CSE 176 - Report 3

Gavin Abrigo, Ronan Tangaan, Christopher Hernandez

Introduction

For this part of the project, we were assigned to pick out our own datasets and analyze them. We chose one classification dataset and one regression dataset. The algorithm we chose was the *HistGradBoostRegressor* and *HistGradBoostClassification* for our respective classification and regression datasets. This algorithm differs from traditional gradient boosting by applying histogram based approaches to compute splits. We begin by binning feature values into histograms, significantly reducing the number of potential split points, which reduces training time as a result. Compared to traditional gradient boosting which does an exhaustive search across all feature values.

Datasets:

REGRESSION: New York City Taxi Trip Duration

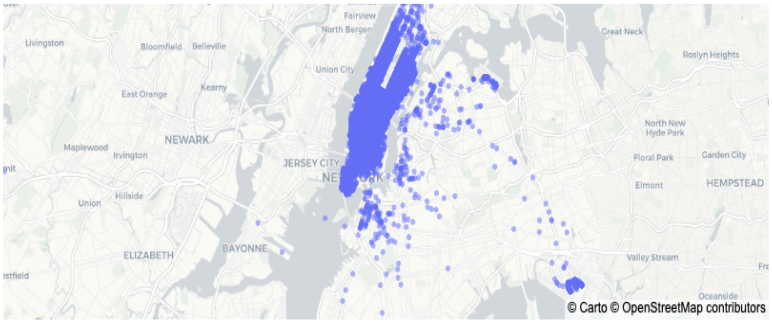
For our regression dataset we chose the New York City Taxi Trip Duration dataset from Kaggle. Kaggle provides two files–train.csv with a shape of (145864, 11). This dataset included features such as:

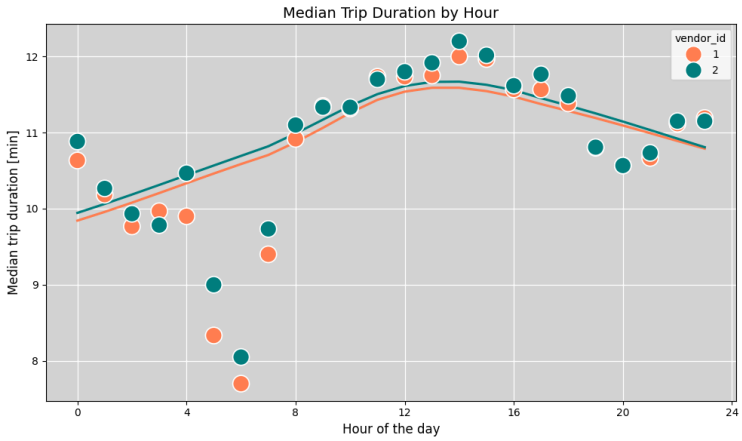
- **id** - a unique identifier for each trip
- **vendor_id** - a code indicating the provider associated with the trip record
- **pickup_datetime** - date and time when the meter was engaged
- **dropoff_datetime** - date and time when the meter was disengaged
- **passenger_count** - the number of passengers 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 (TARGET)

The goal of our dataset was to predict total *trip_duration* time.
Our evaluation metric was RMSLE (root-mean-log-square-error) which was provided in the Kaggle competition. The reason we stuck with this for our project compared to RMSE was because RMSLE penalizes underestimation less harshly than RMSE because the log compresses scale. This makes sense for longer trip durations–the impact of which may be reduced.

Before we began training the model, we added extra features to the dataset and cut some out. We ignored the store_and_fwd_flag, vendor_id, and passenger_count features since they likely wouldn't improve the model's accuracy. We added new features like the haversine distance between pickup and dropoff locations, the distance of the pickup location from Times Square, and an estimation of the distance traveled when taking into account New York's grid based streets. We also modified the pickup and dropoff date time features and mapped them onto a circle to represent time and days of the week cyclically. We did this so that there would not be a discontinuity between 11PM and 12AM and Saturday and Sunday.

FEATURE VISUALIZATION:

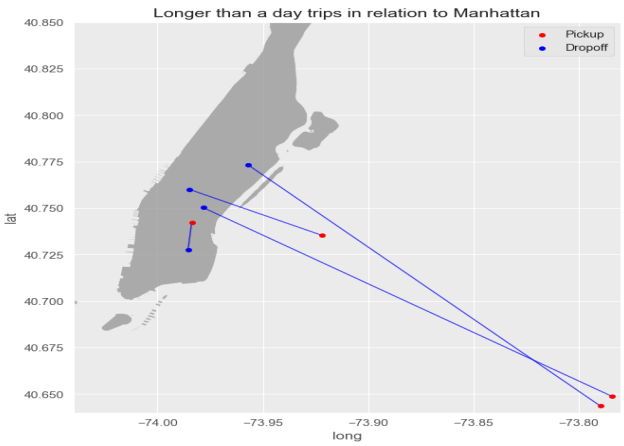


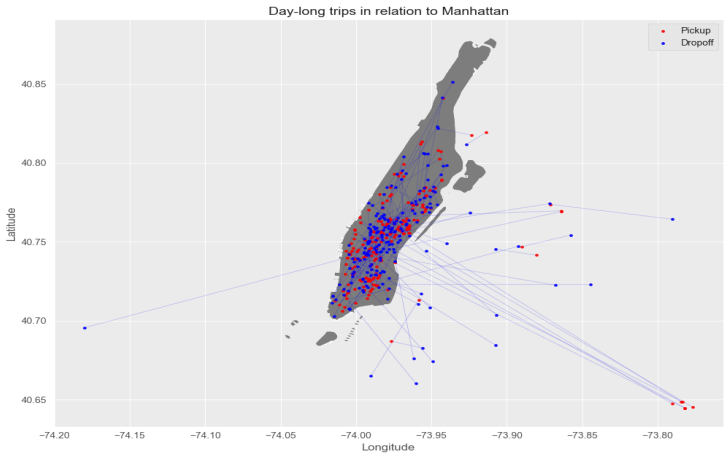


Here in this section we visualize some of the features to get a better understanding for our dataset. From the graph on the left we plot all pickup coordinates in NYC. As we can see, most of the pickup locations from the graph are from the greater Manhattan area. However, we notice there are a couple outliers towards the southeast. We find this to be JFK airport which makes sense as many taxi traffic does pass through there.

In the next graph we compare the hour of the day to the median trip duration. The reason we are using median and not the mean is because median trip duration is robust to outliers. Using the median prevents a few long trip durations from distorting the daily pattern, allowing us to make a clearer comparison of how standard trip durations vary throughout the day. Notably, we also added the *vendor_id* to the graph to observe if there was a disparity of trip durations when comparing vendors, but we found there to be little to no variation. We also added a smoothing layer to the graph to observe the extent of the variation. Over the course of the day, past noon, we can observe that trips peak but then take a dip around 8pm (20:00).

CLEANING

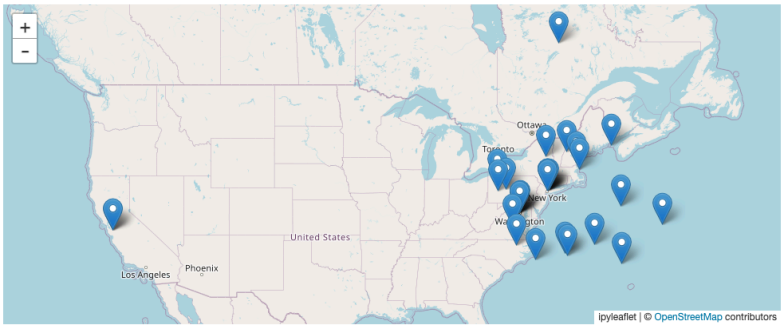




In our cleaning section, we graph and highlight cases that stood out to help us remove outliers that would affect our prediction metric. From the first two graphs we plot trips that were close to 24 hours or longer than. As you can see from the first graph on the left we see that there are only about 4 trips that last longer than 24 hours.

The second graph depicts trips that are close to 24 hours long. We decided to set a limit at 22 hours as an unrealistic travel time for a taxi trip and we got around 1800 trips. However, for the sake of readability we are only plotting 200. We find that there are a few longer distances that stand out such as trips going to JFK airport, but they can possibly be exceptions due to the fact NYC is a big urban city with heavy traffic.

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



The first image on the left is a dataframe in which we are emphasizing short trip durations with high average speeds. We calculate the speed in km/h from: $speed\ (km/h) = \frac{haversine\_distance\ (>0km)}{trip\_duration\ (<5\ min)}$. We can see after applying this filter that there are abnormally high speeds for taxis that go over 9000 km/h which is unrealistic.

In the second picture on the right we are plotting trip coordinates that are over 300 km from NYC (more specifically JFK airport). These are just some interesting pickup/dropoff locations that are just flat out ridiculous that we will be removing during cleaning

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)
)
```

Above is our final cleaning filter that we will be applying to our training set. Here we are filtering trip durations in the range of [10 sec - 22 hours], haversine distances (> 0km) or distances that are close to 0km with trip durations less than a 60 seconds, and JFK airport pickup/ dropoff distances (< 300km), and finally average speeds (< 100 km/h) as a reasonable average speed threshold for NYC taxis.

TRAINING:

Before training the model, we split the dataset into the training, validation, and test sets into a split of 70/20/10% respectively. To train the model, we enabled early stopping to help prevent the model from overfitting, leaving the scoring method to the default value. To tune the hyperparameters, we used a randomized search on hyper parameters (implemented with scikitlearn's RandomizedSearchCV. We preferred to use this over GridSearchCV because of its speed; RandomizedSearchCV only tries a fixed number of parameter settings whereas GridSearchCV tries all of them. While it is likely that GridSearchCV could've given our model better performance, training the model took far too long and the time saved with RandomizedSearchCV outweighed the potential loss of potentially hyperparameters. The hyperparameter distribution we made had mostly arbitrary values so we could just see what would happen. We left the CV folds value to the default of 3, and set the number of parameter settings that are sampled to 5 because it felt like a good middle-ground between speed of training and quality of hyperparameters. To score the performance models against the test set, we used the root mean squared error estimator.

RESULTS

After training the model, we used the test set to see how well the model could generalize. For our RMSLE score, we managed to obtain a value of 0.342, which means we are predicting around a 35% average error which is reasonably acceptable.

But to help us understand the context of what the scoring metric means, we also include the RMSE. The model has an RMSE of ~614.63, which means that on average, a prediction for trip duration was off by 10 minutes. It's not terrible for long trips where the difference of 10 minutes doesn't really matter, but it's bad for short ones where a 10 minute difference practically makes the estimation useless.

NOTE: Before the *train_test_split* call we apply a log transformation (as seen below) to the cleaned prediction dataset as an alternative instead of just applying RMSLE directly to our *random_search* scoring parameter

```
Y_clean = np.log1p(Y_clean)
```

CLASSIFICATION: Credit Card Fraud Detection

Algorithm Description

For the classification task, we used the HistGradientBoostingClassifier, an ensemble method that builds decision trees sequentially and combines them to improve classification performance. The model outputs class probabilities, allowing threshold-based classification. It can handle non-linear interactions between features and is well-suited for tabular datasets.

Dataset Description

The dataset consists of credit card transactions made by European cardholders during September 2013. Transactions were recorded over a two-day period and include a total of 284,807 transactions, of which 492 are labeled as fraudulent. This results in a highly imbalanced dataset, with fraud representing approximately 0.172% of all transactions, or roughly 1 fraud case per 577 normal transactions.

Due to confidentiality constraints, most features are anonymized. The variables V1–V28 are numerical features obtained through Principal Component Analysis (PCA), which removes direct interpretability while preserving variance and structure in the data. These were already transformed in the dataset; we made no changes to them, but we still trained the model on them. The only non-transformed features are Time and Amount. Time represents the number of seconds elapsed between each transaction and the first transaction in the dataset, while Amount represents the transaction value.

Data Preprocessing

Since the PCA-transformed features (V1–V28) are already standardized as part of the PCA process, only the Time and Amount features require additional scaling. These features were normalized using a StandardScaler, which rescales values by subtracting the mean and dividing by the standard deviation. This prevents Time and Amount from dominating the learning process due to scale differences.

The dataset was initially split into 80% training data and 20% test data, using stratification to preserve the original class imbalance in both sets. To prevent overfitting, the model employed early stopping, which internally reserves 10% of the training data as a validation set. As a result, the effective data split becomes:

- 72% training
- 8% validation
- 20% testing

The validation set is used exclusively to monitor performance during training and determine when learning should stop. The model hyperparameters were: learning_rate=0.1, max_depth=6, max_leaf_nodes=31,

min_samples_leaf=20, L2_regularization=1.0, max_iter=300, class_weight='balanced', early_stopping=True (validation fraction = 0.1).

- learning_rate = 0.1: chosen because it's not too small, making it too slow, but also doesn't risk overshooting.
- max_depth = 6, max_leaf_nodes = 31, min_samples_leaf = 20: The default of max_depth is none (unlimited); we changed it to prevent individual trees from becoming too complex, which reduces overfitting.
- L2_regularization = 1.0: This adds a penalty to large weights in the tree; once again, it was changed to help prevent overfitting.
- max_iter = 300 and early_stopping = True: allows the model to train sufficiently while stopping automatically once validation performance plateaus; in this case, training stopped at 41 trees. If it somehow reached more than 300, it would stop, but we knew that it probably wouldn't reach that point because when we did this on the regression set, it didn't even hit 150, so we knew that this smaller and less complex dataset definitely wouldn't reach that point.
- class_weight = 'balanced': We had to change this because the dataset is extremely unbalanced (details stated above).

Model Description

A HistGradientBoostingClassifier was used to perform fraud classification. Conceptually, the model builds a sequence of decision rules that attempt to separate fraudulent and non-fraudulent transactions. Each rule is represented by a small decision tree, and the final prediction is obtained by combining the outputs of many such trees.

The model was trained using binary cross-entropy (log loss) as the optimization objective. This loss function penalizes incorrect predictions more strongly when the model is confident, encouraging reliable probability estimates rather than simple binary decisions.

Handling Class Imbalance

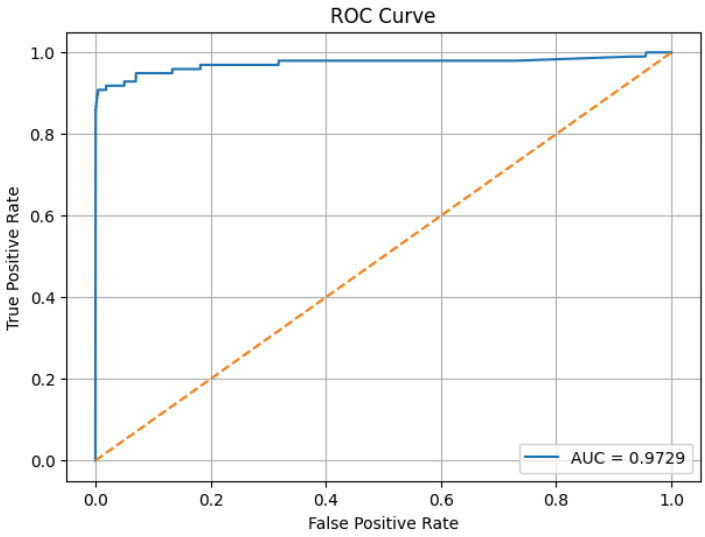
Given the extreme imbalance in the dataset, class weighting was applied during training. This ensures that misclassifying a fraudulent transaction incurs a higher penalty than misclassifying a normal transaction. Without class balancing, the model would be biased toward predicting all transactions as non-fraud, which would yield high apparent accuracy but no meaningful fraud detection capability.

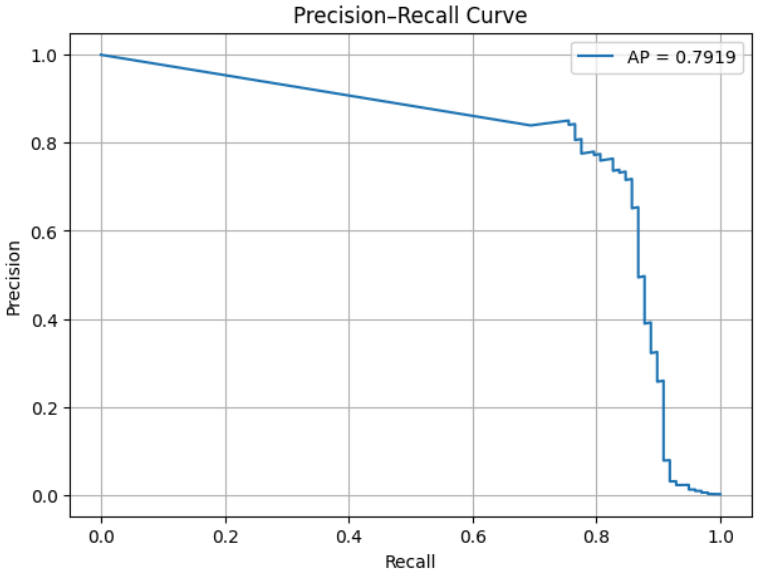
Training Behavior and Early Stopping

The model was allowed to train for up to 300 boosting iterations. However, early stopping automatically halted training once validation performance stopped improving. In this case, training concluded after 41 trees, indicating that additional trees no longer provided meaningful generalization improvements.

Evaluation Metrics

Due to the severe class imbalance, accuracy is not a meaningful evaluation metric. A classifier that always predicts "not fraud" would achieve over 99% accuracy while failing to detect any fraud cases. Instead, model performance was evaluated using ROC-AUC and Precision–Recall AUC (PR-AUC).





ROC-AUC measures the trade-off between the true positive rate and the false positive rate. However, because normal transactions dominate the dataset, ROC-AUC can appear artificially high, as false positives remain rare even for weak classifiers.

PR-AUC focuses exclusively on the fraud class by measuring the trade-off between recall (the proportion of fraud cases detected) and precision (the proportion of flagged transactions that are truly fraudulent). This metric is better suited for highly imbalanced datasets, as it ignores the overwhelming number of true negatives.

The model achieved an Average Precision (AP) of 0.7919, which indicates strong ranking performance. For comparison, a random classifier would achieve an AP approximately equal to the fraud rate, or 0.00172. This demonstrates that the model substantially outperforms random guessing.

Additional Fraud-Only Evaluation

To directly assess fraud detection capability, the model was evaluated on transactions labeled as fraud only. These samples were scaled using the same preprocessing applied during training. Under the chosen classification threshold, the model correctly identified approximately 96% of fraud cases. This value represents recall and does not account for false positives, and therefore should not be interpreted as overall model accuracy.

```
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 %
```

```
import pickle

with open("fraud_model.pkl", "rb") as f:
    saved = pickle.load(f)

model = saved["model"]
scaler = saved["scaler"]
model.get_params()
```

```
{'categorical_features': 'from_dtype',
 'class_weight': 'balanced',
 'early_stopping': True,
 'interaction_cst': None,
 'l2_regularization': 1.0,
 'learning_rate': 0.1,
 'loss': 'log_loss',
 'max_bins': 255,
 'max_depth': 6,
 'max_features': 1.0,
 'max_iter': 300,
 'max_leaf_nodes': 31,
 'min_samples_leaf': 20,
 'monotonic_cst': None,
 'n_iter_no_change': 10,
 'random_state': None,
 'scoring': 'loss',
 'tol': 1e-07,
 'validation_fraction': 0.1,
 'verbose': 0,
 'warm_start': False}
```

Discussion and Conclusion

The results demonstrate that meaningful fraud detection is achievable despite extreme class imbalance. While traditional metrics such as accuracy and ROC-AUC can be misleading in this setting, Precision–Recall analysis reveals strong model performance. The high Average Precision score indicates that fraudulent transactions are consistently ranked ahead of normal transactions, making the model suitable for prioritization and risk-based review.

Class balancing and early stopping were critical components in preventing the model from ignoring the minority class or overfitting the training data. Overall, the model effectively captures fraud-related patterns while maintaining flexibility in operational threshold selection.

Conclusion:

After completing both the regression and classification parts of this project, we're able to see that histogram-based gradient boosting algorithms are flexible and effective for both regression and classification tasks. Our results show how an ensemble of decision trees can be used for many tasks and that the model can be applied to any problem.

We also learned how small value changes can drastically affect training times. Initially, we ran the RandomizeSearchCV with cv set to 10 and n_iterations at 20, and training took more than 30 minutes. After setting n_iterations to 15, we saw training speeds decrease to around 25 minutes.

Overall, these experiments highlight the importance of tailored preprocessing; hyperparameter selection, scaling/processing data, and even choosing the right evaluation metrics, algorithms and cleaning outliers are essential in cutting down training time and lowering error. The histogram-based gradient boosting approach is robust for tabular datasets, capable of handling both regression and classification problems, even under challenging conditions such as outliers or severe class imbalance. This project helped us learn how to use machine learning models in real-world scenarios where data quality, distribution, and imbalance vary across tasks.