Milestone 3 – Feature Engineering + Hyperparameter Tuning

I. Summarize and report key findings

The most important features for the Random Forest Classification aligns with crash factors: whether or not they have prior accidents, if the car has issues, the number of hazard events have they committed, the number of years they have been driving at the company; it is interesting to see that the factor related to speeding or stop sign violation was not completely presented. The model performs very poorly with an AUC of 0.5452 that is slightly above random (0.5), suggesting the model provides minimal discrimination between accidents and non-accidents overall. The AUPRC of 0.00276 is extremely low, reflecting the difficulty of achieving both high precision and high recall on the minority class in such an imbalanced dataset.

The most important predictive features for the Gradient Boosted Tree Classifier also aligned with intuitive crash factors: safety events flagged by Motive, cell phone usage, speed, and number of vehicle inspections. Despite the promising AUC of 0.91, the model's practical utility is limited by the high false positive rate (19.75%) and extremely low positive predictive value (only 0.68% of positive predictions are correct).

Looking back at our project, we have encountered significant data challenges throughout our modeling efforts. The accident records contained substantial gaps, dates were often unreliable, and Motive system adoption is inconsistent across drivers. We hypothesize that drivers who avoided using the Motive system may be more prone to traffic violations, potentially creating bias in our dataset. These issues required extensive manual data cleanup and numerous assumptions.

Rather than pursuing an advanced model with questionable reliability, we recommend taking a more pragmatic approach that delivers immediate value. A dashboard highlighting key risk indicators, such as speeding events, harsh braking patterns, and inspection compliance, would leverage the existing data effectively. Meanwhile, FusionSite should focus on establishing systematic data collection protocols to strengthen the foundation for future modeling efforts . Implementing a simple risk scoring system could guide targeted coaching interventions as we work to improve our data infrastructure. Capturing navigation data, or at a minimum, the origin and destination points for each trip, would enhance the utility of external location data. This would allow the team to incorporate crash statistics and weather conditions along actual routes, rather than relying on broad geographical assumptions.

All in all, the quality of any predictive model or system can only ever be as good as the data that goes into it, so if FusionSite wants to build out their data science capabilities, they first need to focus on the systems to gather consistent, high-quality data or at least be aware of where they have data gaps.

II. Perform necessary feature engineering on your dataset, describe any:

New features introduced since the last milestone"

We performed most of our feature engineering in the last milestone. We engineered features from multiple data sources to build our driver risk prediction model. For the Motive API data, we aggregated driving events by driver and trip date to create features capturing trip counts, days since last trip, and driving history. We transformed safety events into features counting event types (speeding, distractions, etc.) and their severity levels. For vehicle inspections, we calculated metrics like inspections per trip and issues per trip. We also processed idle events to measure efficiency indicators like idle duration and frequency. For location-based data, we established 40-mile service radius zones around each site and calculated statistical measures (mean, median, max, min, IQR) for both crash statistics and precipitation within these zones. We created temporal features using lookback windows at different scales - using 1, 3, and 6-month windows for crash data and 1, 2, and 3-day windows for precipitation data.

In this milestone, we introduced a new variable called has_prev_accidents to flag whether a driver has had any previous accidents. This variable is a binary indicator (0 or 1), created when we join the insurance data to the internal accident data.

Since there aren't many exact date matches between the insurance data and the internal accident data, the join operation uses the closest dates. To ensure accuracy and avoid data leakage with this new variable, we first identify the accidents that we are predicting for each driving event, and only then look for previous accidents. This ensures that future accident data doesn't influence the prediction of past events.

Updated features and specific transformations performed since the last milestone

After compiling all our features, our original dataset had over 300 features. To maintain the integrity of the prediction model, we automatically dropped features corresponding to the current day. This is because we're predicting the likelihood of an accident for a specific day based on historical data only — so, we only use features related to past events.

We had calculated various driving history and location-based statistics using various lookback windows. To avoid having so many columns with redundant information, we selected a single lookback window to use for the moving averages for all the variables except precipitation for which we kept the 1, 2, and 3 day windows under the hypothesis that short-term and medium-term precipitation trends may have different impacts.

We excluded all data before March 1, 2023, since adherence to Motive cameras was quite low early that year and the number of accidents reported in that window was quite low compared to what we have observed with more complete data.

III. Perform hyperparameter tuning on your dataset with the newly engineered features through your base pipeline from milestone #2

We experimented with two types of model: Random Forest and Gradient Boosted Trees. Additionally, we setup the evaluator for Random Forest and GBT to be the same (AUC) to evaluate and compare performance between the 2 models.

Random Forest

Step 1: Hyper Parameter tuning with All Features with Cross Validation

We noticed that since our data is incredibly imbalanced, using the default Spark CrossValidator wouldn't guarantee that each fold contains a balanced number of positive class examples (which we have very few of). Because of that, we decided to implement hyperparameter tuning using a "manual" cross-validation approach. For each set of hyperparameters, the training data is split into train and validation sets, and the model is evaluated using 4-fold cross-validation to ensure consistent performance.
We choose the following hyperparameters to finetune:

- NumTrees `[50, 60, 75, 80, 90, 100, 110, 120]` - allowing exploration for both smaller and more robust ensembles
- MaxDepth `[5, 10, 15]` help control overfitting while capturing non-linear patterns
- MaxBins `[32, 64]` - ensure sufficient granularity for handling continuous and categorical features

The above range are commonly used in practice for fine tuning tree-based models

Step 2: Identify best model
After evaluating each hyperparameter combination using our manual 4-fold cross-validation approach, we identified the best-performing model based on its validation on AUC. This model configuration demonstrated the most consistent performance across folds. Once the best hyperparameters were selected, we will use this to retrain the model using the entire original training dataset (without any internal validation split). This ensures that the model benefits from the full amount of available labeled data

Step 3: Feature Importance Analysis and Filtering

From the best model, we found feature importance and identified 26 features with zero importance and dropped them before retraining the model with full train data

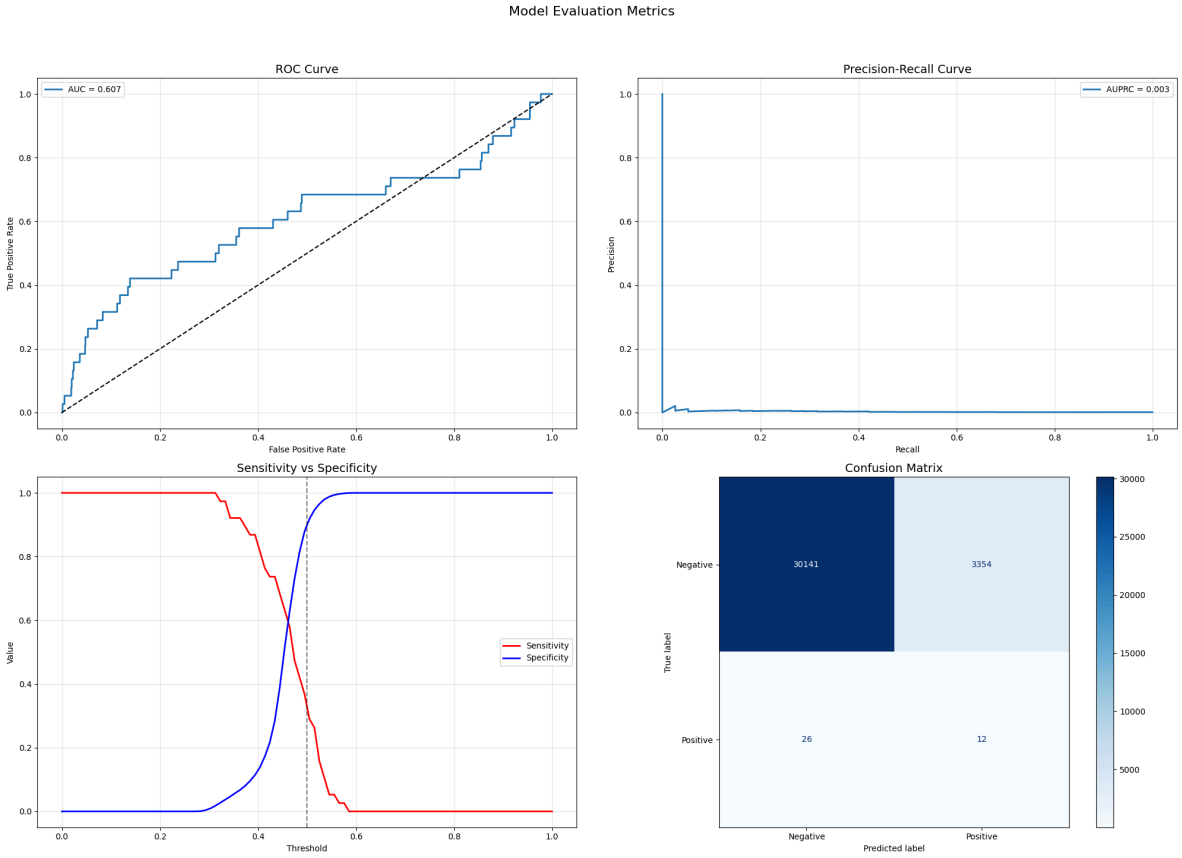
Step 4: Retrain the model with the same above hyper parameters and ensure model performance with valid data
In this step, we do exactly the same as step 1 with a little twist of removing all the zero importance columns. Once we identify the best model from retraining the model, we will do one more round of identifying any trailing zero importance and drop that before our train model

Step 5: Final Model Training
The best model parameters that we are going to use for this step would be:

- Number of tree: 110
- Max Depth 5,
- Max Bin 64

We then use the optimal hyperparameters identified from above, we trained our final model on the complete training dataset with all the data points

Step 6: Holdout Evaluation
Finally, we evaluated our optimized model on a completely separate holdout test set that had not been used in any part of the tuning or training process. This provides an unbiased estimate of how the model will perform on unseen data in production.



The weighted Precision (0.9953) and the Recall (0.8716) are both very high, primarily because the dataset is dominated by the negative class (no accident). In this case of heavily imbalanced, these values are misleadingly optimistic. Conversely, when looking specifically at the positive class, the true positive rate is 0.1667 and positive predictive value is 0.0031. This indicates the model catches only about 16.7% of actual accidents and, among predictions labeled as accidents, just 0.31% are correct.

We have an AUC of 0.5452 is only slightly above random (0.5), suggesting the model provides minimal discrimination between accidents and non-accidents overall.. The AUPRC of 0.00276 is extremely low, reflecting the difficulty of achieving both high precision and high recall on the minority class in such an imbalanced dataset. The AUC value and AUPRC value are pretty similar to when we did the model validation step.

The most important features are has_prev_accidents, total_vehicles_prev_3m_avg, ratio_driver_num_driving_events_roll15d_per_year, years_since_min_trip_date, ratio_driver_sum_minutes_driving_roll15d_per_year, rolling_15day_total_minutes, vehicle_cum_issues, vehicle_rolling_15trip_sum_travel_dist, prev_num_driving_events, total_fatalities_prev_3m_avg, and driver_sum_travel_distance_roll15d.

Gradient Boosted Trees

Step 1: Initial Hyperparameter Tuning with All Features

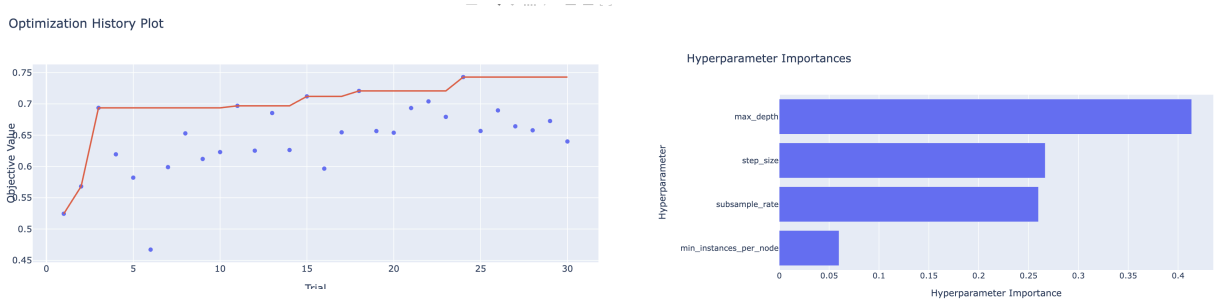
We started hyperparameter tuning using all available features in our dataset. We implemented a stratified train/validation split to maintain the same class distribution in both subsets, which is critical for imbalanced datasets to prevent bias in model evaluation.

We utilized Optuna for GBT classifiers since they can be more prone to overfitting and sensitive to hyperparameters. This package makes use of a Tree-structured Parzen Estimator (TPE), a Bayesian optimization approach that constructs probability models of the objective function using previous evaluations. Unlike random or grid search, TPE adapts its search strategy based on previous trials.

We chose to focus on the following hyperparameters:

- max_depth: Tuning this parameter helps find the right balance between underfitting (too shallow) and overfitting (too deep) specific to our dataset's complexity and signal-to-noise ratio (3,15)
- step_size: Directly affects convergence speed and final model quality, with different datasets requiring significantly different values (0.05, 3)

- subsample_rate: Creates ensemble diversity and helps avoid the model becoming too specialized to training peculiarities (0.5, 1.0)
- min_instances_per_node: Important since our datasets is imbalanced; helps ensure the model doesn't create overly specific rules for uncommon patterns while still capturing meaningful relationships (1,20)



Step 2: Candidate Model Selection with Cross-Validation

After the initial tuning, we identified the 5 best candidate models based on validation performance. To ensure more robust evaluation, we evaluated these candidates using stratified cross-validation, which maintains class distributions across all folds and provides a more reliable estimate of model performance by testing on multiple validation sets.

Step 3: Feature Importance Analysis and Filtering
We analyzed feature importance scores from our best performing models and identified 85 features with zero importance and dropped them as they were adding complexity without any meaningful information.

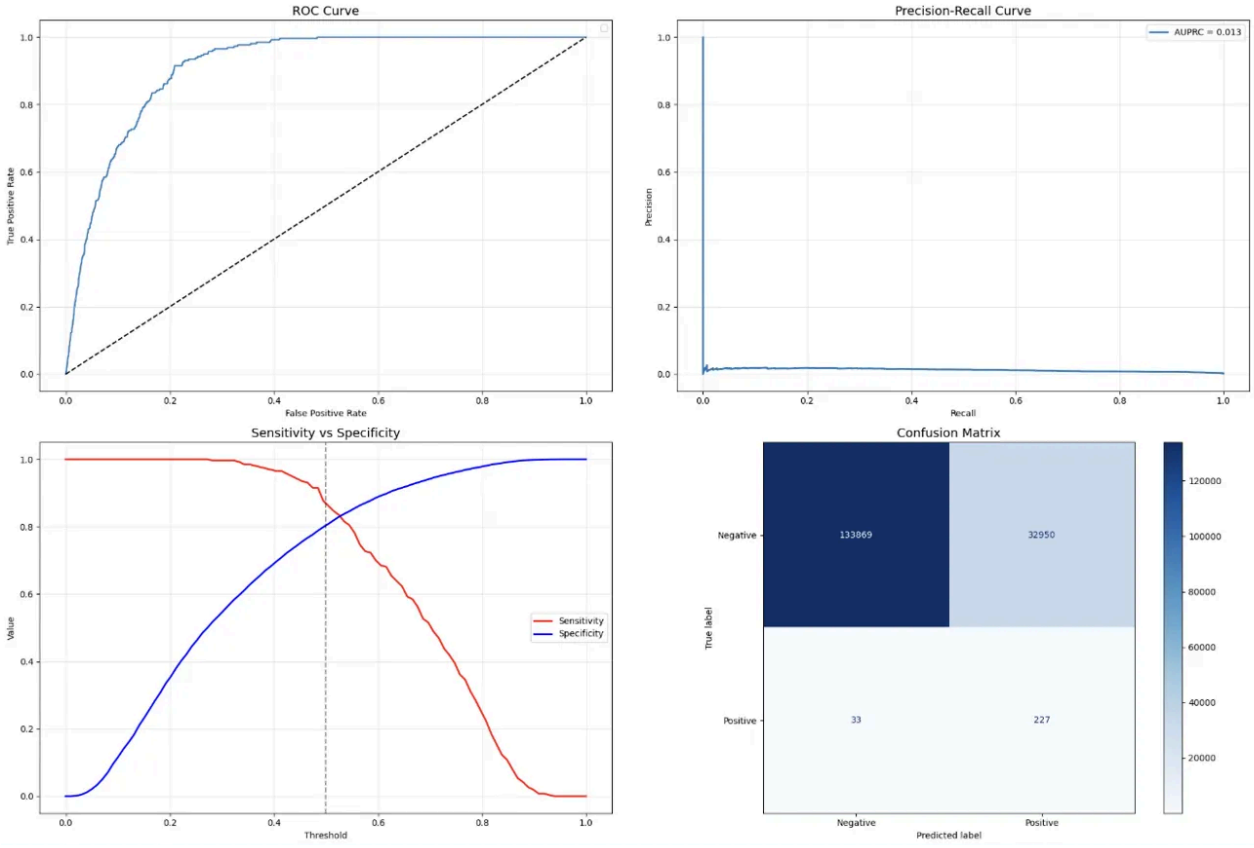
Step 4: Refined Hyperparameter Tuning on Filtered Dataset
With our refined feature set, we conducted a second round of hyperparameter tuning using Optuna. This time, we focused specifically on: step_size, subsample_rate, and min_instances_per_node. We fixed max_depth at 3 since all five of our best performing models from the previous round consistently showed this value to be optimal.

For this round, we implemented stratified cross-validation directly in the tuning process for more robust performance evaluation, rather than using a single validation split as in the initial phase. The hyperparameters for the best model were step_size: 0.26, subsample_rate: 0.90, min_instances_per_node: 12 and it achieved an AUC of 0.68 in 3-fold cross validation.

Step 5: Final Model Training
Using the optimal hyperparameters identified in the second tuning round, we trained our final model on the complete training dataset to leverage all available data for learning.

Step 6: Holdout Evaluation

Finally, we evaluated our optimized model on a completely separate holdout test set that had not been used in any part of the tuning or training process. This provides an unbiased estimate of how the model will perform on unseen data in production.



Our model demonstrates a precision of 0.9982, recall of 0.8026, F1 score of 0.8890, AUC of 0.9110 and AUPRC of 0.0126. Despite the seemingly good AUC, the model has a false positive rate of 0.1975 (meaning nearly 20% of negative cases are incorrectly flagged as positive) and a positive predictive value of only 0.0068. This extremely low PPV means that only 0.68% of positive predictions are actually correct - in other words, for every 1,000 cases our model flags as positive, only about 7 are true positives.

The AUC on the test set was much higher than what we expected from cross-validation. We think this could be due to two reasons. First, the model might have overfit to the validation folds during cross-validation, particularly given the extreme class imbalance observed in our data. As shown in our fold distribution, each fold contained approximately 55,600 examples of class 0 and only 86-87 examples of class 1, creating a significant imbalance ratio of roughly 640:1. With such severe class imbalance, the model likely struggled to generalize properly during

cross-validation, leading to inconsistent performance across folds. Secondly, the training/validation data and the test data might come from different distributions. For example, the test set may contain more representative or easier examples compared to the training data because adherence to the Motive API has increased with driver incentives, resulting in more high-quality complete data.

The most important features in our model are total_events_per_trip_roll15d (safety events flagged by Motive), cell_phone_roll15d (cell phone usage), rolling_15trip_avg_speed_mph (speed), and vehicle_rolling_15trip_num_inspect (number of vehicle inspections), which aligns with what we would expect to be associated with crashes. We have found no evidence of data leakage in the variables, but before something like this is put into production, we would want to run it silently in the background and evaluate its performance in real time.