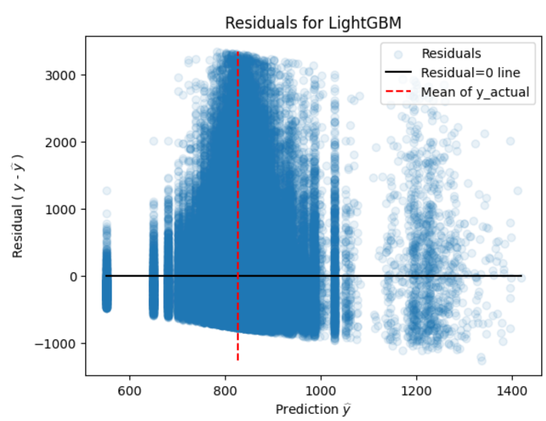
6 Interpretation

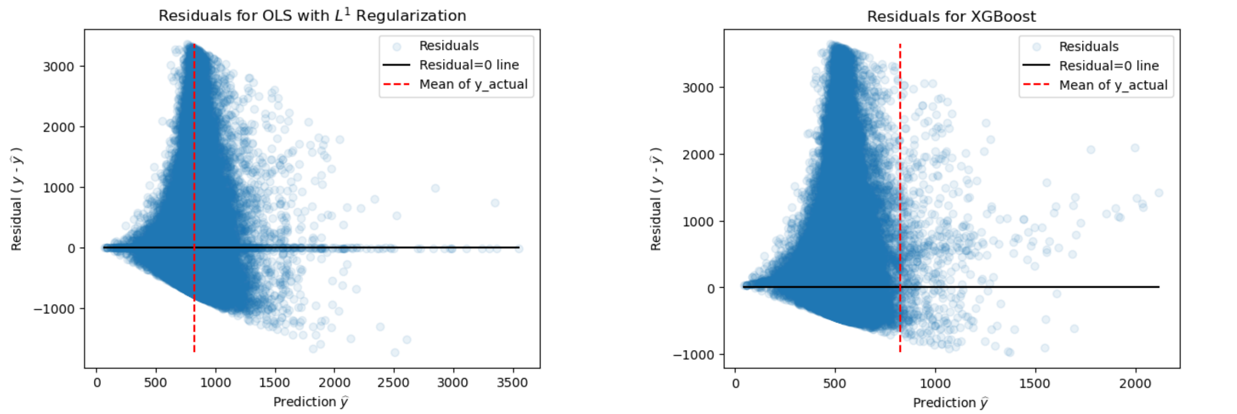
The residual plots below indicate that none of our models performed very well. Instead of accurate prediction, each model clusters their predictions around the mean value for all taxi trip duractions. While this clustering indicates that our models are underfit, we can still explore the unique aspects of each residual graph.

The Random Forest residual plot reveals a multimodal-like distribution with three main hills. This layout implies that the Random Forest model identified several clusterings with different means from the taxi data. Additionally, the LightGBM residuals appear bi-modal, suggesting that the model found an additional clustering. The bin-like prediction values for the LightGBM model is likely a result of LightGBM's use of histogram bining to speed up the training process.

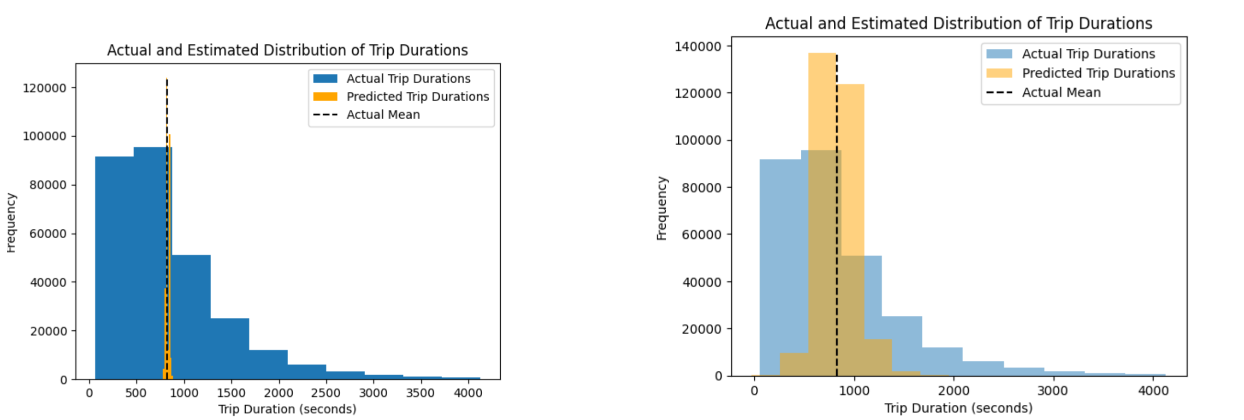
The Lasso model's slightly right-skewed normal distribution likely stems from its underfitting linear model. Despite its limitation, the Lasso model's residual plot aligns remarkably well with the actual trip duration distribution depicted in the histograms below. Furthermore, the XGBoost model appears very similar to the Lasso model, but with more skew to the right. This difference likely arrises because the dataset has many more shorter trips than longer ones, and the model is more likely to predict the mode over the mean.



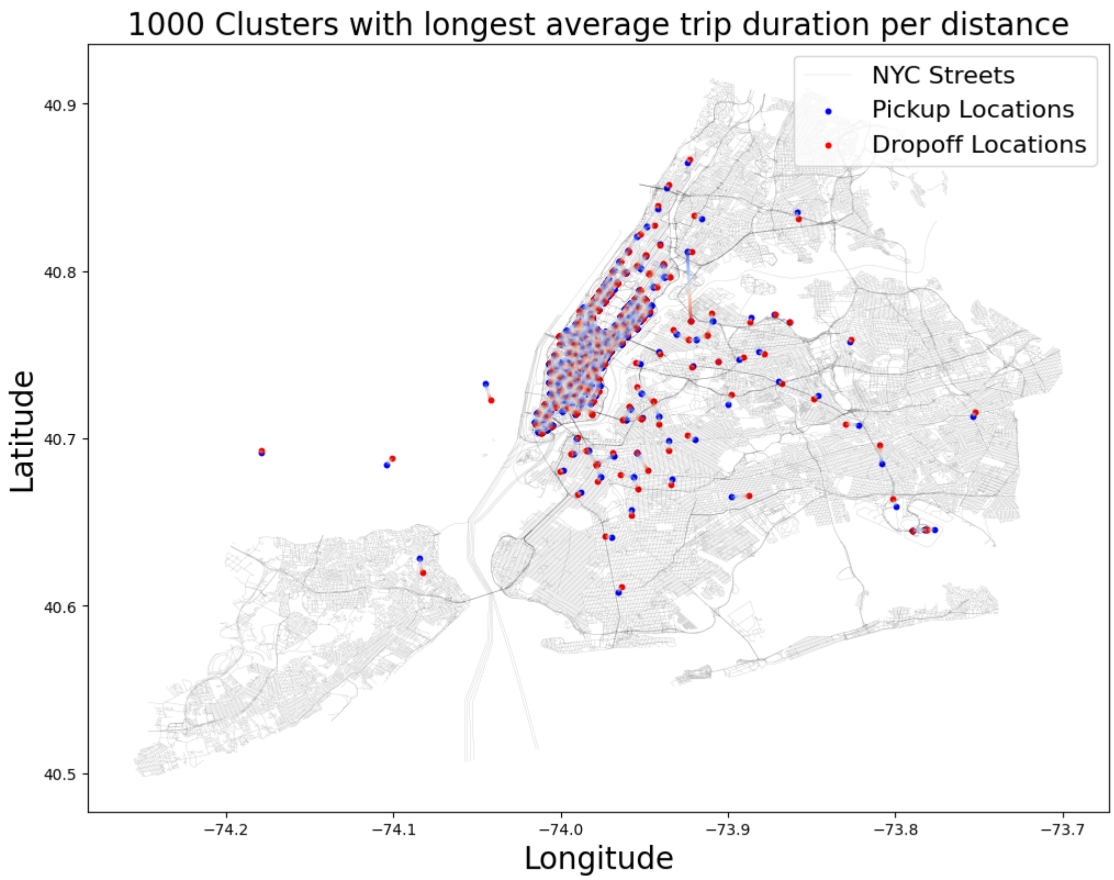




The figures below depict the actual distribution of taxi trip durations along with our model's predicted distributions. The plot on the left depicts the distribution of outputs from a LightGBM model with optimal parameters (100 `estimators`, 30 `leaves`, and a `max_depth` of 50). This model is not very strong, as it predicts the mean of the actual distribution almost exclusively. This poor performance implies that the feature space is not large enough for the model to create a more accurate approximation of the distribution. We generated the plot on the right with a different LightGBM model using much larger parameters (100,000 `estimators`, 500 `leaves` and a `max_depth` of 50). This model took a full hour to train and around 15 minutes to make predictions. As the prediction distribution demonstrates, this model makes predictions farther from the mean, indicating that the model fits the data better as we raise its complexity and compute capabilities. With enough compute resources and time, we could increase the model parameters to higher values and create more features for the data. This upgrade would allow the model better approximate the data instead of simply predicting the mean.



The maps below plots cluster pairs which have the highest travel time to distance traveled ratio on average. Interestingly, some of the cluster paths cross major roadway, implying that there should be more support roads that cross under or over the major highways. Furthermore, a majority of the largest duration trips per distance occur in Manhattan. This demonstrates that the public transit system could be improved to optimize taxi traffic use.



7 Ethical Implications

Our research involves analyzing a large dataset created by tracking some of the life-style patterns of real people living in New York during 2016, raising concerns about privacy and responsible data usage for individual behaviors, locations, and travel patterns. Our dataset and model protect this data by excluding all personally identifiable information to ensure that only aggregate information can be meaningful, and the patterns of individuals remain indiscernible.

The risk of misinterpretation or misuse of our predictive models is very real. Users could misunderstand predictions, leading to inappropriate decision-making. Our predictive model may contain and inadvertently perpetuate biases that we are unaware of, such as inappropriate associations with certain neighborhoods and taxis. Furthermore, users might misunderstand the predictive and uncertain nature of the model and treat its estimates as certainties. For instance, if taxi companies or transportation authorities were to make decisions solely based on the model's predictions without considering broader traffic management strategies, it could inadvertently lead to concentrated traffic, worsening congestion in certain areas. If our model were deployed in conjunction with algorithms influencing taxi availability, the system might inadvertently create self-fulfilling feedback loops, disproportionately affecting certain areas or demographics.

To address these issues, clear communication about the model's limitations, potential biases, and intended use is crucial. Providing educational resources, user-friendly interfaces, adequate documentation, and implementing fairness-aware algorithms can contribute to responsible and ethical deployment. Regular assessments, periodic audits, and interventions are necessary to avoid reinforcing existing biases. We have also considered ethical responsibilities such as the responsible disclosure of findings, ensuring the public benefits from the research, and avoiding any unintentional harm. Active engagement with potential stakeholders and the community can further help address concerns and foster ethical practices.

8 Appendix

8.1 Citations

[RR22] - Roy, B., Rout, D. (2022). Predicting Taxi Travel Time Using Machine Learning Techniques Considering Weekend and Holidays. In: Abraham, A., et al. Proceedings of the 13th International Conference on Soft Computing and Pattern Recognition (SoCPaR 2021). SoCPaR 2021. Lecture Notes in Networks and Systems, vol 417. Springer, Cham. https://doi.org/10.1007/978-3-030-96302-6_24

[HPMP20] - Huang, H., Pouls, M., Meyer, A., Pauly, M. (2020). Travel Time Prediction Using Tree-Based Ensembles. In: Lalla-Ruiz, E., Mes, M., Voß, S. (eds) Computational Logistics. ICCL 2020. Lecture Notes in Computer Science(), vol 12433. Springer, Cham. https://doi.org/10.1007/978-3-030-59747-4_27

[BLM18] - Bai, M., Lin, Y., Ma, M., Wang, P. (2018). Travel-Time Prediction Methods: A Review. In: Qiu, M. (eds) Smart Computing and Communication. SmartCom 2018. Lecture Notes in Computer Science(), vol 11344. Springer, Cham. https://doi.org/10.1007/978-3-030-05755-8_7

8.2 0. Imports

[1]:
```python
# python imports
import geopandas as gpd
import pandas as pd
import matplotlib.pyplot as plt
import pickle
from copy import deepcopy
from sklearn.cluster import KMeans
import numpy as np
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.metrics import mean_squared_error

# Native imports
from py_files.features import generate_features
from py_files.data_manager import get_X_y, get_nyc_gdf
```

8.2.1 1. Lasso Feature Selection

[ ]:
```python
def lasso_feature_selection(X, y):
    """
    Performs feature selection using the LassoLarsIC method

    Parameters
```

```
    - X (dataframe): dataframe of input features
    - y (series): series of target values

    Returns:
    - (dict): dictionary of results (optimal alpha, optimal BIC, lasso␣
↪coefficients, important features)
    """
    lasso_lars_ic = make_pipeline(StandardScaler(with_mean=False),␣
↪LassoLarsIC(criterion="bic", normalize=False)).fit(X, y)

    results = pd.DataFrame(
        {
            "alphas": lasso_lars_ic[-1].alphas_,
            "BIC criterion": lasso_lars_ic[-1].criterion_,
        }
    ).set_index("alphas")

    optimal_alpha = results[results['BIC criterion'] == results['BIC␣
↪criterion'].min()].index

    # Train a Lasso model with the optimal alpha for feature selection
    lasso = linear_model.Lasso(alpha=optimal_alpha)
    lasso.fit(X, y)

    return {'Optimal Alpha': optimal_alpha.values[0], 'Optimal BIC': results.
↪loc[optimal_alpha].values[0].tolist()[0],
            'Lasso Coeffs': lasso.coef_.round(4), 'Important Features': X.
↪columns[lasso.coef_ != 0].values}
```

1.1 Lasso Regression Model

```
[ ]: def lasso_regression_model(optimal_alpha):
    """
    This function takes in the optimal alpha from the Lasso Lars IC Feature␣
↪Selection and trains a Lasso Regression
    model on the important features.

    Parametes:
    - optimal_alpha (float): The optimal alpha from the Lasso Lars IC Feature␣
↪Selection

    Returns:
    - Dictionary with the RMSE of the model
    """
    # Important features selected from Lasso Lars IC Feature Selection
    important_features = ['pickup_minute', 'distance_km', 'temperature_2m␣
↪(°C)', 'cloudcover (%)', 'avg_cluster_duration']
```

```
    # Create dataframe of important features
    X2 = X[important_features]

    # Get test train split
    X_train, X_test, y_train, y_test = train_test_split(X1, y, test_size=0.2, random_state=42)

    # Create and fit the model.
    model = linear_model.Lasso(alpha=optimal_alpha)
    model.fit(X_train, y_train)

    # Predict on test data and compute RMSE
    y_pred = model.predict(X_test)
    return {'RMSE': mean_squared_error(y_test, y_pred, squared=False)}
```

8.2.2 2. LightGBM Hyperparameter Selection

[ ]:
```
def lightgbm_hyperparameter_search(X, y):
    """
    Performs hyperparameter search for LightGBM model

    Parameters:
    - X (dataframe): dataframe of input features
    - y (dataframe): dataframe of target variables

    Returns:
    - Dictionary of best parameters and best RMSE
    """
    # Train test split
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

    # Create param grid
    param_grid = {
        'boosting_type': ['gbdt', 'dart'],
        'num_leaves': [30, 40],
        'learning_rate': [0.01, 0.05],
        'n_estimators': [100, 200],
        'max_depth': [10, 20],
        'reg_alpha': [0.1, 0.5],
        'reg_lambda': [0.1, 0.5],
    }

    # LightGBM
    lgb_train = lgb.LGBMRegressor()

    # Grid search
```

```
    grid_search = GridSearchCV(estimator=lgb_train, param_grid=param_grid,␣
↪cv=3, scoring='neg_root_mean_squared_error', verbose=1)
    grid_search.fit(X_train, y_train)

    # Validate
    y_pred = grid_search.predict(X_test)

    return {'Best parameters from grid search': grid_search.best_params_, 'Best␣
↪RMSE': mean_squared_error(y_test, y_pred, squared=False)}
```

8.2.3 3. XGBoost Hyperparameter Selection

[4]:
```
def xgboost_hyperparameter_search(X, y):
    """
    Performs a grid search on the XGBoost model

    Parameters:
    - X (DataFrame): The input features
    - y (Series): The target variable

    Returns:
    - None
    """
    # Train test split
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

    param_grid = {
        'booster': ['gbtree', 'dart'],
        'n_estimators': [100, 200],
        'learning_rate': [0.01, 0.05],
        'max_depth': [10, 20],
        'alpha': [0.1, 0.5],
    }

    # XGBoost
    xgb_model = xgb.XGBRegressor()

    # Grid search
    grid_search = GridSearchCV(estimator=xgb_model, param_grid=param_grid,␣
↪cv=3, scoring='neg_root_mean_squared_error', verbose=1)
    grid_search.fit(X_train, y_train)

    # Best params
    print('Best parameters from grid search: ', grid_search.best_params_)
```

4. XGBoost Model

[8]:
```python
def xgboost_model(X, y):
    """
    XGBoost model with optimal hyperparameters

    Parameters:
    - None

    Returns:
    - Dictionary of RMSE results
    """
    # Train test split
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

    # XGBoost
    xgb_model = xgb.XGBRegressor(booster='gbtree', n_estimators=100,␣
 ↪learning_rate=0.01, max_depth=10, alpha=0.1)

    # Fit
    xgb_model.fit(X_train, y_train)

    # Validate
    y_pred = xgb_model.predict(X_test)

    return {'RMSE': mean_squared_error(y_test, y_pred, squared=False)}
```

8.2.4 5. Random Forest Hyperparameter Selection

[ ]:
```python
def random_forest_gridsearch():
    """
    Performs a hyperparameter gridsearch with cross validation
    to find the optimal parameters for a RandomForestRegressor

    Parameters:
    - None

    Returns:
    - None
    """

    # get the X and y, and add the features
    X, y = get_X_y(force_clean=True)
    feature_X = generate_features(X, y)

    # drop the pickup datetime feature since sklearn RandomForest does
    # not accept datetime columns
    feature_X = feature_X.drop(columns=['pickup_datetime'])
```

```
    # get the X and y for training
    X_train = feature_X.copy()
    y_train = y.copy()
    X_train = X_train.reset_index(drop=True)
    y_train = y_train.reset_index(drop=True)

    # to speed up the grid search, we will use the first four instances
    # of each cluster-to-cluster pair of data points
    df = X_train.copy()
    df = df.sort_values(by='avg_cluster_duration')

    dfs = []

    sample_per_class = 4
    for _ in range(sample_per_class):
        firsts = df['avg_cluster_duration'] != df['avg_cluster_duration'].
↪shift(1)
        dfs.append(df.loc[firsts].copy())
        df = df.loc[~firsts].copy()

    # combine all of the reprsentative samples
    final_df = pd.concat(dfs, axis=0).sort_values('avg_cluster_duration')

    # shuffle the data so that it is no longer sorted by avg_cluster_duration
    X_train = final_df.copy().sample(frac=1)
    y_train = y_train.loc[X_train.index]

    X_train = X_train.values.astype(np.float32)
    y_train = y_train.values.astype(np.float32)

    # perform the RandomForest gridsearch to find the best
    # hyperparameters
    param_grid = {
        'n_estimators': [100, 200, 400, 800, 1000],
        'max_depth': [None, 3, 5, 10, 20],
        'max_features': ['sqrt', 'log2'],
        'min_samples_leaf': [1, 2, 3, 5],
    }
    rf = RandomForestRegressor(warm_start=False)
    grid_search = GridSearchCV(estimator=rf, param_grid=param_grid, verbose=3,␣
↪n_jobs=-2, cv=4).fit(X_train, y_train)
    best_params = grid_search.best_params_

    # save the grid_search_model for future loading
    with open("models/rf_grid_search.pkl", "wb") as f:
        pickle.dump(grid_search, f)
```

```
    # print the best parameters
    print("Best parameters RandomForest:", best_params)

    # train a model and compute the RMSE on the test set
    X_train, X_test, y_train, y_test = train_test_split(feature_X, y,␣
↪test_size=0.2, random_state=42)

    rf = RandomForestRegressor(**best_params)
    rf.fit(X_train, y_train)
    y_pred = rf.predict(X_test)
    rf_rmse = mean_squared_error(y_test, y_pred, squared=False)

    print("RandomForest RMSE:", rf_rmse)
```

8.2.5 6. KMeans Clustering

```
[ ]: def kmeans_pickup_dropoff_model(df, n_clusters=200):
    """
    Fits KMeans models for pickup and dropoff locations, labels each location␣
↪by its cluster,
    computes the average duration between each cluster, and merges it onto the␣
↪original dataframe.

    Parameters:
    - df (DataFrame): The input DataFrame containing pickup and dropoff␣
↪locations.
    - n_clusters (int): The number of clusters for KMeans.

    Returns:
    - DataFrame: The modified DataFrame with cluster labels and average cluster␣
↪duration.
    """
    # fit the kmeans models and label each pickup and dropoff location by its␣
↪cluster
    kmeans_pickup = (KMeans(n_clusters=n_clusters)
        .fit(df.loc[:, ['pickup_longitude', 'pickup_latitude']].values))
    kmeans_dropoff = (KMeans(n_clusters=n_clusters)
        .fit(df.loc[:, ['dropoff_longitude', 'dropoff_latitude']].values))
    df['pickup_cluster'] = kmeans_pickup.predict(df[['pickup_longitude',␣
↪'pickup_latitude']].values)
    df['dropoff_cluster'] = kmeans_dropoff.predict(df[['dropoff_longitude',␣
↪'dropoff_latitude']].values)

    # compute the average duration between each cluster and merge this onto the␣
↪original dataframe
```

```
    group_durations = (df
        .groupby(['pickup_cluster', 'dropoff_cluster'])['trip_duration']
        .mean()
        .reset_index()
        .rename(columns={'trip_duration': 'avg_cluster_duration'}))
    df = pd.merge(
        left=df, right=group_durations, how='left',
        left_on=['pickup_cluster', 'dropoff_cluster'],␣
↪right_on=['pickup_cluster', 'dropoff_cluster'])

    # fill the missing values with the mean of the average duration from␣
↪cluster to cluster
    df['avg_cluster_duration'] = df['avg_cluster_duration'].
↪fillna(df['avg_cluster_duration'].mean())
    df.drop(columns=['pickup_200_cluster', 'dropoff_200_cluster',␣
↪'trip_duration'], inplace=True)
    return df
```

6.1 KMeans Clustering Plot

[ ]:
```
# constants/parameters for this code cell
SHOW_PLOTS = True
LOAD_SAVED_KMEANS_MODELS = True

# load in the cleaned training data and the NYC geopandas dataframe
# with all of the NYC streets
X, y = get_X_y(force_clean=True)
nyc_gdf = get_nyc_gdf()


#########################
# PLOT PICKUP LOCATIONS #
#########################
def plot_pickup_locations(X):
    """
    Plots the NYC streets and pickup locations as a scatter plot on top of the␣
↪streets.

    Parameters:
    - X (DataFrame): The DataFrame containing pickup locations.

    Returns:
    - None
    """
    # plot the nyc streets
    plt.gcf()
```

```
    nyc_gdf.plot(linewidth=0.1, edgecolor='black', figsize=(12, 12), alpha=0.5,␣
↪label="NYC Streets")

    # plot the pickup locations as a scatter plot on top of the nyc streets
    plt.scatter(X['pickup_longitude'], X['pickup_latitude'], c='red', alpha=0.
↪75, s=0.1, label="Pickup Locations")
    leg = plt.legend(loc='upper left')
    for lh in leg.legend_handles:
        lh.set_alpha(1)
    plt.title("Pickup Locations")
    plt.xlabel("Longitude")
    plt.ylabel("Latitude")

    # save the plot
    plt.savefig("images/pickup_locations_save.png")
    plt.show() if SHOW_PLOTS else plt.clf()


##########################
# PLOT DROPOFF LOCATIONS #
##########################
def plot_dropoff_locations(X):
    """
    Plots the NYC streets and dropoff locations as a scatter plot on top of the␣
↪streets.

    Parameters:
    - X (DataFrame): The DataFrame containing dropoff locations.

    Returns:
    - None
    """
    # plot the nyc streets
    plt.gcf()
    nyc_gdf.plot(linewidth=0.1, edgecolor='black', figsize=(12, 12), alpha=0.5,␣
↪label="NYC Streets")

    # plot the dropoff locations as a scatter plot on top of the nyc streets
    plt.scatter(X['dropoff_longitude'], X['dropoff_latitude'], c='green',␣
↪alpha=0.75, s=0.1, label="Dropoff Locations")
    leg = plt.legend(loc='upper left')
    for lh in leg.legend_handles:
        lh.set_alpha(1)
    plt.title("Dropoff Locations")
    plt.xlabel("Longitude")
    plt.ylabel("Latitude")
```

```
    # save the plot
    plt.savefig("images/dropoff_locations_save.png")
    plt.show() if SHOW_PLOTS else plt.clf()


#####################
# KMEANS CLUSTERING #
#####################
def kmeans_pickup_dropoff_predict(df, n_clusters=200):
    """
    Applies KMeans clustering to predict pickup and dropoff locations, or loads␣
 ↪pre-trained models if available.

    Parameters:
    - df (DataFrame): The DataFrame containing pickup and dropoff locations.
    - n_clusters (int): The number of clusters for KMeans. Default is 200.

    Returns:
    - df (DataFrame): The DataFrame with predicted clusters for pickup and␣
 ↪dropoff locations.
    - pickup_200_centers (array): The cluster centers for pickup locations.
    - dropoff_200_centers (array): The cluster centers for dropoff locations.
    """
    df = deepcopy(X)

    # load kmeans_pickup and kmeans_dropoff from the models folder using pickle
    if LOAD_SAVED_KMEANS_MODELS:
        with open("models/kmeans_200_pickup.pkl", "rb") as file:
            kmeans_200_pickup = pickle.load(file)
        with open("models/kmeans_200_dropoff.pkl", "rb") as file:
            kmeans_200_dropoff = pickle.load(file)


    # fit kmeans_pickup and kmeans_dropoff with 200 clusters
    else:
        n_clusters = 200
        kmeans_pickup = (KMeans(n_clusters=n_clusters)
            .fit(df.loc[:, ['pickup_longitude', 'pickup_latitude']].values))
        kmeans_dropoff = (KMeans(n_clusters=n_clusters)
            .fit(df.loc[:, ['dropoff_longitude', 'dropoff_latitude']].values))

        # save the models to pickle files for loading later
        with open("models/kmeans_200_pickup.pkl", "wb") as file:
            pickle.dump(kmeans_pickup, file)
        with open("models/kmeans_200_dropoff.pkl", "wb") as file:
            pickle.dump(kmeans_dropoff, file)
```

```
    # predict the clusters for each pickup and dropoff location
    df['pickup_200_cluster'] = kmeans_200_pickup.
 ↪predict(df[['pickup_longitude', 'pickup_latitude']].values)
    df['dropoff_200_cluster'] = kmeans_200_dropoff.
 ↪predict(df[['dropoff_longitude', 'dropoff_latitude']].values)

    # get the centers
    pickup_200_centers = kmeans_200_pickup.cluster_centers_
    dropoff_200_centers = kmeans_200_dropoff.cluster_centers_
    return df, pickup_200_centers, dropoff_200_centers


#######################################
# PLOT PICKUP LOCATIONS WITH CLUSTERS #
#######################################
def plot_cluster_pickup(df, pickup_200_centers):
    """
    Plots KMeans clustering for pickup locations.

    Parameters:
    - df (DataFrame): The DataFrame containing pickup locations and their␣
 ↪associated clusters.
    - pickup_200_centers (array): The cluster centers for pickup locations.

    Returns:
    - None
    """
    # plot the nyc streets
    plt.gcf()
    nyc_gdf.plot(linewidth=0.1, edgecolor='black', figsize=(12, 12), alpha=0.5,␣
 ↪label="NYC Streets")

    # plot the cluster locations and the pickup locations color-coded
    # to their associated cluster
    plt.scatter(df['pickup_longitude'], df['pickup_latitude'],␣
 ↪c=df['pickup_200_cluster'], cmap='magma', alpha=1.0, s=0.1, label="Pickup␣
 ↪Locations")
    plt.scatter(pickup_200_centers[:, 0], pickup_200_centers[:, 1], c='red',␣
 ↪alpha=1, s=10, label="Cluster Centers")
    leg = plt.legend(loc='upper left')
    for lh in leg.legend_handles:
        lh.set_alpha(1)
    plt.title("200-KMeans Clustering for Pickup Locations")
    plt.xlabel("Longitude")
    plt.ylabel("Latitude")
```

```python
    # save the plot
    plt.savefig("images/kmeans_200_pickup_save.png")
    plt.show() if SHOW_PLOTS else plt.clf()


########################################
# PLOT DROPOFF LOCATIONS WITH CLUSTERS #
########################################
def plot_cluster_dropoff(df, dropoff_200_centers):
    """
    Plots KMeans clustering for dropoff locations.

    Parameters:
    - df (DataFrame): The DataFrame containing dropoff locations and their␣
 ↪associated clusters.
    - dropoff_200_centers (array): The cluster centers for dropoff locations.

    Returns:
    - None
    """
    # plot the nyc streets
    plt.gcf()
    nyc_gdf.plot(linewidth=0.1, edgecolor='black', figsize=(12, 12), alpha=0.5,␣
 ↪label="NYC Streets")

    # plot the cluster locations and the pickup locations color-coded
    # to their associated cluster
    plt.scatter(df['dropoff_longitude'], df['dropoff_latitude'],␣
 ↪c=df['dropoff_200_cluster'], cmap='viridis', alpha=1.0, s=0.1,␣
 ↪label="Dropoff Locations")
    plt.scatter(dropoff_200_centers[:, 0], dropoff_200_centers[:, 1], c='blue',␣
 ↪alpha=1, s=10, label="Cluster Centers")
    leg = plt.legend(loc='upper left')
    for lh in leg.legend_handles:
        lh.set_alpha(1)
    plt.title("200-KMeans Clustering for Dropoff Locations")
    plt.xlabel("Longitude")
    plt.ylabel("Latitude")

    # save the plot
    plt.savefig("images/kmeans_200_dropoff_save.png")
    plt.show() if SHOW_PLOTS else plt.clf()
```

8.2.6 7. Visualizations

```
[ ]: def plot_actual_vs_predicted_distributions(show=False):
        """
        Plots histograms of actual and predicted trip durations on the test set and␣
     ↪compares their distributions.

        Parameters:
        - show (bool): If True, displays the plot; if False, saves the plot as␣
     ↪'images/actual_vs_predicted.png'.

        Returns:
        - None
        """

        # load in the actual and predicted trip durations
        X, y = get_X_y()
        feature_X = generate_features(X, y)
        feature_X = feature_X.drop(columns=['pickup_datetime'])
        X_train, X_test, y_train, y_test = train_test_split(feature_X, y,␣
     ↪test_size=0.2, random_state=42)

        # plot a histogram of the actual trip duration distribution in the test set
        # and compute the mean
        counts, bins, patches = plt.hist(y_test, label='Actual Trip Durations')
        actual_mean = np.mean(y_test)

        # plot a histogram of the predicted trip duration distribution on the test␣
     ↪set
        counts_pred, bins_pred, pathces_pred = plt.hist(y_pred, label='Predicted␣
     ↪Trip Durations', color='orange')

        # plot a verticle line at the actual mean of the distribution
        top = max(np.max(counts), np.max(counts_pred))
        plt.vlines([actual_mean], 0, top, color='black', linestyles='dashed',␣
     ↪label='Actual Mean')

        # set other plot parameters and show the plot
        plt.legend()
        plt.title("Actual and Estimated Distribution of Trip Durations")
        plt.xlabel("Trip Duration (seconds)")
        plt.ylabel("Frequency")
        plt.savefig('images/actual_vs_predicted.png')

        if show:
            plt.show()
        else:
```

```
        plt.clf()
```

8.2.7 8. Data Cleaning

[ ]:
```
"""
this py file contains all of the data loading, cleaning, and saving logic.
The methods will automatically pull in a cached version of the dataframe
unless force_clean=True. If force_clean=True, then the dataframe will be
cleaned and saved to the data folder.
"""

import pandas as pd
import os
import numpy as np
import json
import geopandas as gpd
from shapely.wkt import loads

from config import (
    data_path, cols_to_drop, SET_VENDOR_ID_TO_01,
    PICKUP_TIME_TO_NORMALIZED_FLOAT
)
from py_files.helper_funcs import p


def clean_data(df, df_name, verbose=False):
    """
    Loads in the train.csv and test.csv and cleans them according
    to the constants in config.py. Saves the cleaned dataframes as
    train_clean.csv and test_clean.csv

    Parameters:
    - df (pandas dataframe): The dataframe to be cleaned
    - df_name (str): The name of the dataframe, either 'train' or 'test'
    - verbose (bool): If True, prints out the progress of the cleaning

    Returns:
    - df_clean (pandas dataframe): The cleaned dataframe
    """

    # only keep the relevant columns based on the config
    p("dropping columns") if verbose else None
    curr_cols_to_drop = [c for c in df.columns if c in cols_to_drop]
    df_clean = df.drop(columns=curr_cols_to_drop)
    p() if verbose else None

    # setting vendor_id to a 0 or 1 instead of 1 and 2
```

```
    if SET_VENDOR_ID_TO_01:
        p("setting vendor_id to 0 or 1") if verbose else None
        df_clean['vendor_id'] = df_clean['vendor_id'] - 1
        p() if verbose else None

    # Drop rows with trip duration < 60 seconds
    p("dropping rows with trip duration < 60 seconds") if verbose else None
    df_clean = df_clean[df_clean['trip_duration'] >= 60]

    # Drop rows with outlier locations
    p("dropping rows with outlier locations") if verbose else None
    json_file_path = './misc/lat_long_bounds.json'
    # Read in coordinates
    with open(json_file_path, 'r') as json_file:
        # Load the JSON data from the file
        coords = json.load(json_file)
    df_clean = df_clean[(df_clean['pickup_latitude'] >= coords['lat']['min']) &␣
↪(
        df_clean['pickup_latitude'] <= coords['lat']['max'])]
    df_clean = df_clean[(df_clean['pickup_longitude'] >= coords['lon']['min'])␣
↪& (
        df_clean['pickup_longitude'] <= coords['lon']['max'])]
    df_clean = df_clean[(df_clean['dropoff_latitude'] >= coords['lat']['min'])␣
↪& (
        df_clean['dropoff_latitude'] <= coords['lat']['max'])]
    df_clean = df_clean[(df_clean['dropoff_longitude'] >= coords['lon']['min'])␣
↪& (
        df_clean['dropoff_longitude'] <= coords['lon']['max'])]

    # Keep only <99.5% of trip duration
    p("dropping rows with trip duration > 99.5%") if verbose else None
    df_clean = df_clean[df_clean['trip_duration'] <=
                        df_clean['trip_duration'].quantile(0.995)]

    # Split apart pickup_datetime
    df_clean['pickup_datetime'] = pd.to_datetime(df['pickup_datetime'])
    df_clean['pickup_month'] = df_clean['pickup_datetime'].dt.month
    df_clean['pickup_day'] = df_clean['pickup_datetime'].dt.day_name()
    df_clean = pd.get_dummies(
        df_clean, columns=['pickup_day'], drop_first=True)
    df_clean['pickup_hour'] = df_clean['pickup_datetime'].dt.hour
    df_clean['pickup_minute'] = df_clean['pickup_datetime'].dt.minute

    # Create a pickup period.
    df_clean['pickup_period'] = pd.cut(df_clean['pickup_hour'], bins=[
                                       -1, 6, 12, 18, 24], labels=['night',␣
↪'morning', 'afternoon', 'evening'])
```

```
    # Get dummies for the pickup period.
    df_clean = pd.get_dummies(
        df_clean, columns=['pickup_period'], drop_first=True)

    # Add cyclic data.
    df_clean['pickup_hour_sin'] = np.sin(
        2 * np.pi * df_clean['pickup_hour'] / 24)
    df_clean['pickup_hour_cos'] = np.cos(
        2 * np.pi * df_clean['pickup_hour'] / 24)

    # convert pickup and dropoff times to floats from 0 to 1
    if PICKUP_TIME_TO_NORMALIZED_FLOAT:
        df_clean['pickup_datetime_norm'] = pd.to_datetime(
            df_clean['pickup_datetime']).view('int64') // 10**9
        df_clean['pickup_datetime_norm'] = (df_clean['pickup_datetime_norm'] -␣
↪df_clean['pickup_datetime_norm'].min()) / (
            df_clean['pickup_datetime_norm'].max() -␣
↪df_clean['pickup_datetime_norm'].min())

    # Drop the id column
    df_clean = df_clean.drop(columns=['id'])

    # save the cleaned dataframe
    p("saving cleaned dataframe") if verbose else None
    df_clean.to_csv(f"{data_path}/{df_name}_clean.csv", index=False)
    p() if verbose else None

    return df_clean


def get_train_data(force_clean=False):
    """
    Either creates the cleaned train dataframe from the train.csv
    or it loads it from the data folder

    Parameters:
    - force_clean (bool): If True, forces the data to be cleaned

    Returns:
    - train (pandas dataframe): A dataframe of the train data
    """
    if not os.path.exists(f"{data_path}/train_clean.csv") or force_clean:
        train = pd.read_csv(f"{data_path}/train.csv")
        return clean_data(train, 'train')
    else:
        return pd.read_csv(f"{data_path}/train_clean.csv")
```

```python
def get_X_y(return_np=False, force_clean=False):
    """
    Returns the X and y dataframes from a dataframe

    Parameters:
    - return_np (bool): If True, returns numpy arrays instead of
    - force_clean (bool): If True, forces the data to be cleaned

    Returns:
    - X (pandas dataframe): A dataframe of the input data
    - y (pandas dataframe): A dataframe of the label data
    """
    df = get_train_data(force_clean=force_clean)
    X = df.drop(columns=['trip_duration'])
    y = df['trip_duration']

    if return_np:
        X, y = X.values, y.values

    return X, y


def get_test_data():
    """
    Either creates the cleaned test dataframe from the test.csv
    or it loads it from the data folder

    Parameters:
    - None

    Returns:
    - test (pandas dataframe): A dataframe of the test data
    """
    if not os.path.exists(f"{data_path}/test_clean.csv"):
        test = pd.read_csv(f"{data_path}/test.csv")
        return clean_data(test, 'test')
    else:
        return pd.read_csv(f"{data_path}/test_clean.csv")


def get_clean_weather():
    """
    Loads in the NYC_Weather_2016_2022.csv and cleans it according
    to the constants in config.py. Saves the cleaned dataframe as
    weather_clean.csv
```

```
    Parameters:
    - None

    Returns:
    - weather (pandas dataframe): A dataframe of the weather
    """
    if not os.path.exists(f"{data_path}/weather_clean1.csv"):
        weather = pd.read_csv(f"{data_path}/NYC_Weather_2016_2022.csv")
        weather = weather.dropna()
        weather['time'] = pd.to_datetime(weather['time'])
        weather = weather[weather['time'] <= '2016-07-01']
        weather = weather.drop(columns=['rain (mm)',
                                        'cloudcover_low (%)',
                                        'cloudcover_mid (%)',
                                        'cloudcover_high (%)',
                                        'windspeed_10m (km/h)',
                                        'winddirection_10m (°)'])
        weather.to_csv(f"{data_path}/weather_clean1.csv", index=False)
        return weather
    else:
        return pd.read_csv(f"{data_path}/weather_clean1.csv")


def get_google_distance():
    """
    Loads in the train_distance_matrix.csv and cleans it according
    to the constants in config.py. Saves the cleaned dataframe as
    google_distance_clean.csv

    Parameters:
    - None

    Returns:
    - google_distance (pandas dataframe): A dataframe of the google
    """
    if not os.path.exists(f"{data_path}/google_distance_clean.csv"):
        google_distance = pd.read_csv(f"{data_path}/train_distance_matrix.csv")

        columns_to_keep = ['id', 'gc_distance', 'google_distance']
        google_distance = google_distance[columns_to_keep]

        google_distance.to_csv(
            f"{data_path}/google_distance_clean.csv", index=False)
        return google_distance
    else:
        return pd.read_csv(f"{data_path}/google_distance_clean.csv")
```

```python
def get_nyc_gdf():
    """
    Loads in the NYC street centerline data and returns it as a
    geopandas dataframe

    Parameters:
    - None

    Returns:
    - gdf (geopandas dataframe): A geopandas dataframe of the NYC
    """
    nyc_df = pd.read_csv(f"{data_path}/Centerline.csv")
    nyc_df = nyc_df.loc[:, ['the_geom']]

    # Convert the "the_geom" column to Shapely geometries
    nyc_df['the_geom_geopandas'] = nyc_df['the_geom'].apply(loads)

    # Create a GeoDataFrame
    gdf = gpd.GeoDataFrame(nyc_df, geometry='the_geom_geopandas')

    return gdf
```

8.2.8 9. Feature Engineering

[ ]:
```python
"""
this py file holds all of the logic behind the feature engineering.
The generate_features function takes in an X and a y, and it adds
feature columns based on the config.features_toggle
"""

from config import features_toggle
from py_files.data_manager import get_clean_weather, get_google_distance
import numpy as np
import pandas as pd
import pickle


def distance(df):
    """
    Calculate the Manhattan distance in kilometers between pickup and dropoff␣
 ↪locations
    and add it as a new column 'distance_km' to the DataFrame.

    The Manhattan distance, also known as the L1 distance or taxicab distance,␣
 ↪between two points
```

```
    on the Earth's surface is calculated by finding the absolute differences␣
↪between their respective
    longitudes and latitudes and summing them up. This function computes the␣
↪Manhattan distance
    in kilometers between the pickup and dropoff locations in a DataFrame,␣
↪assuming a constant
    Earth radius of 6371 kilometers.

    Parameters:
    df (pandas.DataFrame): A DataFrame containing pickup and dropoff␣
↪coordinates with columns
                           'pickup_longitude', 'pickup_latitude',␣
↪'dropoff_longitude', and 'dropoff_latitude'.

    Returns:
    pandas.DataFrame: A DataFrame with an additional 'distance_km' column␣
↪representing the Manhattan
                      distance in kilometers between pickup and dropoff␣
↪locations.
    """
    # Radius of the Earth in kilometers
    earth_radius_km = 6371.0

    # Get the pickup and dropoff coordinates
    lon1 = df['pickup_longitude']
    lat1 = df['pickup_latitude']
    lon2 = df['dropoff_longitude']
    lat2 = df['dropoff_latitude']

    # Convert latitude and longitude from degrees to radians
    lon1, lat1, lon2, lat2 = map(np.radians, [lon1, lat1, lon2, lat2])


    # Calculate the differences in latitude and longitude
    delta_lat = abs(lat1 - lat2)
    delta_lon = abs(lon1 - lon2)

    # Calculate the Manhattan distance in kilometers
    df['distance_km'] = earth_radius_km * (delta_lat + delta_lon)

    return df


def add_weather_feature(df):
    """
```

```
    Add weather-related features to a DataFrame by merging it with a weather␣
↪dataset.

    This function merges the input DataFrame with a weather dataset based on␣
↪the rounded pickup time
    and adds weather-related features to the DataFrame. It rounds the␣
↪'pickup_datetime' column to the
    nearest hour to match the weather data's time resolution.

    Parameters:
    df (pandas.DataFrame): The input DataFrame containing pickup-related data.

    Returns:
    pandas.DataFrame: A DataFrame with added weather-related features merged␣
↪from the weather dataset.
    """
    # Get weather data
    weather = get_clean_weather()

    # Round the pickup time to the nearest hour (to merge with weather)
    df['rounded_date'] = pd.to_datetime(df['pickup_datetime']).dt.round('H')
    weather['time'] = pd.to_datetime(weather['time'])

    # Merge with weather
    df = df.merge(weather, left_on='rounded_date', right_on='time')

    # Drop unnecessary columns and return the dataframe
    df = df.drop(columns=['rounded_date', 'time'])
    return df


def add_google_distance(df):
    """
    Add Google Maps distance and duration features to a DataFrame by merging it␣
↪with a Google Maps dataset.

    Parameters
    - df (DataFrame): The input DataFrame containing pickup and dropoff␣
↪coordinates.

    Returns
    - df (DataFrame): The DataFrame with added Google Maps distance and␣
↪duration features.
    """

    # Get the google distance
```

```
    google_distance = get_google_distance()

    # Merge with the dataframe (verify 1:1)
    df = df.merge(google_distance, on='id', validate='1:1')

    return df


def add_avg_cluster_duration(df, y):
    """
    Adds a column 'avg_cluster_duration' to the DataFrame representing the␣
 ↪average duration
    from cluster to cluster based on pickup and dropoff locations.

    Parameters:
    - df (DataFrame): The input DataFrame containing features.
    - y (array-like): The array of target values (trip durations).

    Returns:
    - df (DataFrame): The DataFrame with added 'avg_cluster_duration' column.
    """
    df = df.copy()
    df['trip_duration'] = y

    # load kmeans_pickup and kmeans_dropoff from the models folder using pickle
    with open("models/kmeans_200_pickup.pkl", "rb") as file:
        kmeans_200_pickup = pickle.load(file)
    with open("models/kmeans_200_dropoff.pkl", "rb") as file:
        kmeans_200_dropoff = pickle.load(file)

    # predict the clusters for the pickup and dropoff locations using the␣
 ↪kmeans_pickup and kmeans_dropoff
    df['pickup_200_cluster'] = kmeans_200_pickup.
 ↪predict(df[['pickup_longitude', 'pickup_latitude']].values)
    df['dropoff_200_cluster'] = kmeans_200_dropoff.
 ↪predict(df[['dropoff_longitude', 'dropoff_latitude']].values)

    # get the centers
    pickup_200_centers = kmeans_200_pickup.cluster_centers_
    dropoff_200_centers = kmeans_200_dropoff.cluster_centers_

    # compute the average duration from cluster to cluster
    group_durations = (df
        .groupby(['pickup_200_cluster', 'dropoff_200_cluster'])['trip_duration']
        .mean()
        .reset_index()
        .rename(columns={'trip_duration': 'avg_cluster_duration'}))
```

```
    # merge the average duration from cluster to cluster with the main dataframe
    df = pd.merge(
        left=df, right=group_durations, how='left',
        left_on=['pickup_200_cluster', 'dropoff_200_cluster'],␣
↪right_on=['pickup_200_cluster', 'dropoff_200_cluster'])

    # fill the missing values with the mean of the average duration from␣
↪cluster to cluster
    df['avg_cluster_duration'] = df['avg_cluster_duration'].
↪fillna(df['avg_cluster_duration'].mean())
    df.drop(columns=['pickup_200_cluster', 'dropoff_200_cluster',␣
↪'trip_duration'], inplace=True)

    return df


def generate_features(df, y=None):
    """
    Generates additional features based on the specified feature toggles and␣
↪appends them to the DataFrame.

    Parameters:
    - df (DataFrame): The input DataFrame containing base features.
    - y (array-like, optional): The array of target values. Required if␣
↪'avg_cluster_duration' feature is enabled.

    Returns:
    - feature_df (DataFrame): The DataFrame with added features.
    """
    # append the features to the dataframe
    feature_df = df

    # add the distance feature
    if features_toggle['distance']:
        feature_df = distance(feature_df)

    # add the weather feature
    if features_toggle['weather']:
        feature_df = add_weather_feature(feature_df)

    # add the google distance feature
    if features_toggle['google_distance']:
        feature_df = add_google_distance(feature_df)

    # add the avg_cluster_duration feature
```

```python
    if features_toggle['avg_cluster_duration']:
        # check if y is None
        if y is None:
            raise Exception("y must be passed to generate_features if␣
↪avg_cluster_duration is True")
        feature_df = add_avg_cluster_duration(feature_df, y)

    # return the feature dataframe
    return feature_df
```