NYC Taxi Duration Prediction

December 5, 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. This research question is very relevant for urban commuters and transportation systems and incorporates the myriad factors influencing taxi ride times, including traffic patterns, congestion, time of day, and external variables such as 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 individual convenience, the ability to predict taxi ride times carries practical implications for optimizing taxi fleet management, resource allocation, and enhancing overall traffic flow in the city.

Kaggle published the dataset we are working on 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 also includes fields for the 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 our primary investigation is focused on determining taxi cab trip duration, we are also interested in several other questions. What dates, 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? Which pickup and drop off locations have the highest average time by distance? 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.

2 Feature Engineering

In our pursuit to enhance the predictive power of our model, we identified the necessity for additional features in our dataset. This realization led to the development of three distinct feature groups: datetime, distance, and weather.

2.1 Data Cleaning

The taxi cab duraction dataset from Kaggle contained several outlier data that required to 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 were far outside New York City, which we fixed by removed any point outside of city limits.

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`, `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, that 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 scaled 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. Images of these are 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 Appendix for the full code implementation.

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, 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 figure out which of our features is most important, we utilized L^1 regularization since L^1 regularization naturally sets unneeded feature coefficents to 0, and typically out performs step-wise feature removal.

[2]:
```
# lasso_feature_selection performs feature selection using the LassoLarsIC␣
↪method
lasso_feature_selection(X, y)
```

```
Optimal Alpha     : 1.0806
Optimal BIC       : 18056884.5229
Lasso Coefficients: [ 0.    , 0.     , 0.    ,  0.    ,  0.    ,  0.    ,  0.
, 0.    , 0.    , 0.    ,
                      0.    , 0.     , 0.    ,  0.0169,  0.    ,  0.    ,  0.
, 0.    , 0.    , 0.    ,
                     -0.0345, -0.1089, 0.    ,  0.0147,  0.0043]
Important Features: ['pickup_minute', 'distance_km', 'temperature_2m (°C)',
'cloudcover (%)', 'avg_cluster_duration']
```

4 Data Exploration and Visualization

4.1 Data Exploration

Before visualizing the data, we first want to do basic data exploration, especially after adding the features mentioned above.

For this project, we built the function `get_X_y()` that performs basic feature engineering, and returns two Pandas dataframes, X and y. Additionally, we built the function `generate_features()` that performs more advanced feature engineering, returning again two Pandas dataframes, X and y.

[20]:
```
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,))
```

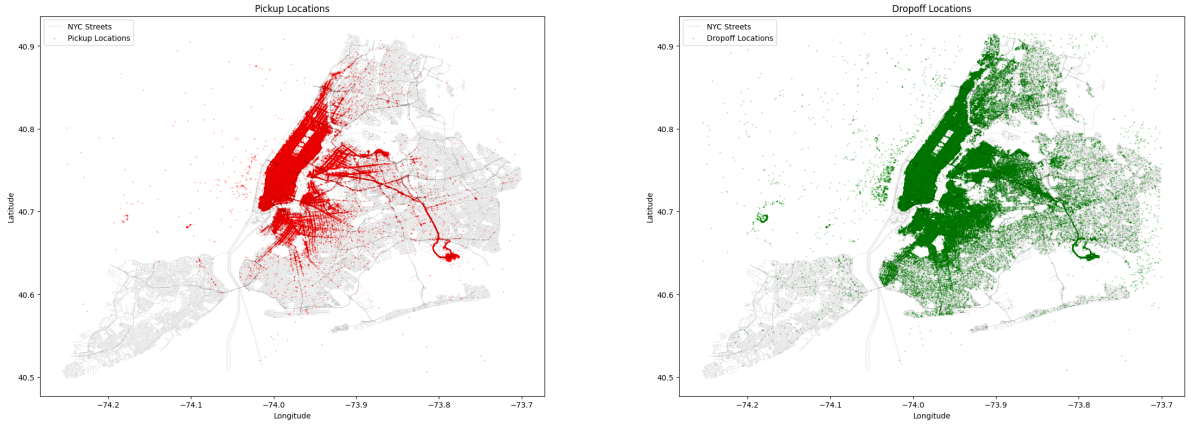
[28]:
```
print("Example of the features for the first row of the data:")
print("{:>20} = {:<20} {:>20} = {:<20}".format("Feature", "Value", "Feature",␣
↪"Value"))
for i in range(0, len(feature_X.columns)-1, 2):
    col1, val1, col2, val2 = ( feature_X.columns[i], feature_X.iloc[0][i],␣
↪feature_X.columns[i + 1], str(feature_X.iloc[0][i + 1]))
    print(f"{col1:>20} = {val1:<20} {col2:>20} = {val2:<20}")
col1, val1 = ( feature_X.columns[i+2], feature_X.iloc[0][i+2])
print(f"{col1:>20} = {val1:<20}")
```

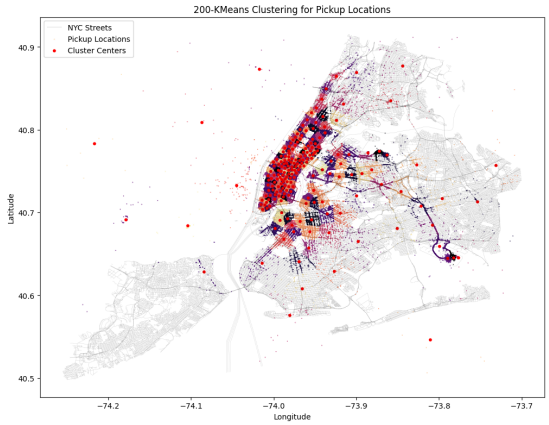
```
Example of the features for the first row of the data:
             Feature = Value                          Feature = Value
           vendor_id = 1                    pickup_datetime = 2016-03-14
17:24:55
     passenger_count = 1                   pickup_longitude =
```

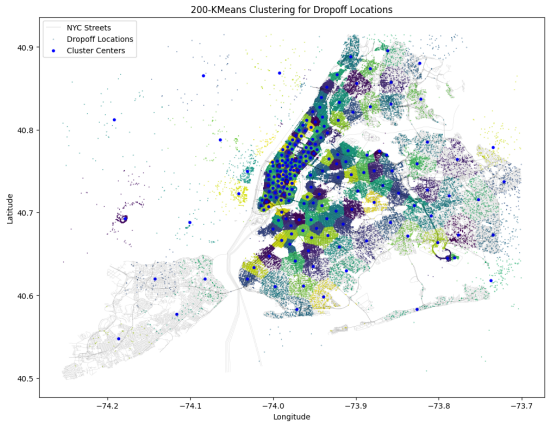
```
-73.98215484619139
     pickup_latitude = 40.76793670654297       dropoff_longitude =
-73.96463012695312
    dropoff_latitude = 40.765602111816406           pickup_month = 3
   pickup_day_Monday = 1                      pickup_day_Saturday = 0
   pickup_day_Sunday = 0                      pickup_day_Thursday = 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 =
-0.9659258262890683
     pickup_hour_cos = -0.25881904510252063 pickup_datetime_norm =
0.40508581306349817
         distance_km = 2.2082549595298886    temperature_2m (°C) = 6.4
  precipitation (mm) = 0.2                        cloudcover (%) = 100.0
avg_cluster_duration = 814.7282608695652
```

4.2 Data Visualization

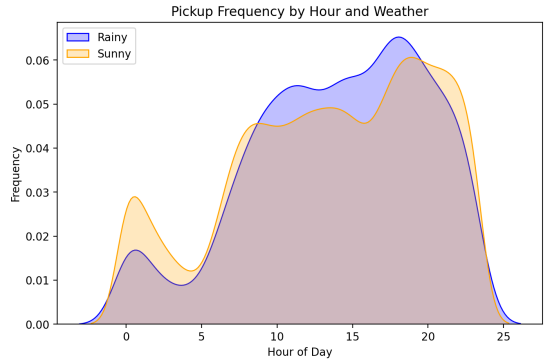
In the following figure, the top two graphs visualize the pickup and dropoff locations overlaid over a map of NYC. The bottom two graphs shows the pickup and dropoff locations clustered into groups using K-means clustering. The pickup locations are more heavily clustered around downtown (Manhattan), while the dropoff locations are more evenly distributed throughout the city.

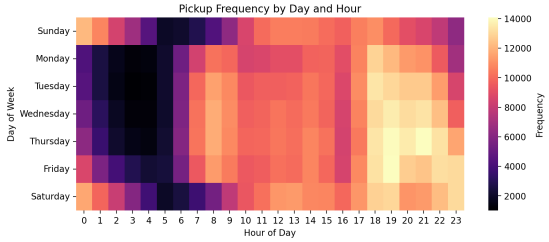






We can also learn about the factors that influence a New Yorker's decision to take a taxi from the data. For example, in the figure below, the graph on the left displays the most popular times of day to hail a taxi, which peeks around 6:00 PM. We also see that poor weather encourages more people to take 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.





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

5.1.1 Light GBM with Grid Search

We implemented a grid search over hyperparameters on LightGBM to identify the best version of this model to predict taxi trip duration. This model that runs quickly with the high dimensional data in our dataset.

[4]:
```
# The above function takes 45 minutes to run and produces the following results
lightgbm_hyperparameter_search(X, y)
```

```
{'Best parameters from grid search': {'boosting_type': 'gbdt', 'learning_rate':
0.01, 'max_depth': 20,
                                      'n_estimators': 100, 'num_leaves': 30,
'reg_alpha': 0.1, 'reg_lambda': 0.5},
'Best RMSE': 606.9699}
```

5.1.2 Lasso Regression

[6]:
```
# Lasso Regression is quite quick. This took only a few seconds.
lasso_regression_model(optimal_alpha=1.0806)
```

```
{'RMSE: 606.9680'}
```

5.1.3 XGBoost

[5]:
```
#Above function takes 9 hours to run and produces the following results
xgboost_hyperparameter_search(X, y)
```

```
Best parameters from grid search:  {'alpha': 0.1, 'booster': 'gbtree',
'learning_rate': 0.01, 'max_depth': 10, 'n_estimators': 100}
```

[9]:
```
xgboost_model(X, y)
```

[9]:
```
{'RMSE': 603.7398110737926}
```

5.1.4 Random Forests

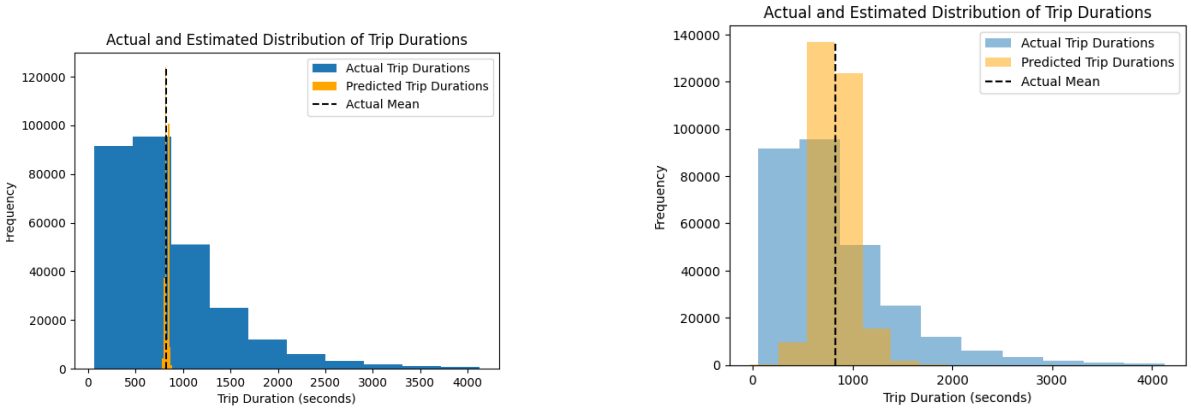
[2]:
```
# This function takes 6 hours to run and computes the optimal parameters
# for a RandomForestRegressor using 4-fold cross validation and attains
# the Root Means Squared Error as follow
random_forest_gridsearch()
```

```
Best parameters RandomForest: {'max_depth': 3, 'max_features': 'log2',
'min_samples_leaf': 2, 'n_estimators': 200}
RandomForest RMSE: 605.3372479314295
```

5.1.5 Model Hyperparameters

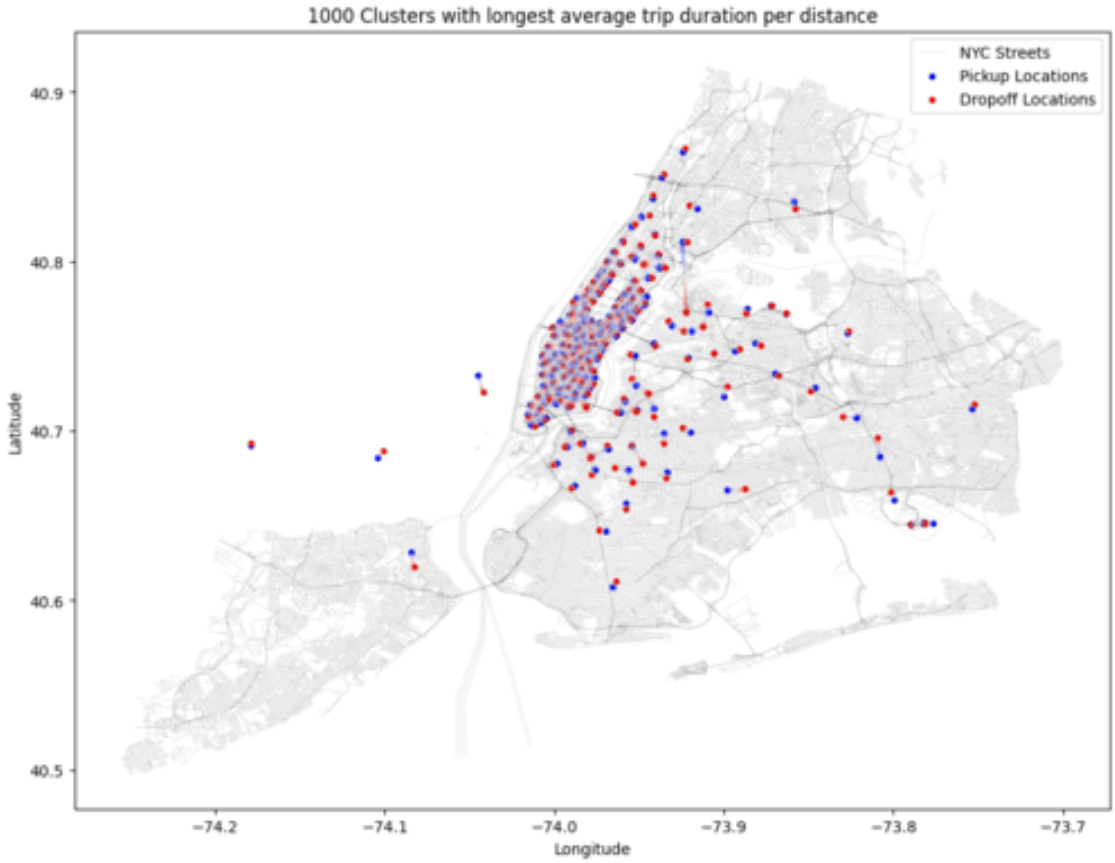
Parameter	Lasso Params	LightGBM Params	LightGBM_large Params	RF Params	XGBoost Params
boosting_type	–	gbdt	gbdt	–	gbtree
learning_rate	–	0.01	0.01	–	0.01
max_depth	–	20	50	3	10
n_estimators	–	100	100000	200	100
num_leaves	–	30	500	–	–
reg_alpha	1.0806	0.1	0.1	–	0.1
reg_lambda	–	0.5	0.5	–	–
max_features	–	–	–	log2	–
min_samples_leaf	–	–	–	2	–
Best RMSE	606.9680	606.9699	622.4705	605.1684	603.7398

6 Interpretation



The plot on the left demonstrates the predicted distribution of a LightGBM model with only 100 estimators, 30 leaves, and a max_depth of 50. The plot on the right demonstrates a similar LightGBM model but with 100,000 estimators, 500 leaves and a max_depth of 50. Interestingly, as we increase the compute abilitity of the model, the predictive distribution better matches the overall distribution; however, this model has a slightly worse RMSE. This evidences that the model is less overfit.

Also note that both models tend to predict the mean. This implies that the feature space is not large enough for the model to create a more accurate approximation of the distribution. If we had more compute resources and time, we could raise model parameters to higher values and create more features. This would allow the model to better fit the actual distribution of trip durations.



Here, we depict which cluster pairs on average take the longest time to travel between them per distance. Interestingly, some of the cluster paths cross major roadways. This implies that there should be more support roads that cross under or over the major highways. Also, 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 delves into a comprehensive dataset meticulously crafted by tracking the lifestyle patterns of real individuals residing in New York throughout 2016. This exploration raises paramount concerns surrounding privacy and the responsible usage of data pertaining to individual behaviors, locations, and travel patterns. To safeguard the integrity of this data, our dataset and model meticulously exclude all personally identifiable information, ensuring that only aggregate information is extracted. This approach guarantees that the distinctive patterns of individuals remain indiscernible, prioritizing privacy and ethical data handling.

However, the potential risks associated with the misinterpretation or misuse of our predictive models cannot be understated. Users may misconstrue predictions, leading to inappropriate decision-making. Unintentional biases within our predictive model may persist, potentially reinforcing inappropriate associations with specific neighborhoods and taxi services. Additionally, users may

overlook the inherent uncertainties and predictive nature of the model, treating its estimates as certainties.

Such misinterpretations could have tangible consequences. For example, if taxi companies or transportation authorities solely rely on the model's predictions without considering broader traffic management strategies, it may inadvertently exacerbate congestion in certain areas. Deploying our model in conjunction with algorithms influencing taxi availability could create self-fulfilling feedback loops, disproportionately impacting specific areas or demographics.

To mitigate these challenges, transparent communication about the model's limitations, potential biases, and intended use is paramount. We emphasize the importance of providing:

- Educational resources
- User-friendly interfaces
- Comprehensive documentation

Implementing fairness-aware algorithms is essential to ensure ethical deployment. Regular assessments, periodic audits, and proactive interventions are crucial to prevent the reinforcement of existing biases.

Furthermore, we recognize our ethical responsibilities, including the responsible disclosure of findings, ensuring public benefit from the research, and avoiding unintentional harm. Actively engaging with potential stakeholders and the community fosters ethical practices, addressing concerns and promoting responsible data science. Our commitment extends to ongoing efforts to uphold the highest standards of privacy, responsibility, and transparency in our research endeavors.

8 Appendix

8.1 Code

[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
```

[ ]:
```python
# KMeans Clustering code
def kmeans_pickup_dropoff_model(df, n_clusters=200):
    # 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
```

[ ]:
```
# lasso feature selection
def lasso_feature_selection(X, y):
    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}
```

[ ]:
```
# LightGBM Hyperparameter Selection
def lightgbm_hyperparameter_search(X, y):
    # 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)}
```

[4]:
```
# LightGBM Hyperparameter Selection
def xgboost_hyperparameter_search(X, y):
    # 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_)
```

[8]:
```
def xgboost_model(X, y):
    # 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)}
```

[ ]:
```
# Lasso Regression Model
def lasso_regression_model(optimal_alpha):
    # 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)}
```

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

    # 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)
```

[ ]:
```
# 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):
    # plot the nyc streets
    plt.gcf().set_dpi(500)
```

```
    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):
    # plot the nyc streets
    plt.gcf().set_dpi(500)
    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):
    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):
    # plot the nyc streets
    plt.gcf().set_dpi(500)
    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")

    # 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):
    # plot the nyc streets
    plt.gcf().set_dpi(500)
    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()
```

[ ]:
```
def plot_actual_vs_predicted_distributions(show=False):

    # 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()
```

[ ]:
```
####################
# DATA_MANAGER.py #
####################

"""
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
```

```python
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
    """

    # 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
    """
    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")


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


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

[ ]:
```
###############
# FEATURES.py #
###############

"""
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.
    """

    # 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):

    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):

    # 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
    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
```