**Group 2 Detailed Project Report**

Members: Jayden (p2112790), Wee Loon (p2112802), Johnnie (p2007476)

**Introduction**

Following the successful completion of our first sprint, our team held a retrospective meeting to review our progress. During this meeting, we presented our initial phase deliverables to stakeholders, who were impressed with the results of our database, dashboard, and future plans. Both stakeholders and the scrum team are eager to move forward to phase 2, where we will begin our second sprint. Our aim is to incorporate more advanced methodologies such as multivariate outlier detection, model training, and local model deployment into the project. With Jayden leading the way, we will kick-off phase 2 with the creation of the product backlog.

**WEEK 12 & 13**

**Product Backlog**

Jayden created a clear and concise Product Backlog. It includes 8 items ordered based on priority with much detail so that everyone on the team knew exactly what needed to be done. In addition, information on man-hours was included to get a good idea of how long each product would take.

**Final Product Backlog:**

Graphical user interface, text, application, email

Description automatically generated

Text, timeline

Description automatically generated

Jayden created a Gantt Chart indicating our team’s plan for the project progress.

**Data Pre-processing**

Johnnie first implemented the Dask framework when importing data and used appropriate datatypes to save RAM. E.g., changing float64 to float32.

**Data Imputation**

Jayden found out that for most of the columns, there are quite a few null values.

A picture containing text, receipt

Description automatically generated

To handle the null values, we decided to use an iterative imputer with the algorithm set to linear regression. An iterative imputer is a method for handling missing values in a dataset. It works by using a model to estimate the missing values, and then using those estimated values to improve the model's ability to estimate missing values in subsequent iterations. Through using this technique, our imputed values will be more accurate as opposed to using other methods such as simply using a forward fill or backward fill. Also, we concluded that imputing missing accuracy does not make sense, thus, we decided to simply drop rows with missing values for accuracy.

**Outlier Detection on Sensor Data**

We decided that we should test two different outlier removal methods. The first being the univariate outlier detection, using boxplots, and the second being the usage of multivariate outlier detection using Isolation Forest and domain knowledge as well. This results in three different datasets: original dataset (with outliers), inliers from univariate outlier removal technique and inliers from multivariate outlier removal technique. Later, we will train a model on each of these datasets and compare the performances.

Johnnie first used boxplots on the sensor data to identify and remove the outliers. We believe that using domain knowledge for some of the columns will make more sense. For example, there are negative values for bearing and speed, which we know is most probably due to the error in the sensor reading. He then made a function to plot box plot figures and return lower and upper fence outliers. He used a sigma of 12 for the lower and upper fence, values outside of the fence were removed.

For the second method, Johnnie used an isolation forest algorithm on the original dataset (with outliers) and removed 2% of the outliers, keeping only the inliers. This technique assumes that the columns are not independent of each other, whereas the univariate removal technique assumes that the columns are independent.

**Feature Engineering**

Feature engineering can help to create new features that are more informative and better represent the underlying data distribution. This step will help us aggregate the sensor data such that we will be getting a summary of the booking trip. This also allows us to merge the aggregated data with the safety labels data, which contains the target variable. Wee Loon created a function to aggregate the values in the sensor data and merged it with the label and driver data afterwards.

Text

Description automatically generated

For seconds, we simply obtained the max for each booking trip. For the remainder columns, we decided to obtain the mean, minimum, maximum and standard deviation of each column for each trip.

Next, we obtained the 5th quantile, 25th quantile and the 75th quantile. These quantiles can help provide us with useful insights into the distribution of our data.

Lastly, we also obtained kurtosis and skewness for each column. Kurtosis is a measure of the "tailedness" of a probability distribution and skewness is a measure of the asymmetry of a probability distribution. Also, Age column is created from the date of birth for each driver.

This ultimately resulted in our sensor data having a total of 92 columns. These features will capture as much characteristic of the sensor data by extracting as much aggregated information from them.

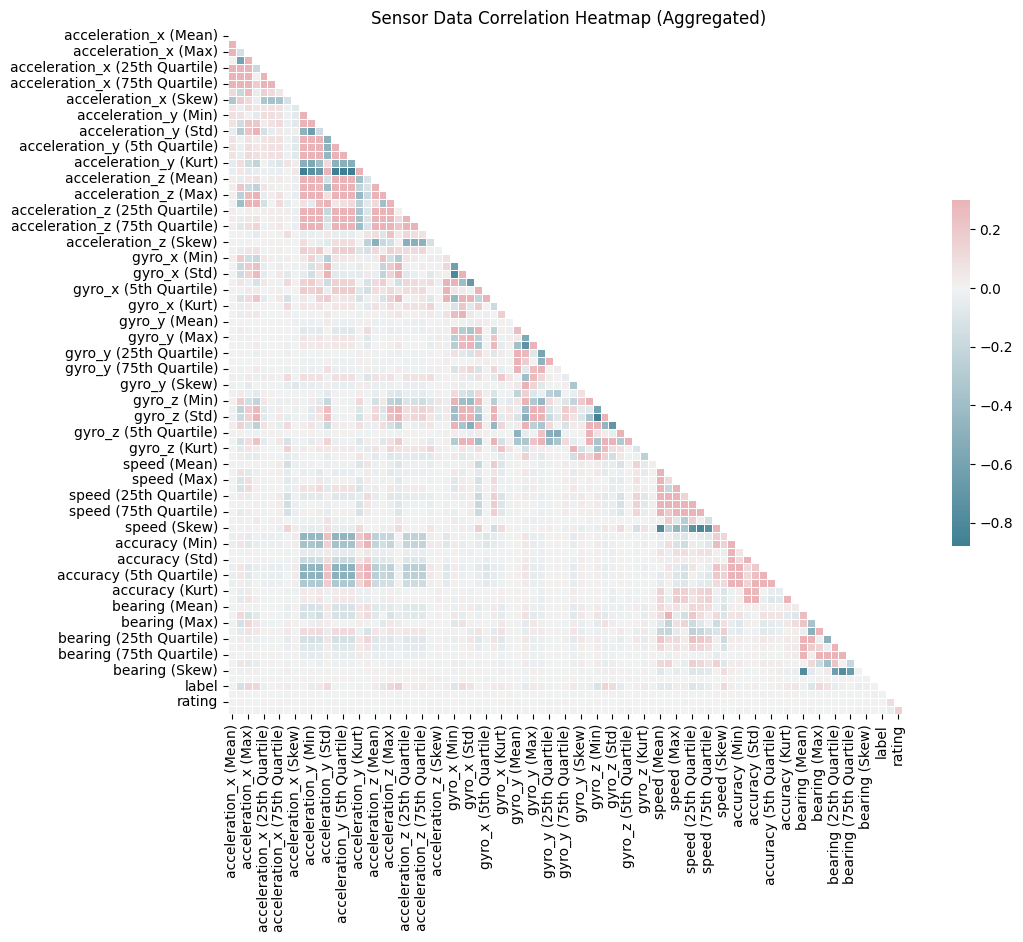
We also added net acceleration which just simply follows the equidistant formula and uses the sine value of bearing since 0 and 360 degrees have no difference but 0/360 and 180 degrees has the highest difference.

**Addressing Multi-Collinearity Issues**

With the drastic increase of columns, there is bound to be highly correlated columns. Johnnie will start to identify columns with correlation coefficient greater than 0.6 and store them in a separate data frame. Identifying columns that have high correlation with one another and removing those columns is extremely important. Correlated columns can cause multi-collinearity, where the columns are providing similar information and can cause instability in the model. Removing correlated columns can help to resolve this issue. Wee Loon created a function that plots the correlations between the existing columns in the sensor data.

A high correlation between two variables is generally considered to be a value greater than 0.7 or less than -0.7. Correlation coefficient of 0.5 or 0.6 can also be considered high correlation.

Next Johnnie looked at the featured engineer data done by Wee Loon, and naturally the correlation increased and can be observed as shown:



Johnnie dropped any columns that has higher correlation than 0.6 starting from the column that have the most correlations with other columns, for example, if net acceleration has a correlation higher than 0.6 with 5 other columns then net acceleration shall be dropped first, this process is iterated until there are no more columns that have a correlation higher than 0.6. This will result in a dataset that only contains columns that has less than 0.6 correlation with one another, we ended up with a dataset that contains 55 columns (from the original 92).

**Tracking Machine Learning Cycles (MLFlow)**

The team has made the decision to adopt MLflow for tracking the progress of our machine learning projects. MLflow will play a crucial role in not just hyperparameter tuning and model training, but throughout the entire process. Whenever there is a need for comparison, we can rely on MLflow. We chose MLflow because it can store valuable information such as hyperparameters, the type of model used, and results.

The MLFlow setup is done on Johnnie’s computer, a virtual environment was made, MLFlow was installed with `pip install mlflow` and ran with `mlflow ui`.

When training is needed to be done the team will train on Johnnie’s MLFlow setup, he would port forward the port 5000 (default MLFlow port) and open port 5000 to public. To connect to MLFlow just simply type <johnnie’s ipaddress>:5000. A better and safer solution is to rent a server. However, the team is too broke ☹.

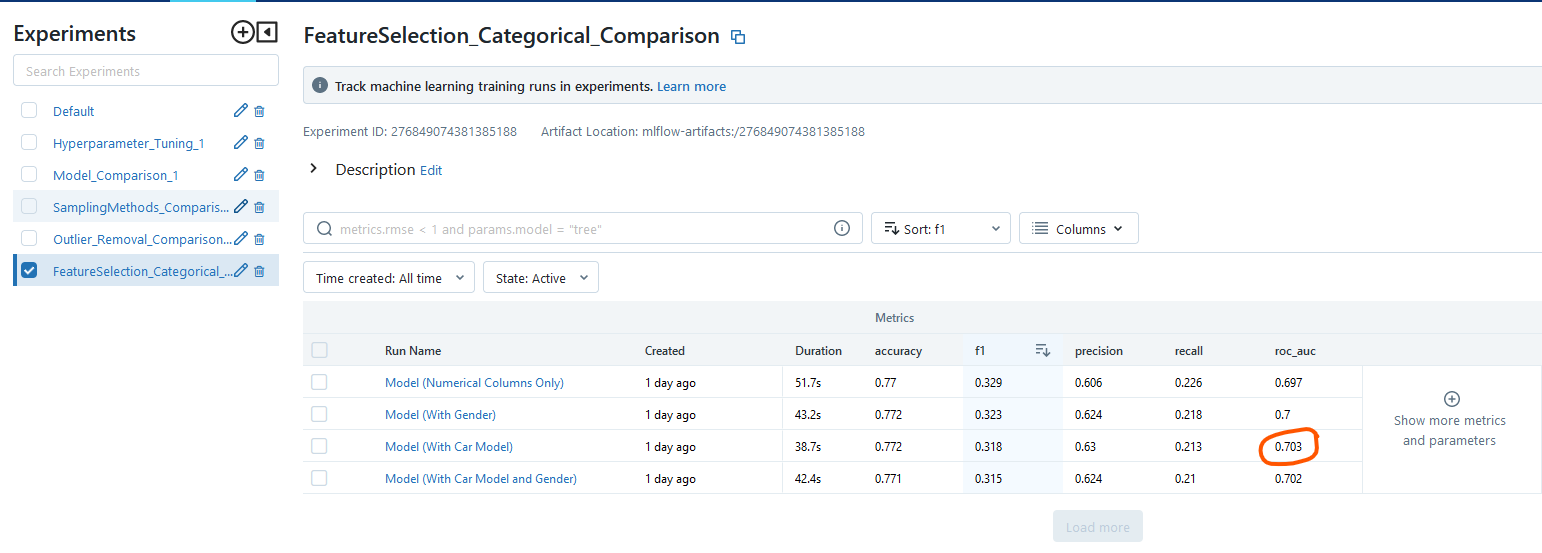
**WEEK 14 & 15**

**Feature Selection (RFECV)**

After Johnnie removed all the highly correlated columns, he used Recursive Feature Elimination with Cross-Validation (RFECV) and the f1-score as the metric to select the most useful features, excluding the categorical columns. After discussing with the team, we determined that the f1-score would be the most objective and unbiased metric for the context of their issue. If we had used accuracy, the ML model might have favoured variables that would always return a "safe" label. If recall was used as the metric, the ML model might have favoured variables that always predict "dangerous." And if precision was used as the metric, the ML model might have favoured variables with a very low recall value since it would have to be more confident in labelling true positives as "dangerous."

Before RFECV, a holdout test set of 30% was set aside and a 4-fold cross-validation was performed on the training set. RFECV returned 5 columns as the most relevant features: "acceleration\_z (Std)," "speed (Std)," "speed (Skew)," "bearing (Std)," and "second (Max)." This suggests that the standard deviation of acceleration, speed, and bearing is more important than the average acceleration and speed when determining if a trip is dangerous or safe. This makes sense because sudden changes in acceleration and speed could be more dangerous than consistent high speeds. High bearing standard deviation may indicate a lot of turns during the ride. The skewness of speed can give insight into the speed distribution and whether a more positive, negative, or neutral speed is related to safe or dangerous driving. The maximum seconds can tell us whether longer or shorter rides are considered dangerous or safe, which could be due to human behaviour.

The categorical columns were added back to the 5 selected columns, and the use of model cars returned a better ROC area under the curve compared to other categorical combinations. The results stored in MLFlow are shown in the image:

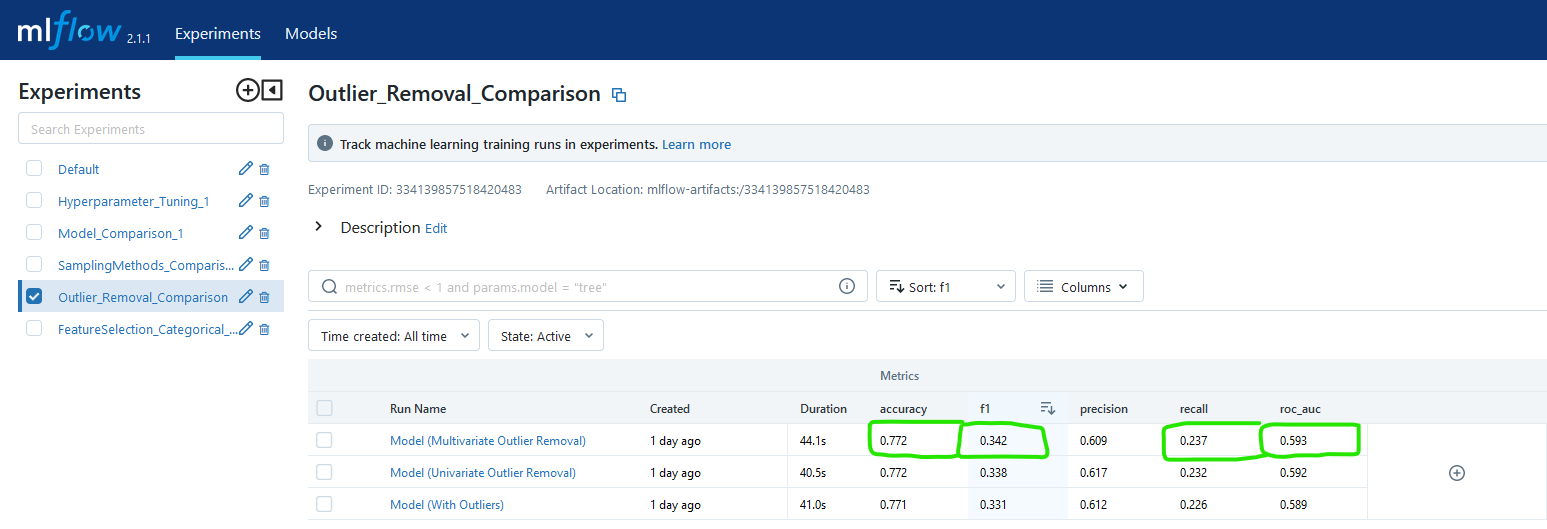


After conducting a feature selection analysis, the team has decided on 6 columns for the ML model. The original 5 selected columns had an ROC area under curve of 0.697, but with the addition of the "car\_model" column, the ROC area under curve improved. The final columns selected are: "acceleration\_z (Std)," "speed (Std)," "speed (Skew)," "bearing (Std)," "second (Max)," and "car\_model."

**Testing Outlier Detection Methods**

With the selected columns determined, the team moved on to testing their outlier detection methods. They implemented two methods: univariate and multivariate outlier detection. To ensure objectivity and prevent data leakage, the experiment was set up using a holdout validation set of which is 30% of the original data. This holdout data will include outliers and was used to replicate real-life unseen data as closely as possible, since real-life data often contains outliers as well (also assuming the sensor used by the taxi driver stays the same).

Three experiments were carried out. First, the team tested the training data with the original outliers. Second, we tested the training data without outliers using the univariate method. Lastly, we tested the training data without outliers but using the multivariate method. To prevent data leakage, all training data was aggregated and split using indices. The results are as shown.



Multivariate outlier removal technique provided the best f1 and ROC area under curve scores compared to other methods. Hence, multivariate outlier removal technique will be used for training.

The below figure shows the confusion matrix of the predictions made:

A picture containing table

Description automatically generated  
Accuracy Score: 0.772, Recall Score: 0.237, Precision Score: 0.609, F1 Score: 0.342,

The rows are the actual classes, and the columns are the predictions made. We can see that for safe trips, the model only made 152 errors in prediction, while classifying 2850 trips correctly.

However, for the dangerous trips, the model made 761 errors while only predicting 237 dangerous trips correctly. This shows that the model is significantly better at classifying safe trips than dangerous trips. A very likely reason is because of the imbalance classes in the dataset. One possible solution is to use sampling methods to balance out the class distribution.

**Testing Sampling Methods**

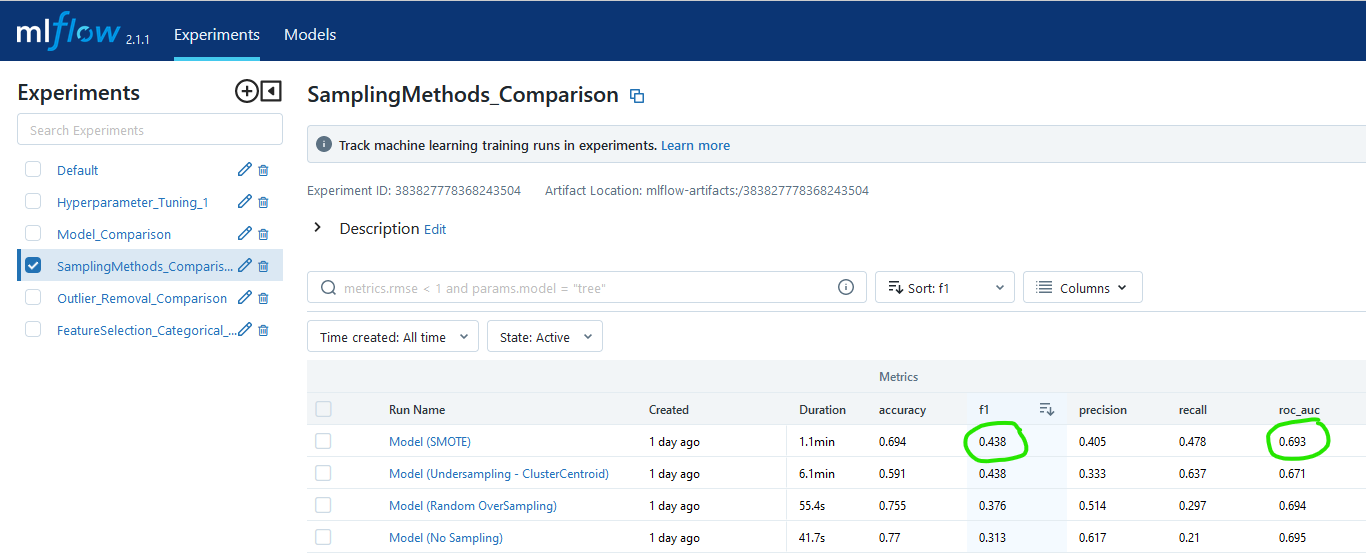
After deciding on the multivariate outlier removal technique, Jayden investigated ways to improve the f1-score and a few methods came to mind. Oversampling, algorithmic oversampling (SMOTE) and under sampling.

Jayden conducted 4 experiments made with the generic pipeline shown:

Diagram

Description automatically generated

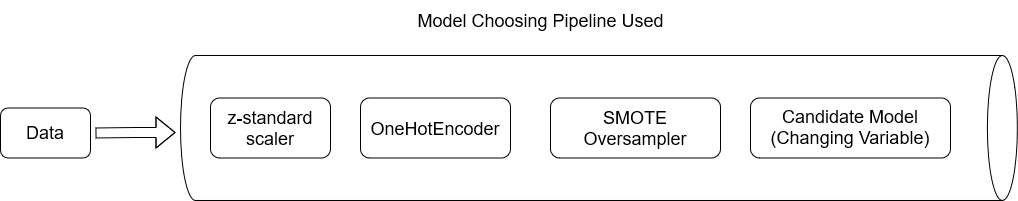
First experiment we did no sampling, second experiment we used under sampling with k means, third experiment we used random over sampling and last experiment we used over sampling with SMOTE k neighbours. All the experiments were done with the original 25% holdout set to be as objective as possible. To prevent any data leakage, the training data is the other 75% of the data but with outliers removed, we did the splitting and shuffling manually with indices. The results are as shown:



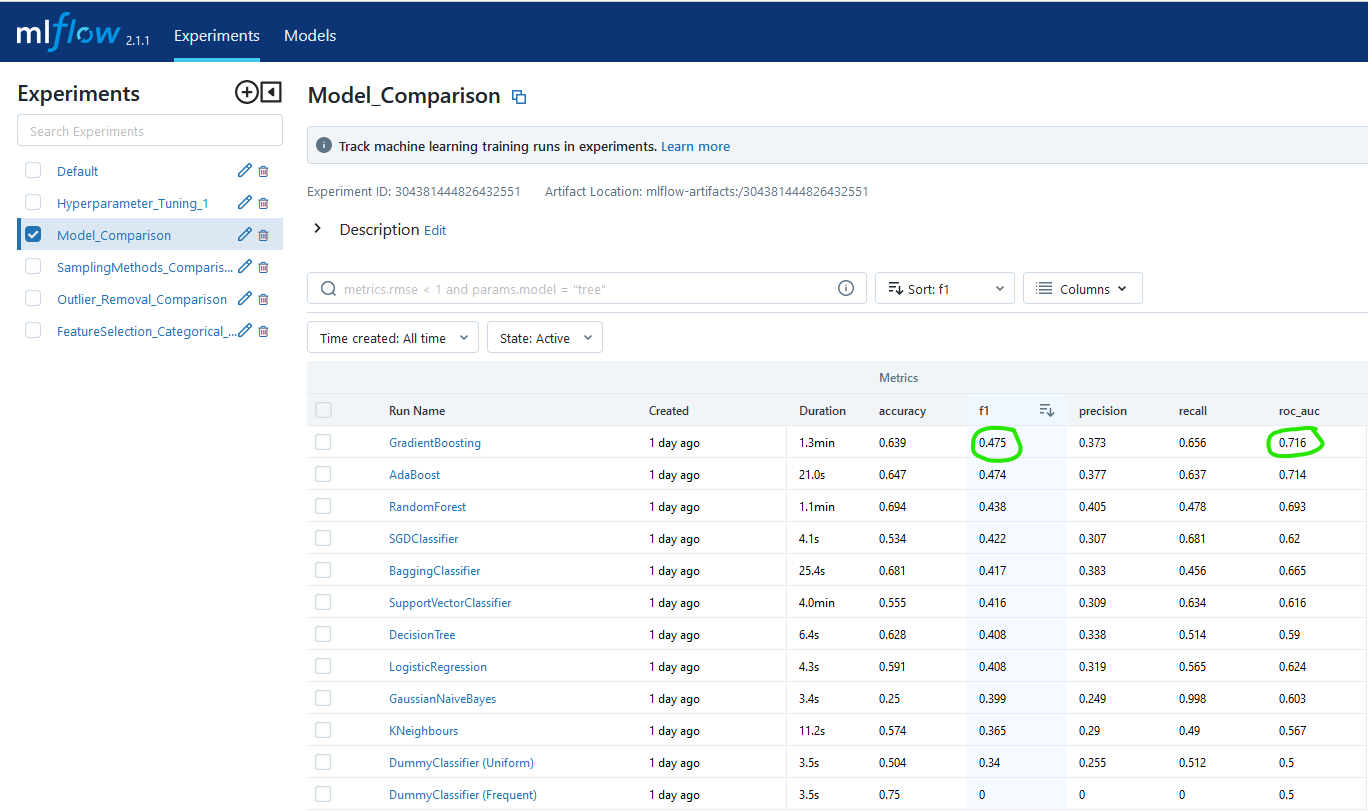
Unsurprisingly, SMOTE with k neighbours provided the best f1-score of 0.438 although accuracy is reduced by 7.2% compared to no sampling. The team evaluated the trade-off and decided that accuracy does not matter much when it comes to the safety of passengers, even if a few drivers or trips are flagged as dangerous even if they are not, it is much better to be safe than sorry. Hence, we will use SMOTE algorithm oversampling as our sampling method as it provided the best f1-score.

**Model Comparison**

After choosing our oversampling method, the team moved on to decide on the algorithm of our model. Wee Loon tested 10 different classifiers, and to decide on which model to use, we chose our model based on the mean of cross-validation f1 score (kfold = 4), ultimately, we will be testing against the holdout 30% original set for analysis since the original set is the closest to real-life data. However, note that we decide our model based on the cross-validation f1-score because we do not want to overfit our model to our 30% holdout original set. Very few or no decisions should be made based on the 30% holdout original data. The pipeline Wee Loon used to train our models can be visualized below.



The results Wee Loon obtained can be observed as such:

****

Two dummy classifiers were used to make comparisons with the performance of the model. Any model that scores an F1-score lower than the dummy classifiers will indicate that their performance is extremely poor and should not be used as our classifier for prediction.

Out of all the classifiers used, the two models that outperformed the rest are:

* GradientBoosting: F1-score of 0.475 and AUC score of 0.716
* AdaBoost: F1-score of 0.474 and AUC score of 0.714.

These F1-scores are better than our previous baseline model (random forest classifier) which produced an F1-score of 0.438.

Both models are boosting algorithms that combine multiple weak models to form a more robust and accurate model. This technique reduces the bias of individual models, which could be a reason why it outperformed the other classifiers.

It seems like GradientBoosting obtained the highest f1-score of 0.475 and the highest AUC score of 0.716. Hence, the team shall use GradientBoosting as the base model when we move on to hyper parameter tuning.

**Hyperparameter Tuning**

Now that we got our base model for hyperparameter tuning, Wee Loon decided on random hyperparameter tuning with cross-validation. Random hyperparameter tuner was chosen since we wanted to keep our model in a pipeline, and we could not use other more complex hyperparameter tuning methods such as hyperband due to incompatibility issues with our pipeline. Nonetheless, this is not a major problem since we are dealing with a relatively small dataset, trying out more trials shouldn’t be an issue.

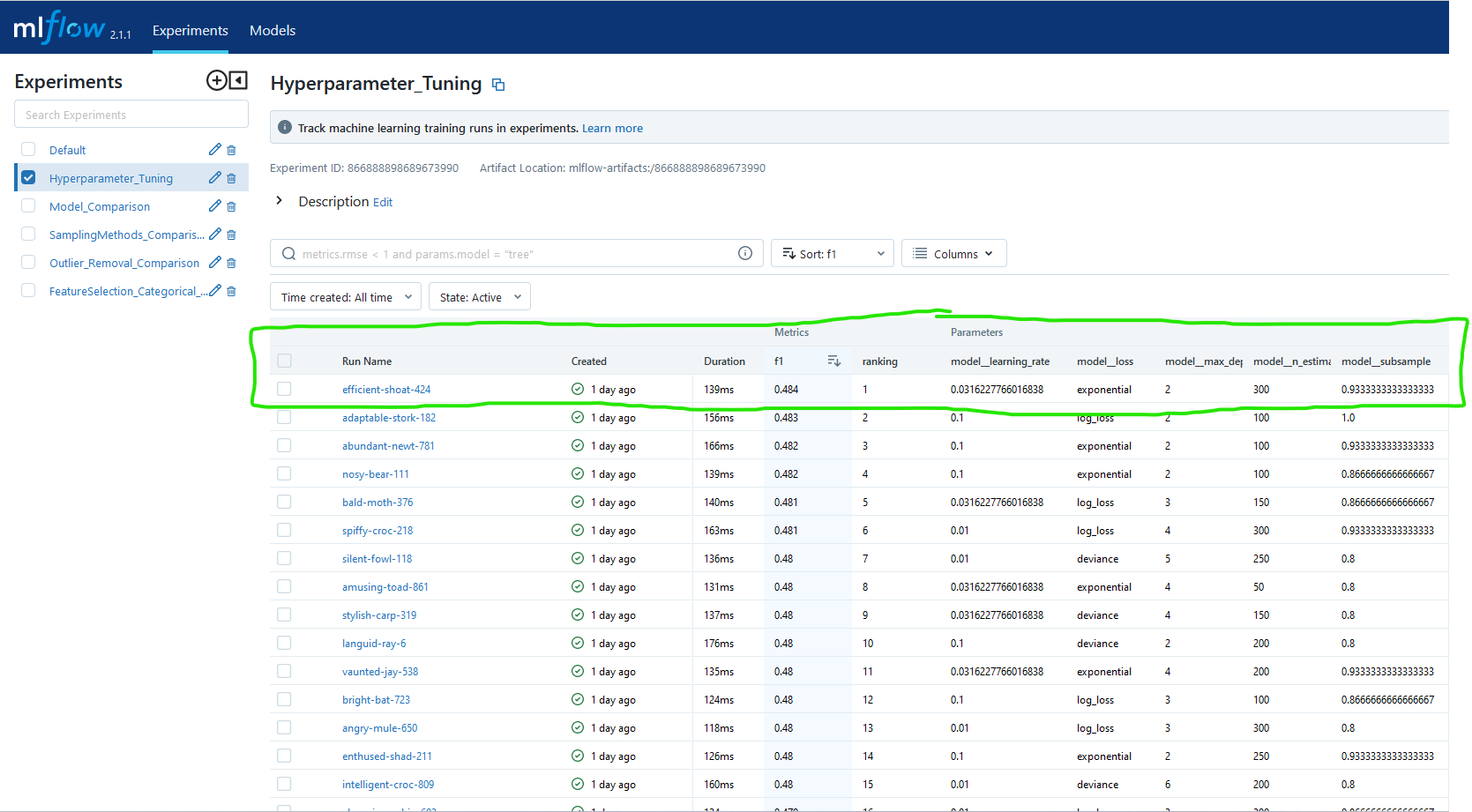
Based on our research, we decided on hyperparameter tuning 5 key hyperparameters. The hyperparameters we are tuning are the following:

* **Learning rate:** Step size at which the model parameters are updated. Too small of a learning rate can lead to slower converge. Too big of a learning rate can overshoot the optimal solution.
* **n-estimators:** Number of decision trees in the model. A larger number of decision trees increases complexity of model, which can lead to overfitting.
* **Max depth:** Maximum depth of each decision tree. Large maximum depth can increase complexity of model, more likely to overfit.
* **Subsample:** Fraction of sample used in each decision tree. Smaller subsamples can lead to a reduction in variance and an increase in bias.
* **Model loss function:** Loss function that the model strives to optimize.

Reference: Jain, A. (2022) Gradient boosting: Hyperparameter tuning python, Analytics Vidhya. Available at: https://www.analyticsvidhya.com/blog/2016/02/complete-guide-parameter-tuning-gradient-boosting-gbm-python/ (Accessed: February 9, 2023).

We felt that these hyperparameters are the most important as these hyperparameters can scale up and down the complexity of our model to prevent over and under fitting by adjusting n-estimators, max depth, and subsample. It also can adjust its training configuration by trying our different learning rate and loss functions. Hence, we chose these 5 hyperparameters to tune.

A cross-validation of 5 was used and the metric to select the best hyperparameters is f1-score. In total there are 1800 possible trials and Wee Loon trained and tried 100 trials, covering about 6% of all possible trials, the best hyperparameter can be observed as shown.

****

After hyperparameter tuning, one of the models produced a highest F1 score of 0.484, which is an improvement from before tuning, which only had an F1 score of 0.475. The hyperparameters for the best model is as follows:

* **Learning rate:** 0.03162, a smaller learning rate than default (0.1), which can indicate that it was able to converge to a more accurate solution but required more iteration to do so.
* **n-estimators:** 300, a larger number of decision trees than default (100), indicating that the model was able to capture more complex patterns in the training data.
* **Max depth:** 2, smaller max depth than default (3), indicates that reducing max depth helped reduce overfitting and improve generalization performance.
* **Subsample:** 0.93333, smaller subsample than default (1), indicates that reducing sample size helped reduced overfitting and improve generalization performance.
* **Model loss function:** Exponential loss, this loss function is more sensitive to errors made by individual trees, which leads to a more robust and accurate model.

**Training Unsupervised Machine Learning**

The team decided it was worth putting some manpower into training and analysing an unsupervised learning model for clustering the data. This will enhance our GUI by providing more insights to the users’ predictions, seeing which cluster our model thinks they belong to, and from there, telling the user more insights about how their data relates to the other drivers in the company.

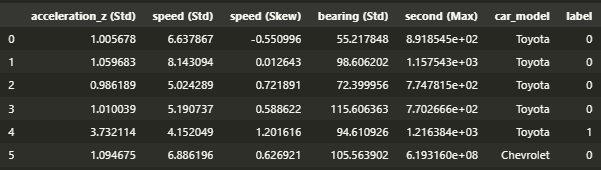
This was done by using the KPrototypes, it is a hybrid clustering algorithm that can process both categorical data and numerical data. We ran the algorithm multiple times to first find out what the best number of clusters was. We made use of an elbow graph to find the best number.

Chart, line chart

Description automatically generated

**The optimal number of clusters is 6.**

To analyse and interpret the clusters we can get the Centroids for each cluster (0 to 5) and view their descriptive statistics.

Text, table, calendar

Description automatically generated with medium confidence







Note that the normal observed distribution for safe and dangerous trips is 25% to 75%:

So, from the information above we can start interpreting the clusters:

- Cluster 1 & 4 has higher than 25% dangerous drivers (observed average dangerous trip).

- Cluster 4 contains 73% dangerous drivers, which is way higher than the observed average dangerous trip of 25%, which means cluster 4 can mean that it contains dangerous drivers. Therefore, we should warn drivers to that are frequently getting trips labelled as 4. Seems like the main cause for its high percentage of dangerous label could be attributed to accelerating (std), bearing (std) & speed (skewness), which makes sense, people should not like violent trips...where you speed up very quickly (acceleration) or turn a lot (bearing)

-Cluster 1 contains 30% dangerous drivers, which could mean these are drivers on the verge of being labelled as dangerous. Cluster 1 also have its centroid distant closest to Cluster 4's centroid. If the trip is labelled as Cluster 1, we can warn the driver to drive more safely before being flagged as a dangerous driver in the future.

- Cluster 0 has the highest safety label at 86% and looking at the attributes it can tell us the characteristics of safe drivers. Seems like safer drivers have lower acceleration standard deviation and speed skewness, which means these drivers are more careful. Drivers frequently labelled as cluster 0 can be given a praise to let them know they are doing good.

- Cluster 5 are clearly outliers, and looking at the values, these are trips where the drivers that clearly driving dangerously especially with the high acceleration (std), speed (skewness) & bearing (std) but these trips are still labelled as safe. This could happen when the client is perhaps nice or in a hurry. Drivers that are labelled as Cluster 5 will also get the same warning as Cluster 1, where we will warn the driver to drive more safely otherwise they will be labelled as dangerous drivers. Another analysis would be since there is only 3/20000 datapoints in this cluster it could also mean the sensor of drivers in Cluster 5 are faulty.

- Cluster 2 & 3 are characteristics of the average driver. They will not get a warning.

After this analysis , we perform PCA on the clustered data and end up with this scatterplot (Note Cluster 5 are outliers so they are not in the plot)

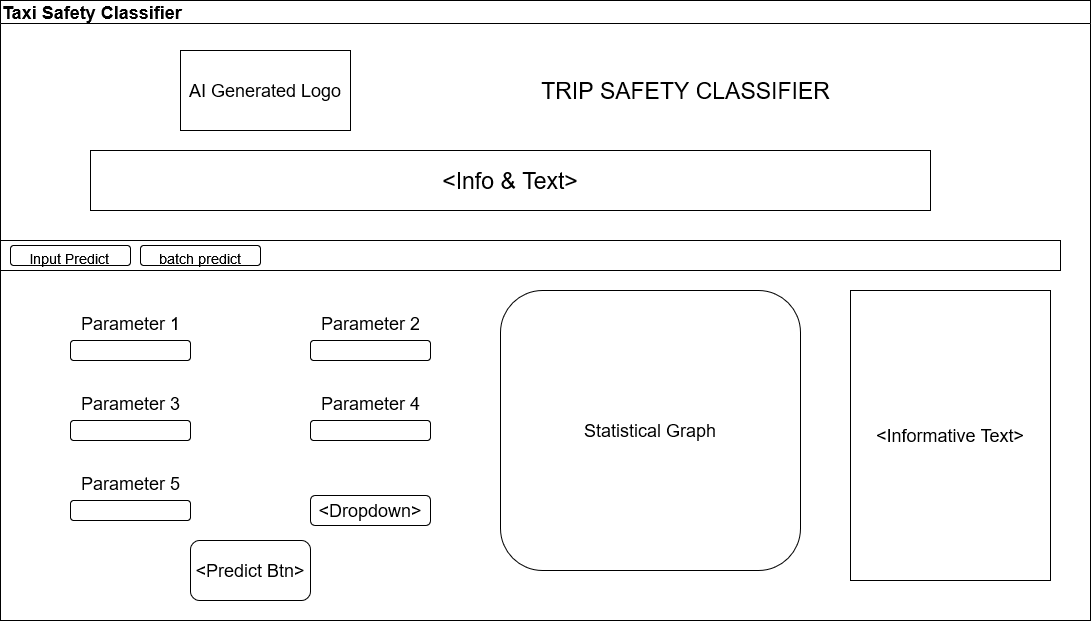
Chart, scatter chart

Description automatically generated

**WEEK 16 & 17**

**Tkinter Application**

After training all our models (both supervised and unsupervised learning models), Jayden started working on our wireframe that all the team can agree on. The wireframe is seen as shown.



We got to work on using Tkinter to create our GUI. We made use of various TTK (Themed tkinter) widgets following a green forest theme. These include Labels for showing text messages, the TTK Notebook for having different tabs, the TTK Entry widgets for user input data, as well as Buttons like for Predicting. We made use of the Tkinter grid system to allow for dynamic resizing of the application to suit the user’s current need/screen. We created functions for predictions where it predicts with the trained model for the safe/dangerous label, then predicts on the unsupervised learning clustering model, then outputs the corresponding PCA graph showing which cluster it belongs to.

**Form Input Page**

Chart

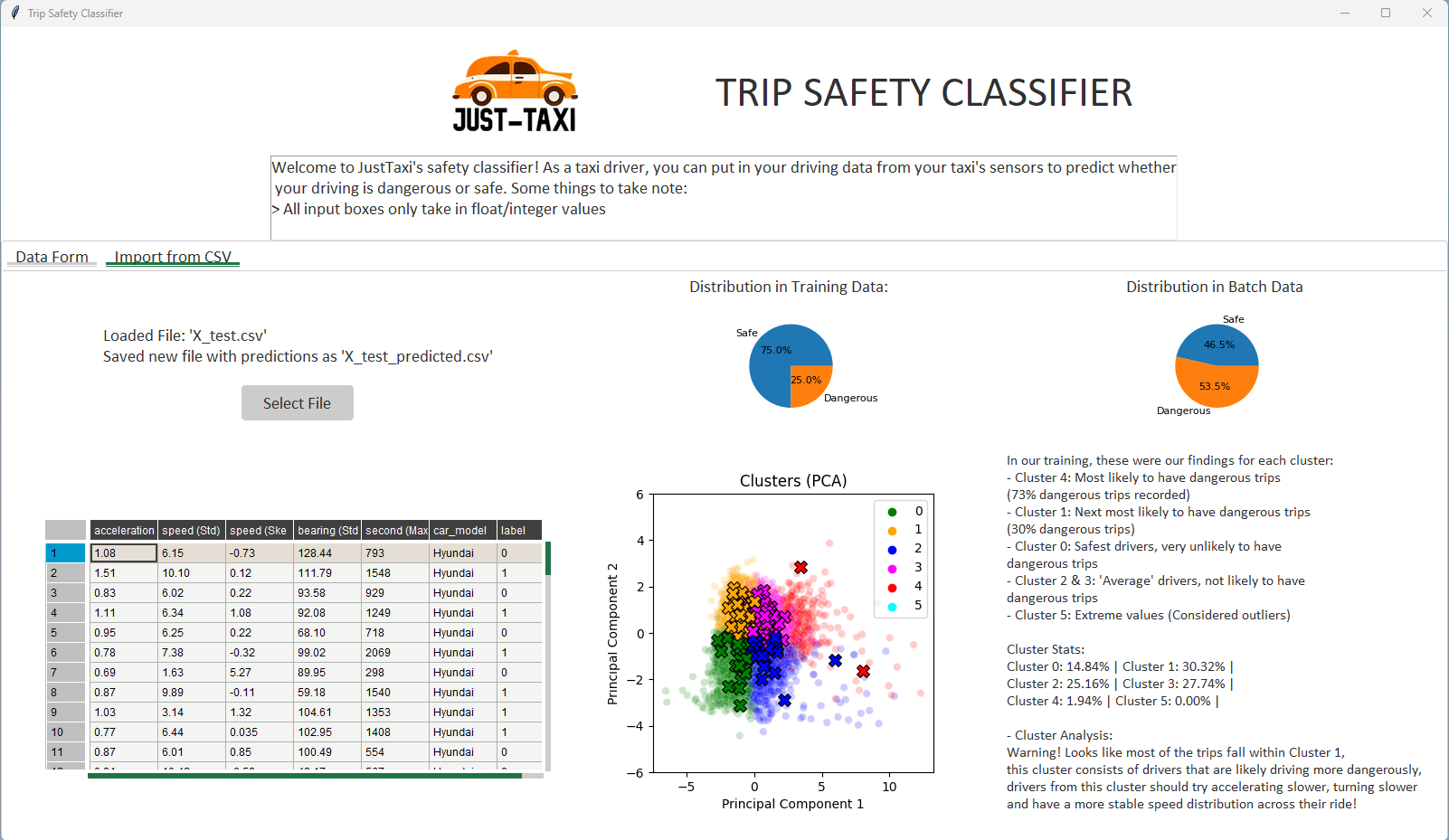
Description automatically generated

The user must first fill up the data form , we assume that this data is readily available from the backend from the taxi car sensors. There is input validation to inform the user if they try to predict with missing data/invalid data type(strings)/inputs out of range (ridiculous numbers).

After prediction, the user will see the Binary prediction result below the predict button. Then the user will see the PCA graph with their new prediction being marked with a cross along with the colour of their predicted cluster. On the PCA graph, the original data is the data used for training and the alpha is lower so that the new prediction is more visible.

In addition, information about every cluster is displayed as well as the information on the individual PC loadings. This is to provide additional context to the user about what all this means.

**CSV File Input Page**

****

This next feature allows the user to do batch-wise predictions.

The user first uploads a CSV file or a text file with comma separated values, if not there will be an error message displayed. **The column names should be same as the example we give in the data folder X\_test.csv**. The order can be shuffled around but best to stick to the format.

After prediction, the user will see a Pandas Table view of the dataframe , it contains the new column labels with the Binary prediction results. This dataframe is saved as a new csv for the user to retrieve the predictions. The user also sees 2 pie charts comparing distribution of safe/dangerous trips between training data and imported batch data. The user will see a similar PCA graph from earlier with their new prediction(s) being marked with a cross along with the color of their predicted cluster. In addition we display the same information about the clusters to provide additional context to the user about what all this means. Furthermore, based on the clustering predictions, our application will give a conclusion about how safe/ unsafe the driving is based on the cluster with the most predictions from the batch.

**Conclusion**

Graphical user interface, application

Description automatically generatedChart

Description automatically generated

In conclusion, our team first applied advanced data preprocessing on the sensor data. This includes Data imputation, feature engineering, feature selection, data aggregation and outlier detection. Then we conducted modelling on the data, using MLFlow to track our experiments. In the end of the experiments, we selected Gradient Boosting as our model. We performed hyperparameter tuning on this model and had it as our final model. After that we created a GUI to deploy our ML model as an executable file. We went through testing to refine the app and our project was concluded.

**Project Documentation** **End (Phase 2)**

References:

1. <https://github.com/rdbende/Forest-ttk-theme> TTK Theme used.