CSE 176 - Model Project

Gavin Abrigo, Ronan Tangaan, Christopher Hernandez

Model (Overview)

Histogram Gradient Boosting Classifier (/Regressor)

- An estimator faster than GradientBoostingClassifier for large datasets (n >=10k)
- Inspired by LightGBM (Microsoft's gradient booster)
- Has a branch for regression/classification instead of needing to create a way to do one or the other with a model that only has either/or.

Histogram Gradient Boosting

- Feature Binning (Histograms)
  - Builds a histogram by dividing values (by default) into 256 bins (this reduces the candidate thresholds to 256 on default instead of thousands)
  - Stores only the index instead of the raw value
- Build Trees using gradient boosting
  - Boosting: builds many small trees, each correcting mistakes of previous trees
  - Gradient boosting: Each tree will calculate the gradient, and each new tree will give positive or negative pushes to the “right direction”, each one correcting the previous trees

Parameters used

Loss - (Objective function)

Learning_rate - (Step size)

Max_iter - (Max iterations, each iteration adds 1 tree to reduce remaining error)

Max_depth - (Max tree size)

Min/Max_samples_leaf - (minimum/maximum # of samples required in a leaf)

L2_regularization - (Penalizes complexity to prevent overfitting, L2 adds the extra part that makes small leaves worse)

Max_bins - (Maximum # of buckets per histogram)

Dataset 1: Taxi Regression

- Train.csv: Shape (145864, 11)

Data Fields

- **id** - a unique identifier for each trip
- **vendor_id** - a code indicating the provider associated with the trip record
- **pickup_datetime** - date and time when the meter was engaged
- **dropoff_datetime** - date and time when the meter was disengaged
- **passenger_count** - the number of passengers in the vehicle (driver entered value)
- **pickup_longitude** - the longitude where the meter was engaged
- **pickup_latitude** - the latitude where the meter was engaged
- **dropoff_longitude** - the longitude where the meter was disengaged
- **dropoff_latitude** - the latitude where the meter was disengaged
- **store_and_fwd_flag** - This flag indicates whether the trip record was held in vehicle memory before sending to the vendor because the vehicle did not have a connection to the server - Y=store and forward; N=not a store and forward trip
- **trip_duration** - duration of the trip in seconds

GOAL

- Predicting the total ride duration of taxi trips in NYC

EVALUATION:

- RMSLE:

$$\epsilon = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\log(p_i + 1) - \log(a_i + 1))^2}$$

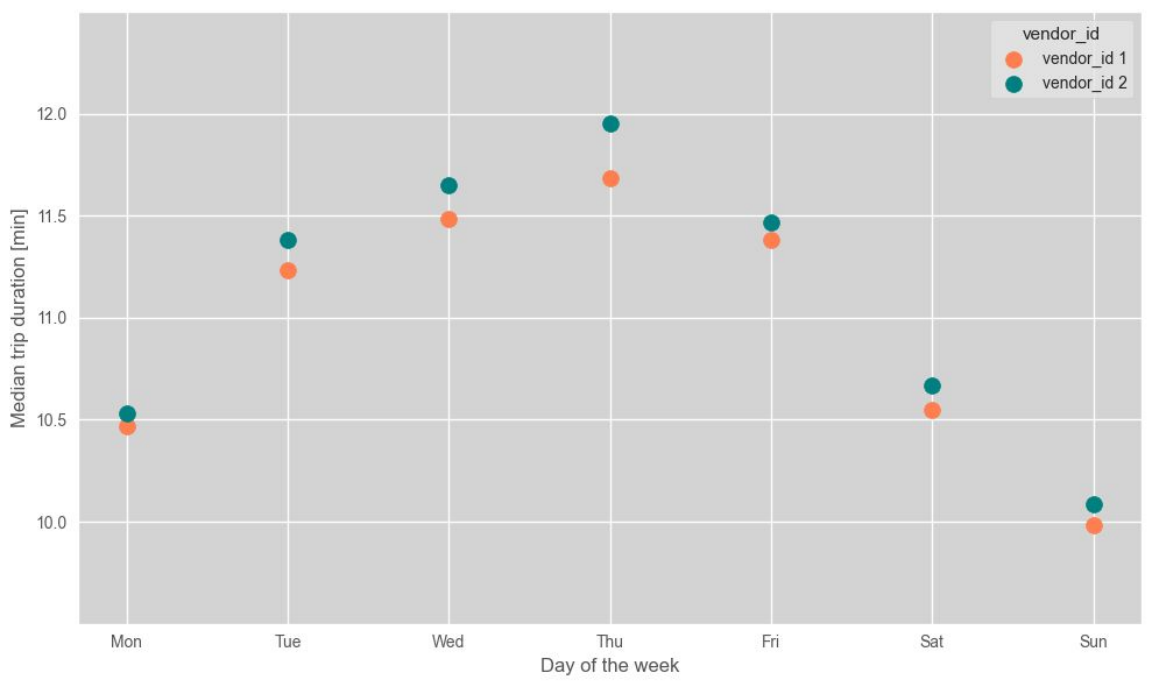
  - n: is the total number of observations in the dataset
  - p_i : is your prediction of tripduation
  - a_i : is the actual trip duration

- This metric was chosen to address:
  - Data skewing (trip times)
  - Percentage error
  - Robustness to outliers

Feature Visualizations

Most pickups are coming from Manhattan, other notable pickup spots are from JFK airport (about 200k points)

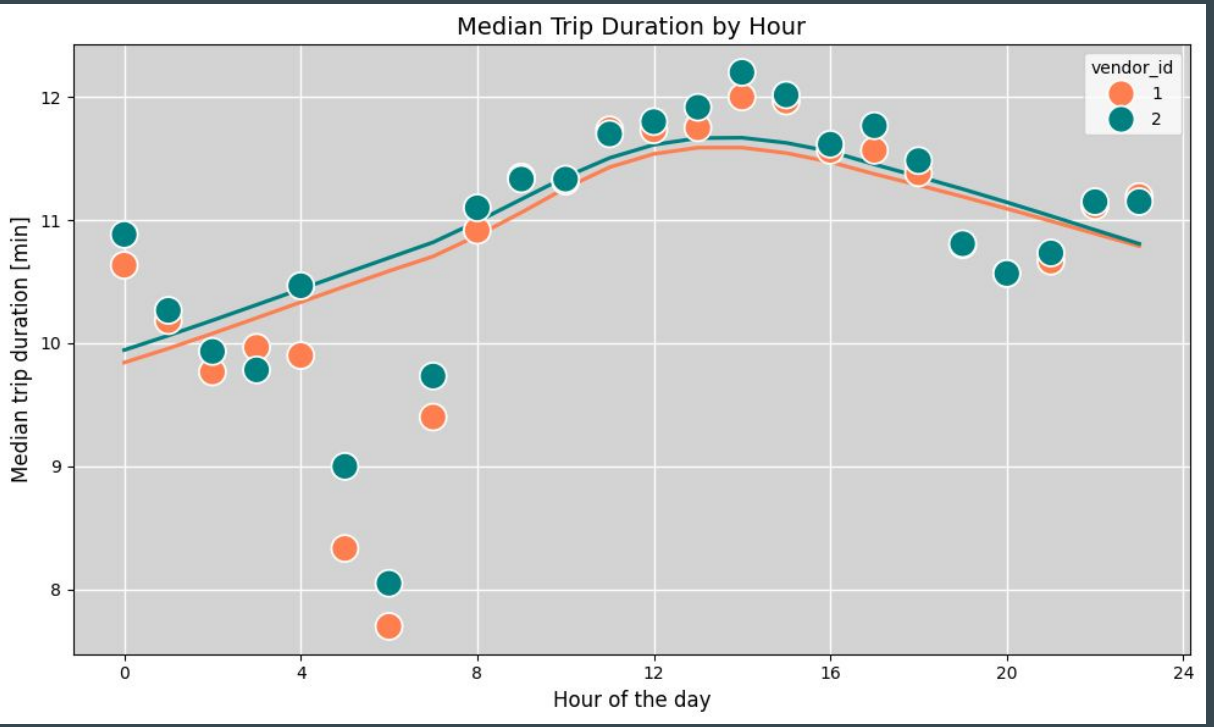
Feature Visualizations



- Here we want to observe variation in trip duration
- Do quieter days and hours lead to faster trips?

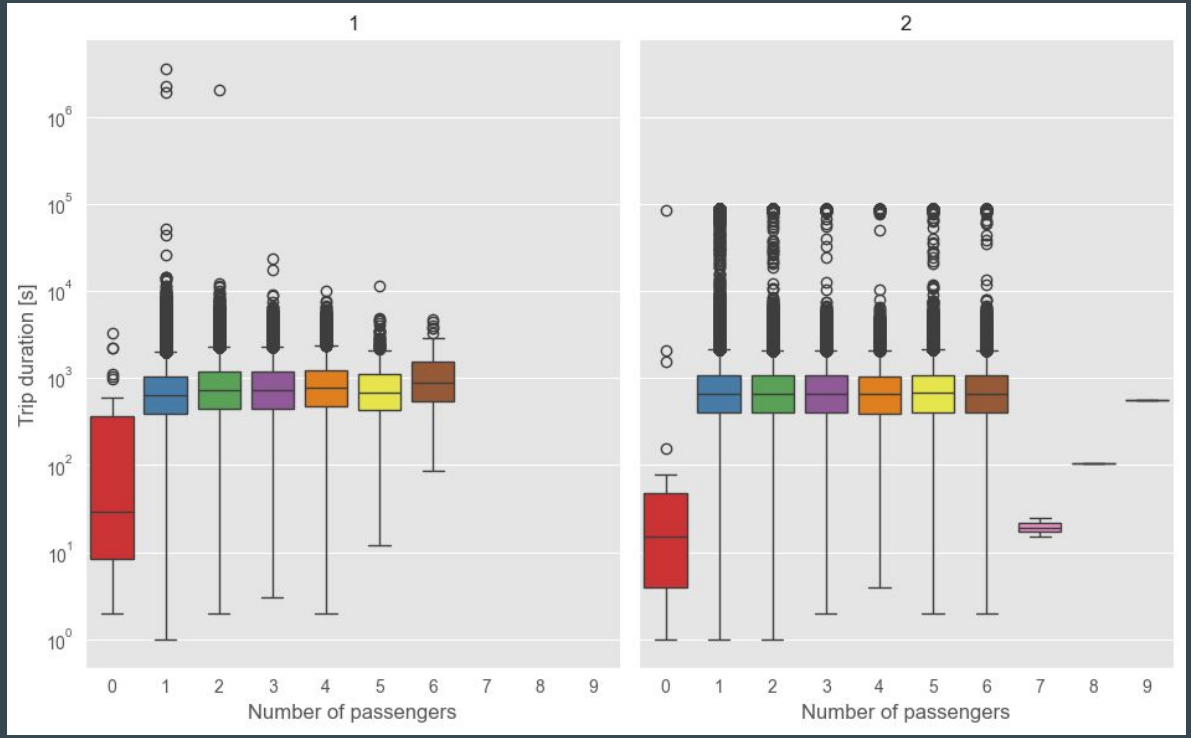
- We add vendor_id as a separate feature

Feature Visualizations



There is a pattern for days of the week and hour of days. Seems that trip duration tend to be quicker towards quieter hours in the day and quieter days in the week.

Feature Visualizations



- Without any passengers both vendors have short trips.

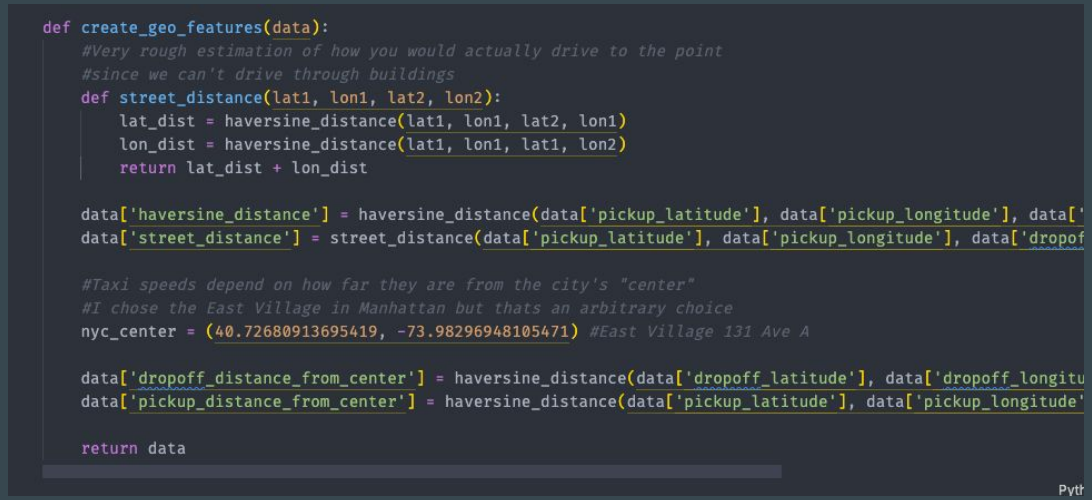
- Vendor 1 has trip_durations **beyond** 24 hours
- Vendor 2 has all (five) trips with more than 6 passengers and more trip durations that approach the 24 hour limit

Feature Creation

```python
# Calculate the shortest distance through 2 points
def haversine_distance(lat1, lon1, lat2, lon2):
    radius = 6371  #Earth radius in km
    lat1, lon1, lat2, lon2 = map[Any](np.radians, [lat1, lon1, lat2, lon2])
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    a = np.sin(dlat/2)**2 + np.cos(lat1) * np.cos(lat2) * np.sin(dlon/2)**2
    c = 2 * np.arcsin(np.sqrt(a))
    return radius * c

def create_datetime_features(data):
    data['pickup_datetime'] = pd.to_datetime(data['pickup_datetime'])

    #Extract time features
    data['pickup_hour'] = data['pickup_datetime'].dt.hour
    data['pickup_day'] = data['pickup_datetime'].dt.day
    data['pickup_dayofweek'] = data['pickup_datetime'].dt.dayofweek
    data['pickup_month'] = data['pickup_datetime'].dt.month

    #Pretty cool thing here cuz you map the time and day to a circle so the model understands
    #That 12AM (hour 0) and 11PM (23) are not far apart for example
    data['pickup_hour_sin'] = np.sin(2 * np.pi * data['pickup_hour'] / 24)
    data['pickup_hour_cos'] = np.cos(2 * np.pi * data['pickup_hour'] / 24)
    data['pickup_dayofweek_sin'] = np.sin(2 * np.pi * data['pickup_dayofweek'] / 7)
    data['pickup_dayofweek_cos'] = np.cos(2 * np.pi * data['pickup_dayofweek'] / 7)

    return data

# separate function to help with cleaning (can't use on test set due us knowing the trip_duration)
def travel_speed(data):
    data['speed'] = data['haversine_distance'] / (data['trip_duration'] / 3600)
    return data
```

- We used haversine distance to calculate the distance between the pickup and dropoff locations based
- Sine and cosine transformations were used to map cyclical features like the days of the week and time into a circle
    - This was done to make the model see that hour 0 (12AM) and hour 23 (11PM) are very close to each other and not a large jump

Feature Creation

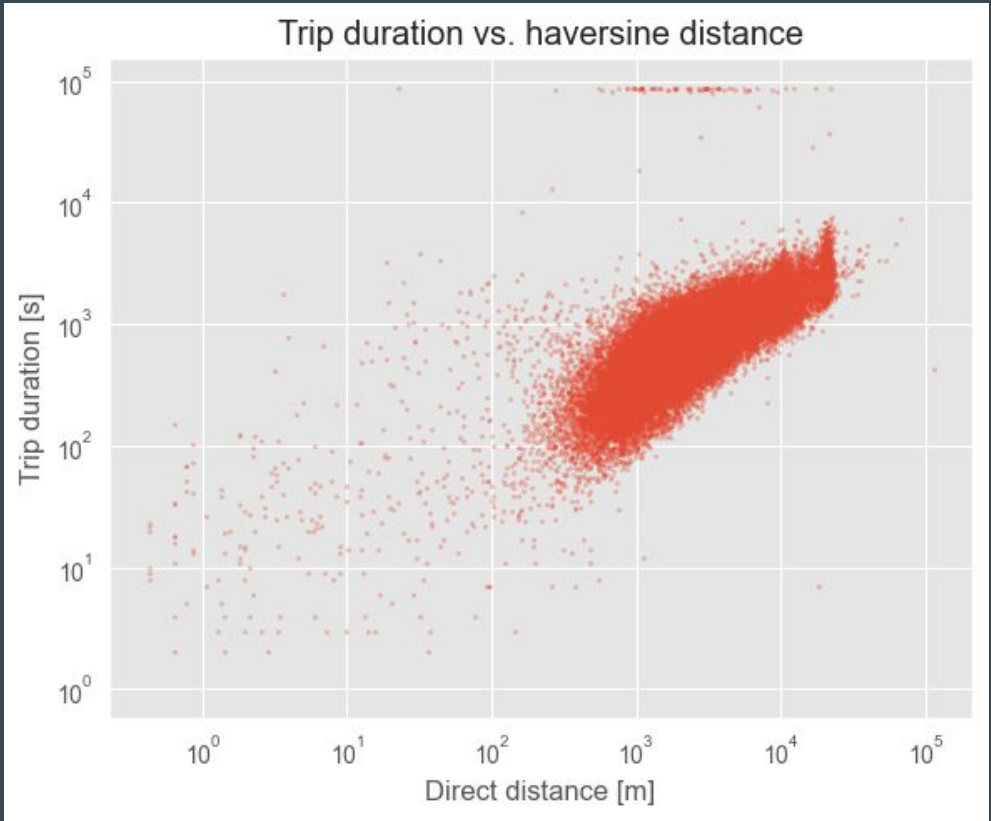


```python
def create_geo_features(data):
    #Very rough estimation of how you would actually drive to the point
    #since we can't drive through buildings
    def street_distance(lat1, lon1, lat2, lon2):
        lat_dist = haversine_distance(lat1, lon1, lat2, lon1)
        lon_dist = haversine_distance(lat1, lon1, lat1, lon2)
        return lat_dist + lon_dist

    data['haversine_distance'] = haversine_distance(data['pickup_latitude'], data['pickup_longitude'], data['
    data['street_distance'] = street_distance(data['pickup_latitude'], data['pickup_longitude'], data['dropof

    #Taxi speeds depend on how far they are from the city's "center"
    #I chose the East Village in Manhattan but thats an arbitrary choice
    nyc_center = (40.72680913695419, -73.98296948105471) #East Village 131 Ave A

    data['dropoff_distance_from_center'] = haversine_distance(data['dropoff_latitude'], data['dropoff_longitu
    data['pickup_distance_from_center'] = haversine_distance(data['pickup_latitude'], data['pickup_longitude'

    return data
```

Pyth

- In addition to the haversine distance, the street distance feature was created to give a more realistic estimation of the distance the trip covered.
  - Haversine distance calculates distance in a straight line, ignoring roads and buildings.
  - Street distance adds the difference between the longitude and latitude together to calculate distance
- The distance from NYC's "center" was also added because trip duration could be affected by increased traffic in more populated areas in New York

Feature Creation

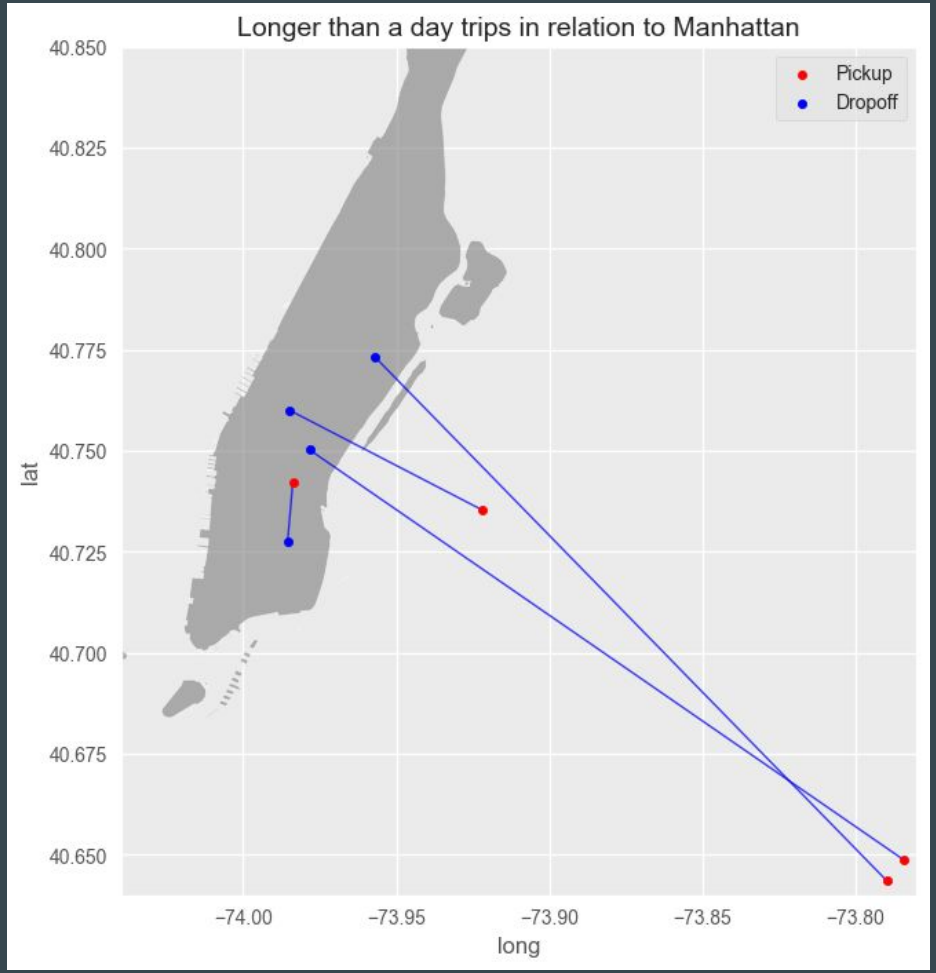


- Distance generally increases with trip duration (log-log space)
- Some trip durations towards the top of the graph look suspicious (will be included in cleaning)
- Numerous trips of very short distances

Cleaning

- Plotting trips that are > 24hrs

- Some of these trips from JFK airport are quite far, they are unlikely to be longer than a day in reality.

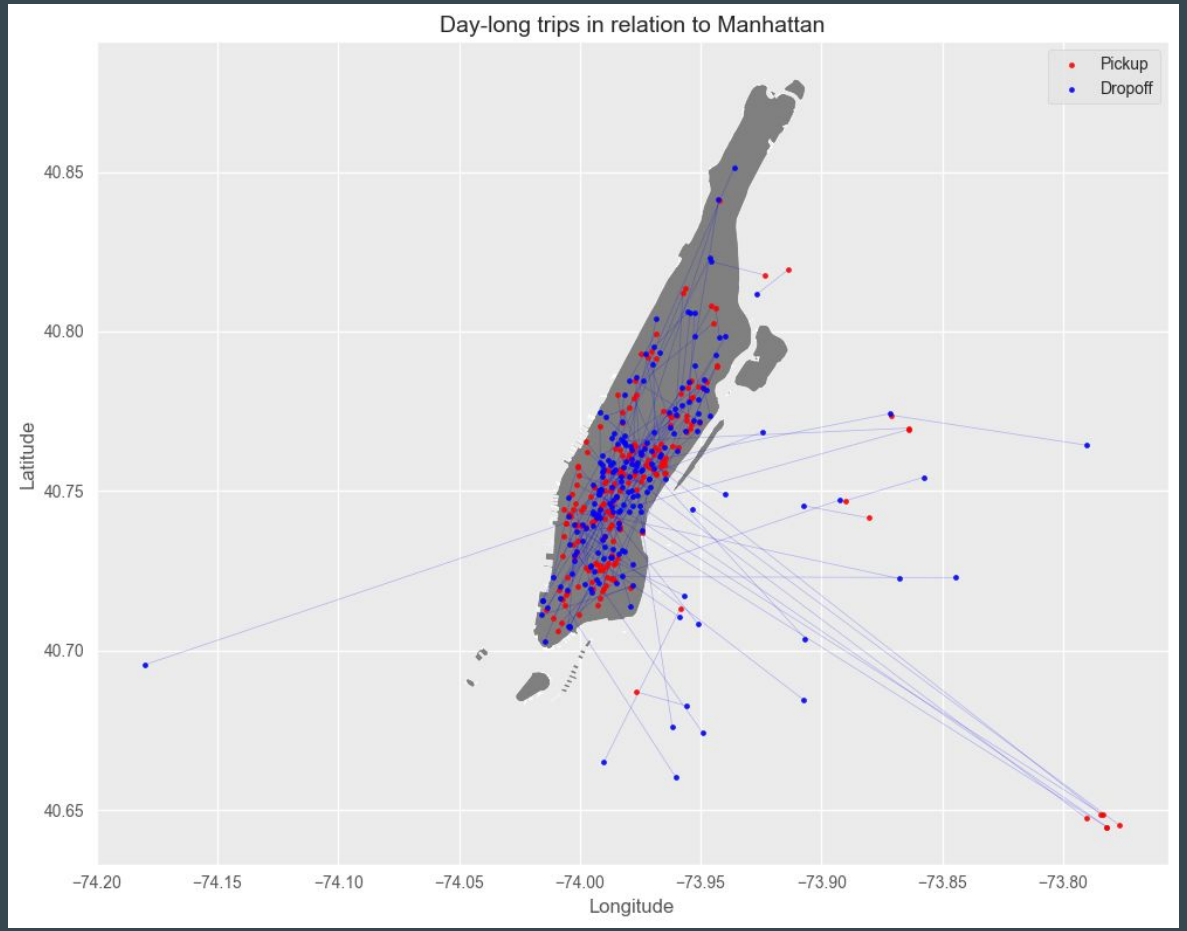


Cleaning

- Only Plotting 200 of the 1800 trips made for readability

- There are a few longer distances that stand out such as trips going to JFK airport, but they can be exceptions due to the fact NYC is a big urban city with heavy traffic.

NOTE: We left out trip durations longer than 22 hours from the exploration of our model



Cleaning

- Short trips with high avg speeds (km/h)

- The top speeds in km/h are unrealistic. So we can filter them from this dataset

trip_duration	haversine_distance	pickup_datetime	speed
7	18.034405	2016-02-13 20:28:30	9274.836731
282	320.125775	2016-04-02 20:33:19	4086.712020
2	0.783347	2016-06-12 06:35:13	1410.024743
51	19.948152	2016-06-19 20:18:35	1408.104838
279	104.759971	2016-03-20 21:07:56	1351.741558
2	0.703065	2016-06-23 13:36:48	1265.516683
20	6.911542	2016-05-28 15:14:19	1244.077521
3	0.912885	2016-05-30 17:12:12	1095.462175
4	1.029739	2016-01-17 03:11:56	926.765395
3	0.760976	2016-06-13 16:34:30	913.171260

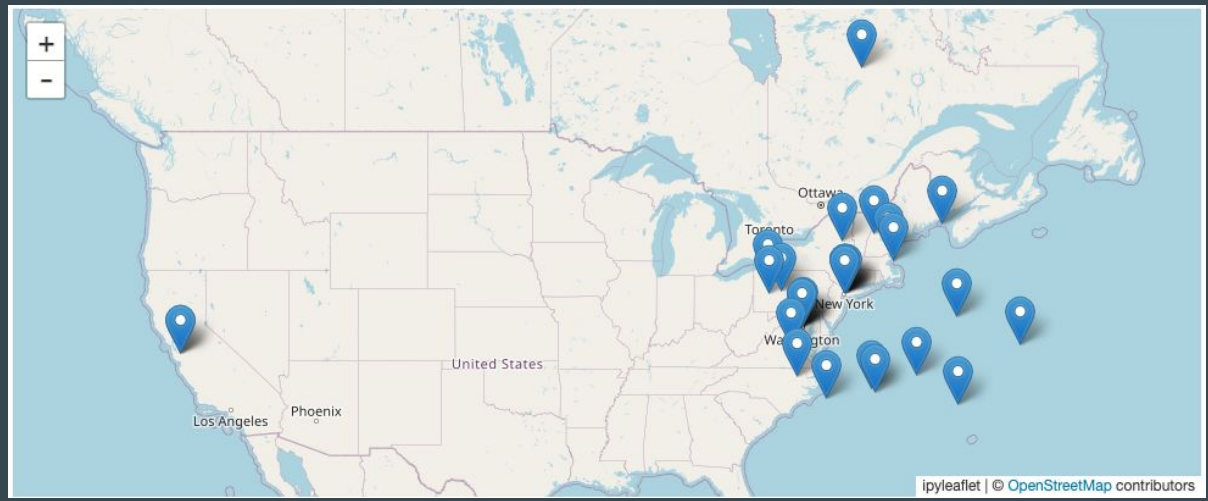
NOTE: We are leaving out speed as a feature in training because trip_duration is part of our testing set (it'd be cheating)
- speed = haversine_dist / trip_duration

Cleaning

- Coordinates of pickup/dropoff locations > 300km from NYC (JFK airport)

- Weird to see some locations in the ocean and in different parts of the continent

Weird Pickup/ Dropoff Locations



Final Cleaning

```
# Note: 3e5 meters = 300 km. Our haversine_distance is in km.
filter_mask = (
    (trainfull['trip_duration'] < 22 * 3600) &          # trip_duration < 22*3600
    (trainfull['trip_duration'] > 10) &                 # trip_duration > 10
    (                                                   # dist > 0 | (near(dist, 0) & trip_duration < 60)
        (trainfull['haversine_distance'] > 0) |
        (np.isclose(trainfull['haversine_distance'], 0) & (trainfull['trip_duration'] < 60))
    ) &
    (trainfull['jfk_dist_pick'] < 300) &                # jfk_dist_pick < 3e5 (300km)
    (trainfull['jfk_dist_drop'] < 300) &                # jfk_dist_drop < 3e5 (300km)
    (trainfull['speed'] < 100)                          # speed < 100 km/h (62 mph)
)
```

- Applying cleaning filter to training set

Training

```
hyperparams = {
    'learning_rate': [0.01, 0.05],
    'max_iter': [500, 1000],
    'max_depth': [5, 7, None],
    'min_samples_leaf': [10, 20],
    'l2_regularization': [0.1, 1, 10],
    'max_leaf_nodes': [63, 127],
    'max_bins': [255],
}

base_model = HistGradientBoostingRegressor(
    random_state=39,
    early_stopping=True,
    validation_fraction=0.2,
    scoring='loss'
)

random_search = RandomizedSearchCV(
    estimator=base_model,
    param_distributions=hyperparams,
    n_iter=5,
    scoring='neg_root_mean_squared_error',
    cv=3,
    n_jobs=-1,
    random_state=39,
    verbose=2
)
```

- To get the best model, we used RandomizedSearchCV, which takes a list of possible hyperparameter values and shuffles them. It trains multiple models with the random hyperparameters along a set range and scores each one on the validation set using RMSE. The model with the lowest error is chosen.
    - RandomizedSearchCV was chosen to tune hyperparameters because of its speed. When using GridSearchCV, tuning took 45+ minutes to run
- The training set was split so that 20% (2917282 rows) of it became the validation set and 10% (1458641 rows) became the test set.

Results

The best parameters we got for our model were: 'min_samples_leaf': 10, 'max_leaf_nodes': 127, 'max_iter': 1000, 'max_depth': None, 'max_bins': 255, 'learning_rate': 0.01, 'l2_regularization': 0.1

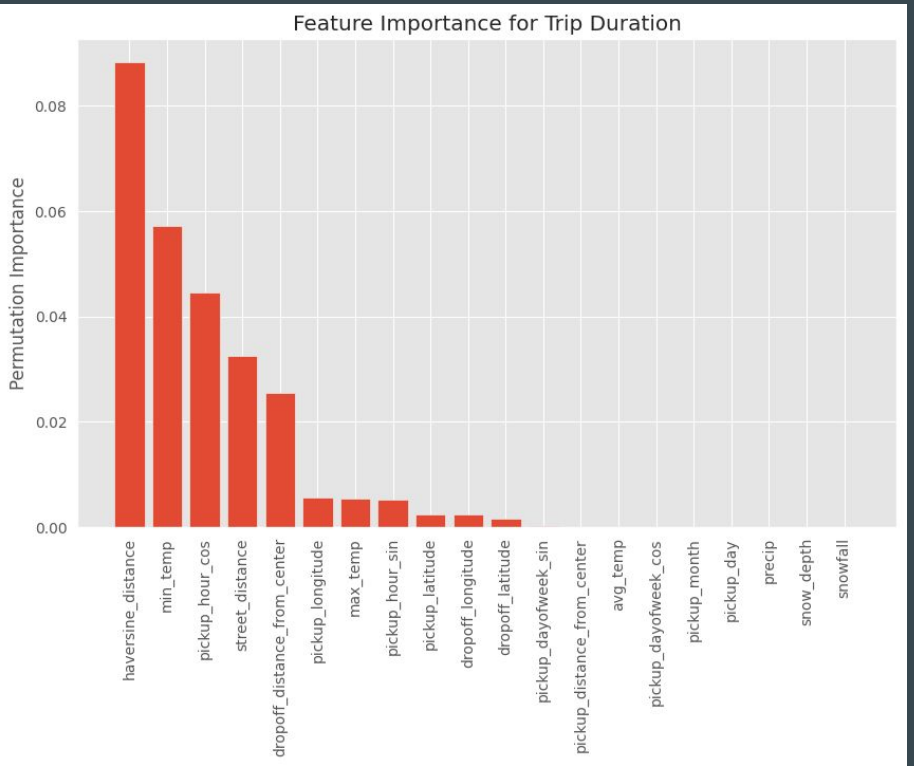
After cross-validating on the test set, our model's prediction for the trip duration was off by 614 seconds. Not very good for predicting short trip durations, but somewhat accurate for long trips.

```
val_rmse = np.sqrt(mean_squared_error(y_test, y_val_pred_log))
print(f"Validation RMSE: {val_rmse}")
```

```
Validation RMSE (in seconds): 614.6298966775482
Validation RMSE: 0.34248816085690903
```

Dataset 1

```
from sklearn.inspection import permutation_importance

# Compute permutation importance (slightly slower, but works for all versions)
result = permutation_importance(model, X, y, n_repeats=10, random_state=42, n_jobs=-1)
```



- "Feature importance"
  - Not a perfect metric as it has many flaws (i.e. correlated feature underrepresentation)
- This graph was created by using permutation_importance.
- Scaling and outliers problem

Dataset 2: Credit Card Fraud Classification Overview

Key Idea

We want a system that can automatically detect transactions that look suspicious or normal, and flag them for customer and company use.

(we want a model that can find patterns for fraud detection)

Dataset 2 Info

- The dataset is made up of transactions by credit cards in September 2013 from European cardholders.
- This dataset presents transactions that occurred in two days, with 492 frauds out of 284,807 transactions
- “Due to confidentiality issues”, the dataset can’t provide details on what the other features (V1-V28) are, but we know that these are principle components obtained with PCA.
    - The only non-transformed features are the time, and amount.
        - Time = seconds elapsed between each transaction + first transaction in dataset
        - Amount = transaction amount

Dataset 2: Credit Card Fraud Classification

- Heavily Imbalanced dataset (~.172% of transactions are fraud)
  - Accuracy is a terrible metric
  - ~284,000 normal transactions -> ~492 frauds = 1 fraud : 577 normal
- Split 80% of samples into training, 20% into testing with stratification (proportionality)
- ROC-AUC is over represented due to the dataset, PR-AUC (precision recall) is better, as it represents the minority class

Parameters

```
model = HistGradientBoostingClassifier(
    learning_rate=0.1,
    max_depth=6,
    max_iter=300, [technically not what it ended with obviously]
    l2_regularization=1.0,
    min_samples_leaf=20,
    early_stopping=True,
    class_weight="balanced"
)
```

Training / Classification

- We split the model in an 80/20 split (training and test set respectfully), we also used early_stopping in our model (10% of training)
  - In reality this means the real split is 72/8/20
- The model tries to create rules to split groups (i.e. If x is very low and y is high this might be fraud), then combines all of these rules (different trees and nodes).
- Class balancing = true
  - If no balancing was put into place, it would make the dominant class standout, balancing is necessary because it makes fraud cases count more
- Predicts on a log loss (entropy) function, once again with early stopping ending with 41 trees on our model.

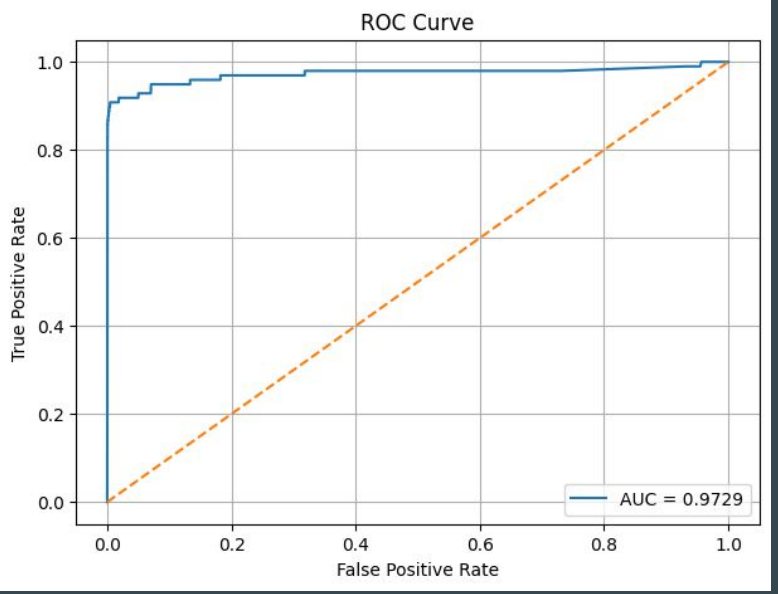
Actual performance:

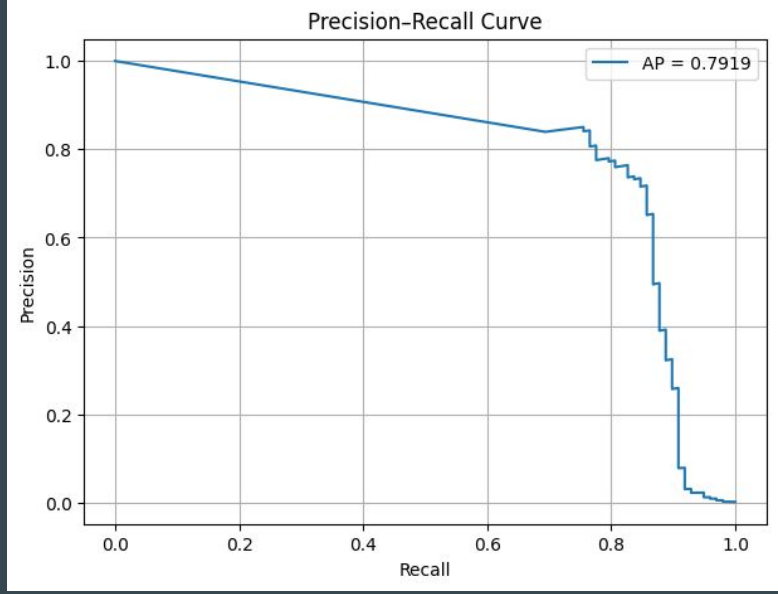
- Choose the rows that are fraud, scale the data again
- For each row predict if it's a fraud
- Calculate the % of frauds detected

```
frauds = df[df["Class"] == 1].drop("Class", axis=1).copy()

# scale the fraud rows
frauds[["Time", "Amount"]] = scaler.transform(frauds[["Time", "Amount"]])

fraud_probs = model.predict_proba(frauds)[:, 1]
fraud_preds = model.predict(frauds)

print("Percent of frauds detected:",
      fraud_preds.mean() * 100, "%")
```

```
Percent of frauds detected: 96.54471544715447 %
```

96% of frauds (w/ conditions)





- As we can see, Receiver Operating Characteristic is very misleading, it's accuracy is ~98%. Once again, this is due to the overrepresentation of true positives.

- We can see that the model is relatively flat until .7 at ~80% precision.
- The **A**verage **P**recision is .7919 (Strong/High)
- As a reminder, a random guess: = ~.172%

Dataset 2 Insights

- AP = .79 (far above random baseline of .0017)
- Excels at identifying most suspicious transactions with high confidence.
  - The threshold for classification can also be changed to aggressively chase final fraction of fraud leads (obviously may increase false positives too in accordance w/ graph earlier).
- 96% of frauds labeled properly (w/ default threshold of .5)

Thank you!

Gavin Abrigo, Ronan Tangaan, Christopher Hernandez