Traffic Congestion

Shi Hong Yi[1], Ong Qi Ren[2], Low Ming Yuan[3] and Loh Jie Hao[4]

[1] Multimedia University, 1211108962@student.mmu.edu.my
[2] Multimedia University, 1211108357@student.mmu.edy.my
[3] Multimedia University, 1211111801@student.mmu.edy.my
[4] Multimedia University, 1211108262@student.mmu.edu.my

Abstract. Traffic congestion in urban centers has negative implications for increases in travel time, fuel consumption and environmental pollution which can limit the effectiveness of the intelligent uses of technology that smart cities can deliver. This project proposes a machine learning-based method for anticipating and controlling traffic jams in smart cities, meaning from our previous projects we can leverage the Huawei Mind Spore framework to develop our solution more effectively. We can collect, analyzing, real-time and historical public dataset of traffic data to create predictive models for traffic congestion prediction, such as Random Forest, Support Vector Machine (SVM), and Long Short-Term Memory (LSTM) networks. Accuracy, precision, recall, and the F1 Score metric were used to train and assess the models. Prediction accuracy is further improved by feature engineering to include traffic volume, weather variances and time-based variables. The proposed solution provides direct integration into smart city infrastructures so that intelligent congestion alerts can be provided and delivery optimized traffic flows using adaptive traffic signal control. This provides social benefits such as reduced commuting time, and potential commercial value for scalability if adopted and deployed in Urban Planning. The project also investigates how data can be processed by and used with the cloud computing processes and offerings available from Huawei so data processing for streamlined model deployment can also be achieved and broadened for future work enhancing model scalability and proposition relevance in real-world settings. Limitations associated with data quality, limited computational resources, and lack of assurance during model training showed some qualities or areas for improvement and exploration to achieve more broadly accepted eligibility for smart cities.

Keywords: Traffic Congestion, Machine Learning, Smart Cities, Huawei MindSpore, Predictive Modelling

.

1 Introduction

One of the biggest issues in cities is traffic congestion. Traffic congestion increases travel time, fuel consumption, harmful air pollutants, and decreases quality of life. With cities transitioning towards smart cities and utilizing new technologies like machine learning (ML), it will be important to improve mobility, sustainability, and efficiency. This project is focused on the Traffic and Transportation domain and aims to develop a ML-based solution aimed at predicting and addressing traffic congestion. The project proposes utilizing the Huawei's MindSpore ML framework and publicly available traffic datasets were possible, to develop a solution to forecast traffic congestion patterns and develop more adaptive traffic control systems. The proposed solution considers real-time external factors such as sense traffic volume and weather; along with time-based variables to optimize traffic flow and commuter experience. The project also aligns with the smart city objectives to provide effective infrastructure for urban living while delivering a solution that can be relatively easily deployed as a SaaS model utilizing Huawei's AI+Cloud technologies. The project promises to offer social and market value in understanding dynamic societies, while improving urban planning and quality of life for citizens.

2 Problem Statement and Objectives

2.1 Identifying Gaps in Literature and Justifying their Importance

Traffic congestion prediction topics tend to report on traditional statistical methods or base machine learning models like linear regression or base decision trees that may have limitations effectively applying to the complex, dynamic or real-time traffic affected by weather, accidents, special events, etc. There's also limited literature on the use of integrating multiple. Deep learning and ensemble models are examples of machine-learning models. that may enhance predictive robustness regarding adapting to changing traffic conditions. Gaps in the literature is problematic since it limits our ability to create viable scalable solutions to real-world urban traffic issues relevant to commuter efficiency, environmental sustainability, and economic productivity.

2.2 Problem Statement

The time spent on traffic jams on urban roads leads to the failure to travel as well as high fuel consumption and pollution that are a source of pressure on smart city infrastructure. The present-day traffic control systems employ either static traffic management or reactive traffic management and as a result, they are only able to predict how traffic will flow or be able to respond in real time. The scarcity of integrated/scalable use cases of ML limits the capabilities of cities to manage traffic patterns and enhance mobility in the city.

2.3 Objectives

- Create and train a machine learning model that provides accurate predictions of traffic congestion using publicly accessible datasets as well as real-time data inputs (incl. traffic count data, weather, and time of day characteristics).

- Implement, and compare at least 3 types of ML approaches (Random Forest, SVM, LSTM) to determine which approach might provide the best performance for traffic congestion predictions.

- Integrate Huawei's MindSpore framework to enable effective model training and scalable deployment in the context of smart cities.

- Propose a traffic management solution that leverages such models to optimize traffic signals and traffic congestion conditions while also contributing to the commutation experience for commuters and sustainable objectives.

3 Related Works

The usage of machine learning (ML) within traffic and transportation systems been the focus of the major research effort in existing literature, and multiple studies have provided literature reviews for researching traffic congestion specifically. In this section, literature syntheses will be summarized from the core document provided here. This literature synthesis will focus on studies relevant to predicting and managing traffic congestion in smart cities and will draw from the systematic literature review by Shi et al. (2025).

Behrooz et al. (2022) completed a review of the application of ML in smart transport, and reported the topics of traffic flow prediction and signal control. The models they identified include neural networks, support vector machines (SVMs), decision trees, and implementations of prediction for traffic congestion utilizing live data from traffic cameras and sensors. Behrooz et al. (2022) concluded ML is effective in improving current traffic systems in cities; however, the accuracy of traffic data/reduction of false data and the implementation of ML models in practice, such as real world verification, remain significant challenges for potential solutions to predicting and addressing congestion at scale.

Alqudah et al. (2020) investigated deep learning (DL) in the domain of environmental transportation, and specifically focused the analysis to short-term forecasting of traffic flow. Alqudah et al. (2020) analyzed Deep Learning (DL) methods (as well as Machine Learning methods) such as convolutional neural networks (CNN), recurrent neural networks (RNNs), Long Short-Term Memory (LSTM), and Gated Recurrent Units (GRUs), the study exposed that DL models were very effective in modeling complex traffic data patterns. Alqudah et al. (2020) recommend a consideration of the options of DL, while acknowledging some challenges of significantly larger data requirements and available computing power that will impact whether is feasible to have implemented specific models in real-time for smart cities.

Kumar et al. (2021) gave a sweeping summation of AI in transportation, encompassing traffic flow methods, accident detection, and driver behavior modelling. They did differentiate ML applications into supervision learning, unsupervised, and reinforcement learning. They noted the value of ML combined with IoT for ease of congestion control. They also identified gaps in standard datasets especially useful in ML and model explainability to develop practical applications.

Pravin et al. (2025) considered the use of ML with smart city platforms (they emphasized routing and modelling traffic light control), and focused on big data and cloud computing to facilitate ML. Pravin et al. (2025) emphasized collaboration among all stakeholders, such as government, industry and researchers to address the issue of privacy and scalability. The contributions in this paper highlight the necessity for applying system-level architectures of the possibilities for applying congestion solutions.

Hammoumi et al. (2025) incorporated ML by predicting traffic jams in Casablanca, Morocco. This paper followed a physical study with actual datasets and demonstrated better accuracies for ML models (e.g. LSTMs) for predicting congestion. Some challenges were to generalize the models to other urban contexts, because of different traffic patterns across cities.

According to Gupta et al. (2025), ML-based approaches to optimizing traffic flow have great potential for managing congestion through efficient use the available real-time sensor and GPS Data. They highlight the critical obligations, if accepted, to consider scalable models in urban settings along the lines of smart cities.

There are extremes of motion in the work that have been considered together, in when it comes to the performance of ML models, for example LSTM, SVM, and reinforcement learning (e.g., DQN/PPO), with reported congestion detection accuracy of up to 90.9% when using spatiotemporal models. However, there are extremes in model reproducibility, standardization of data, and real-time execution on resource-limited devices. Ultimately, the outcomes of this study will inform our project as it underscores the importance of robust and scalable ML implementations with the Huawei MindSpore framework in order to manage city congestion in smart cities.

4 Dataset & Pre-processing

4.1 Dataset Collection

The dataset used in this project, Traffic.csv, came from a reliable public repository, Kaggle, in keeping with the instructions for choosing a dataset for the project. It contains transportation data on traffic frequencies that were collected once every 15 minutes taking into account different types of variables affecting traffic-related situations. The Grid dataset is made up of 2976 records and 8 columns. Temporal variables include Time, Date, and Day of the week. Apart from the temporal features, each row of data references the vehicle frequencies included (CarCount, BikeCount, BusCount, TruckCount, Total) –among others. The 'Traffic Situation' column, as the name says, includes the traffic situation in four different categories (low, normal, high, and heavy) which was later transformed into a binary classification problem to predict severe traffic situations (1 = high or heavy, 0 = low or normal) to serve the purpose of prediction of traffic situations that were prominent in risk of being an actual accident.

The Grid dataset match closely with the domain Smart Cities for Traffic & Transportation because of the well-documented datatypes recorded that enable an understanding of the events of a number of persistent traffic phases throughout days of the week that a A machine learning model need to have the capacity to be developed, with the dataset to predict traffic congestion and the possibility of a situation leading to an accident. The understanding of each of the temporal aspects and each of the vehicle types will also allow for more creative feature engineering to improve upon model performance.

4.2 Data Cleaning

Data preprocessing proved a vital operation to verify that the dataset was appropriate to have trained the logistic regression model. And the procedures taken to clean the data are as follows:

Handling Missing Values:

- The dataset was examined to see if there were missing values using pandas isna().sum() which didn’t return any missing values there as a result we had complete data to move forward.

- As a precaution to make sure any possibility of a missing value in the model were covered, a solid imputation process was introduced in the script: numeric columns (CarCount, BikeCount, BusCout, TruckCount, Total) were always set to be filled with their median values, and the categorical column (Day of the week) with its mode. Essentially all future models would have an imputation prepared for any future missing values data.

Detect and Handle Outliers:

- This code detects and handle the outliers inside all numeric columns of the Data Frame using Interquartile Range (IQR) method. It first selects numeric columns and calculates 25th (Q1) and 75th (Q3) percentiles for each. And the IQR is the values outside of the range [Q1 - 1.5*IQR, Q3 + 1.5*IQR], as well as the difference between Q3 and Q1 are usually considered outliers. Detected outliers are then replaced with the nearest boundary (lower or upper). The script prints the number of outliers found for each feature and the new value range after handling. If no outliers are detected, it states so and moves to the next column.

Target Variable Transformation:

- The Traffic Situation column was transformed to create a binary target variable to standardize with terminating the logistic regression model. Instances of the column traffic situation labelled high or heavy were congruently given a target value of 1(severely trapped in traffic) or low or normal a 0. This makes the classification process simpler and helps with modelling to predict high risk traffic scenarios.

Removing Duplicate Features:

- The Total column, which totals CarCount, BikeCount, BusCount, and TruckCount, was checked for multicollinearity. After checking correlation, Total had a correlation to CarCount, BikeCount, BusCount, and TruckCount above 0.95, producing multicollinearity. Therefore, Total was removed to avoid affecting the logistic regression coefficients because of unstablility due to multicollinearity.

Encoding the categorical features:

- One of the categorical features was the Day of the Week section. The characteristics include the following: Monday, Tuesday, Wednesday, Thursday, Friday, and Day of the Week The day of the week, Saturday Sunday was produced by using pandas.get_dummies() to one-hot encode the categorical feature Day of the week.The features are all binary and this must be done for the logistic regression algorithm that needs the variables to be numeric.

- The Time column was extracted as a time in 15 minute intervals. In order to have a new continuous numeric feature from Time, I extracted hours and minutes and created a new feature which is simply the distance in hours since midnight. The original temporal order can still be preserved by being in hours, and which also makes for a better modeling feature.

- The Date column feature was removed because when comparing overall traffic by the hour, I informed that the Date offered little to no value in explaining daily traffic patterns, particularly in relationship with Day of the week feature, and lastly a Time feature.

4.3 Feature Selection

One of the major areas of our efforts was feature selection to help improve model fit and reduce overfitting. We performed the following actions:

Correlation-based Feature Selection:

- We created a correlation matrix for the various numerical features (CarCount, BikeCount, BusCount, TruckCount) to look at brokered features that were highly correlated. We noted earlier that we removed the Total column, as it was highly correlated with all the other vehicle counts, allowing us to reduce redundancy and noise to highten stability in the model.

Feature Importance:

- After our model training, we conducted analysis of the feature importance of the coefficients associated to the logistic regression model. After looking into the coefficient values of different lengths and scales, we concluded that the featured with the highest importance was likely CarCount followed closely by the BusCount and Time. This exercise confirmed that our final features were relevant in predicting our outcome based on importance.

Temporal Feature Engineering:

- The Time column was transformed into a continuous variable (hours since midnight), which may better allow for capturing the temporal patterns associated with traffic congestion. Day of the week was also one-hot encoded to incorporate differences in traffic behavior during the week (e.g. more congestion on weekdays than on the weekends).

Feature Scaling:

- All numerical features (CarCount, BikeCount, BusCount, TruckCount, and the engineered Time feature) were then scaled using StandardScaler from scikit-learn. This means the data for these distributions have been standardized to mean = 0, and standard deviation = 1. Scaling features is important because standardizing features with different scales (e.g. vehicle counts vs. time) allows them to contribute equally in the logistic regression model, which will be advantageous in model convergence and performance.

4.4 Summary of Pre-processed Dataset

After Preprocessing, the dataset was converted to a feature matrix that looks like this:

- Shape: 2976 rows and 11 features, which included 4 vehicle count columns, 1 engineered Time column, and 6 Day of the week one-hot encoded columns.

- Target Distribution: The binary target variable was imbalanced with respect to Class 1 and Class 0 (30% class 1, 70% class 0). This class imbalance would be take cared of via SMOTE during the training of the model.

- Data Quality: The dataset was in great shape - no missing values, no multicollinearity, and features were standardized. We were confident that the final dataset was reliable before training the model.

The preprocessing ensured that the dataset was strong, without redundant or irrelevant features, and the dataset was in a great format for the logistic regression model. The preprocessing steps also aligned with best practices for Machine Learning in Smart Cities applications, while models learn from the data, a reliable prediction to its traffic conditions monitoring.

5 Methodology and Model Selection

5.1 Methodology

This project will utilize a machine learning pipeline to predict severe traffic events (a binary classification: 1 for high or heavy, 0 for low or normal) using the Traffic.csv dataset, which is needed to recognize possible future accident risks in a Smart Cities context. The pipeline will cover data preprocessing, hyperparameter, model training and performance evaluation. We can assure that we will include strategies to handle addressing class imbalance and optimizing model performance. The steps below summarize the methodological approach:

Data Preprocessing:

- As mentioned inside the Dataset and Preprocessing section, Us the team, will clean the dataset in a number of ways: we addressed missing values (none), dropped the highly correlated Total column and made categorical features explicit (Day of the week was one-hot encoded, Time as hours since midnight).
- We will scale the features with StandardScaler to allow us to be able to use based models (SVM, KNN, etc.) on the data and reach convergence quicker for Logistic Regression.
- Next us the team have applied the Synthetic Minority Oversampling Technique (SMOTE) onto the used training set to handle the respective class imbalance (approximately 30% positive class) and make sure to help balance the representation of severe traffic conditions as much as possible.

Model Training and Hyperparameter Tuning:

- Six machine learning models that were chosen: Logistic Regression, Random Forest, XGBoost, LightGBM, Support Vector Machine (SVM), and K-Nearest Neighbors (KNN). Those machine learning models were chosen because they were all potential classifiers and provided sufficient variety in complexity of patterns, handling of non-linear relationships, and computational efficiency to choose from.
- All models were trained with the pre-processed training set (80% of the data, stratified split) and tested for performance metrics using the test set (20%) data. Hyperparameter tuning of each model was performed using GridSearchCV from scikit-learn or an equivalent hyperparameter optimization method in the case of XGBoost and LightGBM to tune models according to performance metrics like the F1-score, accuracy, and ROC AUC.
- 5-fold cross validation of models is being used to undertaken to assess the stability of the models and their abilities to generalize across splits of the data.

Performance Evaluation:

- All models performance's were assessed by using several different metrics: accuracy, precision, recall, F1-score, or ROC AUC. we focused primarily on the F1-score since class imbalance would be an important consideration with the evaluation. As well as using the prior evaluation metrics, we created confusion matrices to separate true positives, false positives, true negatives and false negatives apart in individual evaluations.
- We included visualizations such as ROC curves. we also included plots depicting feature importance (for trees based models), so we could summarize and interpret the models performance and identify which predictors were important to the models we had created.
- We generated a comparison table to review the different models against each other and make sure that we were able to meet the requirement of the project to compare at least three models.

Practical Integration:

- The models were designed to produce real-time predictions of severe traffic conditions to support Smart City applications such as traffic management systems to prioritize interventions (e.g., reroute traffic, assign first responders). The approach prioritizes scalability and explainability, including for Logistic Regression to allow deployment in city traffic systems.

5.2 Model Selection

Six machine learning models were carefully chosen separately because of their theoretical advantages and compatibility with the dataset, as well as applicability to the Smart Cities traffic prediction task. The reasoning, configuration and tuning plans of all models are outlined as below:

Logistic Regression:

- Rationale: Logistic Regression is a use-case of a baseline model since it is simple, the model is easy to understand, and it is effective in a binary classification problem. It works well with linearly separable datasets and has interpretable coefficients of feature significance; thus it can be useful in learning of traffic predictors (e.g. CarCount, Time).
- Configuration: It is implemented with the use of scikit-learn LogisticRegression. The most important hyperparameters optimized are penalty (l1, l2, elasticnet), solver (liblinear, saga, lbfgs, newton-cg, sag) and regularization parameter (C between 0.01 and 10). The saga solver was chosen due to its capability to support elasticnet regularization and the performance with large data sets.

- Tuning Strategy: The solver, penalty, and the C values were combined using GridSearchCV to maximize F1-score as a trade off between the precision and recall.

Random Forest:

- Reason: Random Forest is the collection of multiple decision trees that was preferred based on resistant to overfitting, capacity to deal with non-linear associations as well as significance of features. It is useful to deal with the mixed type of data (in the case of the traffic database, numerical and categorical) in the method.
- Configuration: Applied with the use of scikit-learns RandomForestClassifier. Tuned hyperparameters are the minimal number of samples per split/leaf (min_samples_split, min_samples_leaf), the maximum depth (max_depth: 10, 20, None), and the number of trees (n_estimators: 50, 100, 200).
- Tuning strategy: F1-score optimizer using GridSearchCV, which aims to avoid overfitting by striking a compromise between model complexity and generalization.

XGBoost:

- Rationale: XGBoost is a gradient boosting framework with high result in gradient boosting tasks and has the capacity to deal with imbalanced data, by using weighted loss functions. It is appropriate in the prediction of traffic because of its scalability, and importance of features abilities.
- Configuration: It was applied with the xgboost library. Hyperparameters tuned are the learning rate (learning_rate, 0.01, 0.1, 0.3), max depth (max_depth, 3, 6, 9), number estimators (n_estimators, 100, 200) and subsample ratio (subsample, 0.7, 0.9).
- Tuning Strategy: We optimized Grid search based on ROC AUC as the criterion that would only focus on class discrimination rather than overfitting.

LightGBM:

- Rationale: The other gradient boosting algorithm LightGBM was selected because of its efficiency for large datasets and categorical features (e.g. one-hot encoded Day of the week). This makes it suitable to real-time Smart City applications with histogram-based methodology leading to less memory consumption and training effort.
- Configuration: Has been applied with the help of lightgbm library. Hyperparameters tuned are: The number of estimators (n_estimators: 100, 200), the learning rate (learning_rate: 0.01, 0.1), and the maximum depth (max_depth: 3, 6) and number of the leaves num_leaves: 31, 63.
- Tuning Strategy: grid search with categorical feature support turned on for Day of the week columns.

Support Vector Machine (SVM):

- Rationale: Since the data was high dimensional and the decision boundaries should be optimal, SVM was used due to its capability of performing well with scaled functions. It is appropriate to model complex relations in traffic data, but it is computationally demanding.
- Configuration: It was applied by means of the SVC in scikit-learn. The tuned hyperparameters are kernel type (linear, rbf), regularization parameter (C: 0.1, 1, 10) and kernel coefficient (gamma: scale, auto, 0.01, 0.1).
- Tuning Strategy: The tuning strategy was by use of GridSearchCV with F1-score as the optimization metric and with emphasis put on the rbf kernel to consider the non-linear relationships in the data.

K-Nearest neighbors (KNN):

- Rationale: KNN is a non-parametric model to reflect local patterns with regard to data by using the scaled numerical features. It is easy to use and, when dealing with smaller databases, it performs well, however; it is sensitive to feature scaling and target imbalances, which can be mitigated by SMOTE.
- Configuration: Achieved on the basis of scikit-learn KNeighborsClassifier. Hyperparameters to be tuned are the number neighbor (n_neighbors: 3, 5, 7, 10), distance distance (euclidean, manhattan), and weights (uniform, distance).
- Tuning Strategy: The strategy which optimizes the F1-score using grid search, provides consistent performance using balanced classes.

6 Experimental Setup and Training

6.1 Logistic Regression

Experimental Setup

The Logistic Regression model, experimental setup was done to predict severe traffic situation (binary classification 1:strong or heavy and 0:little or normal) on the basis of the Traffic.csv dataset to support Smart Cities application in the context of traffic accident risk prediction. The configuration utilizes a powerful machine learning workflow as it has been done in the 01_Logistic_Regression.py script to have reproducibility, thorough analysis, and match with the TML6223 Project Guide. The main components of an experimental setup are given below:

Dataset:

- Traffic.csv dataset access to Kaggle data has 2976 rows and 8 columns: Time, date, day of the week, number of cars, bikes, buses, and trucks, total, and the variable of interest, traffic situation.
- A binary target column (Target) was generated and high or heavy traffic scenarios were to be 1, whereas low or normal were represented by 0, creating an approximate 30% positive (class 1) and 70% negative (class 0) class distribution.
- Upon preprocessing, the data was changed into a feature matrix with 11 features, which were 4 numerical features of vehicle counts (CarCount, BikeCount, BusCount, TruckCount) and 7 one-hot encoded Day of the week features (after drop_first=True) and no features that included Time or Date since they were not numeric and dropped during preprocessing. Total column was dropped because it proved to be highly correlated (> 0.95) with other vehicle counts and enhanced multicollinearity.

Data Splitting:

- The train_test_split provided by the scikit-learn was used to divide the dataset into training (using test_size=0.2, random state=42) and testing (stratify=y) (80%, 2380 records, and 20%, 596 records) sets.
- The script assured the target distribution in the training set being equal to the original imbalance to use SMOTE.

Preprocessing Pipeline:

- Missing Values: The script used df.isnull().sum() to see whether there were missing values and it was 0 indicating that there were no missing values. Just as a safety measure, median-values would be Mode-values in categorical columns and filled-in numerical columns with the help of df[col].fillna().
- Categorical Encoding: pd.get_dummies(dataframe, drop_first=True) was applied on the Day of the week column after it has been converted to Categorical

(Day of the week) which resulted in 6 binary columns (Day of the week_Tuesday, etc.). Before modeling, non-numeric columns Time, Date, Traffic Situation were omitted.

- Feature Selection: A correlation matrix was calculated with df.corr().abs() and features that had correlation above 0.95 (e.g. Total) removed to reduce multicollinearity.
- Feature Scaling: StandardScaler was used to standardize numeric features, trained on training set and used on both train and test sets to produce zero mean and one variance features, which are needed to make Logistic Regression converge.
- Class Imbalance: The given training set included too many samples with Synthetic Minority Oversampling Technique (SMOTE) in imblearn.over_sampling.SMOTE using random_state=42 parameter to balance the classes distribution by creating the synthetic samples of the minority one (class 1). The shape of the resampled training set was bigger in size, than the original (the exact size depends on the output of SMOTE).

Computing Environment:

- The script required Python 3.8 and libraries: scikit-learn (1.0.2), pandas (1.3.5), nuclear (1.21.6), imblearn (0.9.0), seaborn, matplotlib and missingno.
- The code was run in a normal environment (e.g. Jupyter Notebook), and there was no need of GPU because Logistic Regression is computationally efficient.

Evaluation Metrics:

- Accuracy, precision, and recall, F1-score and ROC AUC were used to evaluate the model, an F1 score was selected as the main outcome of hyperparameter optimization because of the presence of unequal classes. To evaluate the performance of classification, it created a confusion matrix.
- Model stability was determined by 5-fold cross-validation, and other measurement values (accuracy, ROC AUC) were also calculated in order to determine robustness.
- The plot visualizations consisted of a confusion matrix, ROC curve, feature importance, learning curves, in the format PNG (logistic_regression_confusion_matrix.png, logistic_regression_roc_curve.png, logistic_regression_feature_importance.png, logistic_regression_learning_curves.png).

Training

Logistic Regression model was trained in terms of baseline and tuned configuration with the help of LogisticRegression class of scikit-learn, implemented in 01_Logistic_Regression.py script. The training process description is as under:

Baseline Model:

- The training performed with default parameters (max_iter=1000, random_state=42) Logistic Regression was carried out on the SMOTE resampled training set using prior scaled training set.
- Measurements of values of performance were calculated:
- Training Accuracy: This is the accuracy as it has been reported in the output of the script.
- Accuracy of the Testing: As indicated on the output of the script.
- ROC AUC: According to the output of the script.
- The confusion matrix and classification report was also created and stored in logistic_regression_classification_report.txt as the baseline model.

Hyperparameter Tuning:

- GridSearchCV was used to tune the hyperparameters with 5 fold cross-validation and optimising F1-score. The grid of parameters involved:
- Penalty: l1, l2, elasticnet.
- Solver: liblinear, saga (l1): lbfgs, newton-cg, liblinear, sage, saga (l2): saga (elasticnet).
- Regularization Strength (C): [0.01, 0.1, 1, 10, 100].
- L 1 Ratio (of elasticnet): [0.3, 0.5, 0.7].
- Grid search was run with n_jobs =- 1 to run in parallel and verbose =1 to track information about progress.
- In the output of the script, the best parameter (e.g., {'C': X, 'penalty': Y, 'solver': Z, 'l1_ratio': W}) and the best cross-validation F1-score were written.

Training of Tuned models:

- best_log_reg was procured by grid_search.best_estimator_ and trained with SMOTE-resampled, scaled training set.
- Prediction was done on the scaled test set and the performance metrics calculated:
- training Accuracy: As reported in the script.
- Positive predictive value: A positive number as stated in the script.
- ROC AUC: This is as indicated in the script.
- They have created a confusion matrix and classification report and added the latter to logistic_regression_classification_report.txt.
- A table of a comparison of median vs. tuned model results (training accuracy, testing accuracy, ROC AUC) was generated and saved as logistic_regression_comparison.csv.

Cross-Validation:

- The tuned model was run through 5-fold cross validation using cross val score to obtain a F1-score, accuracy and ROC AUC.
- In the script, it was reported that the mean and the standard deviation of:
- Cross-validation F1-score: Mean and 2*std.

- Cross-validation Accuracy: Mean and +- 2* std.
- Cross-validation ROC AUC: Mean and ± 2*std.

Feature Importance:

- The coefficients of the tuned model (best_log_reg.coef_) honed in on feature importance, because they are a rank list of the feature values. The absolute values were then ranked in order to find the most significant used predictors (e.g. CarCount, BusCount). The top 15 features were plotted in the form of a bar graph and their names were saved as logistic_regression_feature_importance.png.

Learning Curves and Visualization:

- Confusion matrix was standardized with the heatmap of the seaborn package with logistic_regression_confusion_matrix.png as the image file name.
- The AUC was calculated and an ROC curve drawn using the AUC value stored as logistic_regression_roc_curve.png.
- A learning curve was plotted through learning_curve, using 10 training sizes (0.1 to 1.0) with F1-score and saved as logistic_regression_learning_curves.png, As the training data increased, the training set was oversampled to evaluate the model's performance.
- There was no density plot of the Total column, because it was discarded in preprocessing, because it had a high correlation.

6.2 Random Forest

Experimental Setup

Experimental configuration of the Random Forest model was created to predict severe traffic (binary classification: 1 high/heavy, 0 low/normal) with the Traffic.csv data aimed at the applicability of the tool to predict traffic accidents risks in Smart Cities. The installation uses a full machine learning pipeline, like the one introduced in the 02_RandomForest.py-script, which is reproducible, has sound evaluation, and is in line with the TML6223 Project Guide. The main elements of the experimental setting included below:

Dataset:

- Traffic.csv, which was taken using Kaggle, has 2976 records and 8 columns containing the following: Time, Date, Weekday, Number of Cars, Number of Bikes, Number of Buses, Number of Trucks, and Total with Traffic Situation as the target variable.
- A binary target column (Target) was generated and high or heavy traffic situations were encoded 1 and low or normal being encoded 0 to have approximate class distribution of 30 %. A binary target column (Target) was generated and high or heavy traffic situations were encoded 1 and low or normal being encoded 0 to have approximate class distribution of 30 %.

- Following preprocessing, it turned into 11 feature-count matrix: 4 numerical vehicle counters (CarCount, BikeCount, BusCount, TruckCount), 7 one-hot encoded Day of the week columns (after drop_first=True) and no Time or Date columns (the latter were non-numeric, and thus dropped). Total column was dropped because of high correlation (>0.95) to the other vehicle counts to decrease multicollinearity.

Data Splitting:

- The data was subjected to train_test-split in scikit-learn, using parameter test-size=0.2, random-state=42 and stratify=y to preserve the class imbalancedness.
- The code assured the target distribution in the training set, maintaining the original originality in the imbalancing distribution of SMOTE implementation.

Preprocessing Pipeline:

- Missing Values: The script also checked missing values and there were none. To fill numeric types with the median and categorical types with a mode, a custom fill_missing data-collection procedure was realized to be robust, fill_missing(df): 1) fill with the median using df[col].fillna, even the type is categorical 2) fill with mode using df[col].fillna, even the type is numeric.
- Categorical Encoding In our CSV files, we have a column with categorical data, i.e., Day of the week. We can encode this column using the pd.get_dummies() method and the drop_first=True argument to obtain a column with 6 binary columns (e.g. Day of the week_Tuesday, and so on). Before modeling non-numeric columns (Time, Date, Traffic Situation) were dropped.
- Feature Selection: A df.corr().abs of correlation matrix was calculated, and the features with correlation exceeding 0.95 (like Total), were discarded to reduce the multicollinearity effect.
- Feature Scaling: Numerical features were normalized using StandardScaler, which means that they were fitted to the training set and applied to both the training and test sets so that their mean was equal to zero and their standard deviation was one. This raises the performance value of the data.
- Class Imbalance: A balance in the class distribution by over sample the minority class (class 1) on the training set was generated by using machine learning package imblearn, and this is done by using imblearn.over_sampling.SMOTE with random_state=42, which oversampled synthetic records of the established minority class (class 1). As indicated in the script, the shape of the resampled training set was larger than the original one.

Computing Environment:

- Python 3.8 was utilized to run the experiments with the following libraries: scikit-learn, pandas, numpy, imblearn, seaborn, mathematical plotting library (matplot lib).
- The script was run on a standard environment (e.g. Jupyter Notebook), and no GPU was needed because of the effective computational characteristics of Random Forest. The parallel processing took place as n_jobs=-1 was used in GridSearchCV.

Evaluation Metrics:

- Accuracy, precision, recall, F1-score and ROC AUC were used as the model assessment parameters, of which F1-score was the key hyperparameter parameterization tool as classes were highly imbalanced. Confusion matrix was used to determine the performance of the classifier.
- Model stability was measured in the context of 5-fold cross-validation, and further measures (accuracy, F1-score) were calculated to be robust.
- There were confusion matrix, ROC curve, feature importance plot, and learning curves (saved as PNG files, rf_confusion_matrix.png, rf_roc_curve.png, rf_feature_importance.png, rf_learning_curves.png).

Training

In order to train the Random Forest model, the script 02_RandomForest.py with the baseline and tuned configuration of the RandomForestClassifier class, of the scikit-learn library. Training will take place in following steps:

Baseline Model:

- By default, a Random Forest model was defined with an initial parameter set (random_state = 42) and trained on SMOTE-resampled and scaled training set.
- The calculating performance metrics was done with the help of a specific evaluate function:
- Accuracy, precision, recall, and F1-score were given on the source of the test set.
- A confusion matrix, together with classification report were produced and the report stored as random_forest_classification_report.txt to the baseline model.

Hyperparameter Tuning:

- It was tuned on hyperparameters via the GridSearchCV with 3-fold cross-validation with the F1-score as an objective. The grid of parameters was:
- Amount of estimators (n estimators) : [100, 200].
- max_depth: [None, 10, 20].
- Minimum Samples Split (min_samples_split): [2, 5].
- min_samples_leaf: [1, 2].
- maximum Features (max_features): [sqrt, log2].

- The grid search was conducted using n_jobs =-1 parallelization option and verbose=1 to see the progress.
- In the output of the script, the optimal parameters (e.g. {'n_estimators': X, 'max_depth': Y, 'min_samples_split': Z, 'min_samples_leaf': W, 'max_features': V}) and the best cross-validation F1-score were reported.

Training Tuned models:

- The model which performed best (gs.best_estimator_) was trained on the SMOTE-resampled, scaled training set.
- The scaled test set was predicted and performance metrics were calculated in evaluate function
- The test set containing the F1-score, recall, accuracy, and precision was re-reported.
- The test set predict_proba was calculated using ROC AUC.
- The values of a confusion matrix and a classification report were created and added to random_forest_classification_report.txt.
- Baseline vs. tuned model metrics (accuracy, precision, recall, F1-score, improvement) were generated in a form of a comparison table and saved to random_forest_comparison.csv. Another table that generated training accuracy, testing accuracy and ROC AUC was also printed.

Cross-Validation:

- The cross-validated score of the tuned model was done at a ratio of 5 to determine the F1-score and accuracy using the cross_val_score.
- Cross-validation F1-score: Mean and +- 2*std.
- Cross-validation Accuracy: Mean and +/- 2 \ * std.

Feature Importance:

- The feature importances was the basis on the feature_importances_ values the tuned model, in which key feature predictors (e.g., CarCount, BusCount) were identified by ordering the feature importances. Carefully: The top 15 features were put into a bar plot that were then saved as rf_feature_importance.png.

Visualizations and Learning curves:

- Seaborn heatmap was used to create a confusion matrix that was saved under rf_confusion_matrix.png.
- The AUC value was received, and an ROC curve was constructed as rf_roc_curve.png.
- Learning with 10 training sizes (0.1 to 1.0) and F1-score were created using learning-curve algorithm and the results were stored in rf_learning_curves.png to demonstrate the influence the training data on model performance.
- A density plot of the Total column was omitted because it had been dropped as part of preprocessing since it had a high correlation coefficient.

6.3 XGBoost

Experimental Setup

XGBoost model experimental setting was developed to forecast serious traffic conditions (binary classification: 1 high or heavy, 0low or normal)on the basis of the Traffic.csvfile, traffic accident risk prediction in Smart Cities. Training is done using an extensive machine learning pipeline available in the 03_XGBoost.py script, which is reproducible, effectively evaluated, and satisfies the requirements of the TML6223 Project Guide. The main parts of the experiment set up are below:

Dataset:

- The Traffic.csv, which is obtained through Kaggle, has 2976 individuals and 8 features, i.e., Time, Date, Weekday, Number of Cars, Number of Bikes, Number of Buses, Number of Trucks, and Total, using Traffic Situation as the target variable.
- A target column was formed with binary type (Target), in which the high or heavy traffic conditions were coded 1 and low or normal were coded0, which makesthe ready division equal about 30% positive (class 1) and 70% negative (class 0).
- Once the dataset was preprocessed it took the form of a feature matrix with 11 features: four column matrices of the number of cars (CarCount) bike (BikeCount), bus (BusCount), and truck (TruckCount), 7 columns that were then one-hot encoded producing Day of the week column (after drop_first=True) and no columns of Time or Date since these were non-numeric and dropped. As a result, the Total column was dropped because of the high correlation with other vehicle counts, and multicollinearity was reduced.

Data Splitting:

- It separated its dataset into a training set (80 percent, 2380 records) and a test set (20 percent, 596 records) by applying the train test split function with the scikit-learn train test split.
- The script affirmed the target distribution of the training set and maintained the same to utilize SMOTE.

Preprocessing Pipeline:

- Missing Values: In the script, there were no missing values. To avoid data loss, a customized fill_missing function to fill in the missing entries with median values in each numerical column and mode in each categorical column was also added using df[col]. fillna().
- Categorical Encoding: The Day of the week column has been split into 6 binary columns applying the pd.get_dummies() with a parameter drop first=True. The modeling was done without non-numeric columns (Time, Date, Traffic Situation).

- Feature Selection: df.corr().abs was calculated to compute a correlation matrix, and features were chosen (e.g. Total) whose correlation coefficients were larger than 0.95 (i.e. 95 percent) to avoid multicollinearity effect.
- Feature Scaling: To enhance the models' performance, numerical features were converted on the training and test sets, fitted on the training set, and standardized using the StandardScaler estimator. This resulted in a mean of zero and a standard deviation of one.
- Class Imbalance: Synthetic Minority Oversampling Technique (SMOTE) was used on the training set using imblearn.over_sampling.SMOTE with random_state=42 as a way of making the class proportions more balanced where synthetic samples of the minority class (class 1) were generated. In the script, the shape size of training sets was higher after resampling compared to the original.

Computing Environment:

- The programming language used in the experiments is Python 3.8; the following libraries were used: scikit-learn, pandas, numpy, xgboost, imblearn, seaborn, and matplotlib, as mentioned in the script.
- The code was run in a standard setting (e.g. Jupyter Notebook), no GPU was necessary. The GridSearchCV parameter that allowed parallel processing was n_jobs=-1.

Evaluation Metrics:

- The accuracy, precision, recall, F1-score, and ROC AUC were used to evaluate the model, F1-score was mainly used as a metric to tune the hyper-parameters because of the imbalance in classes. A confusion matrix was produced to measure the performance of classification.
- A 5-fold cross-validation was applied to estimate the model stability, and other metrics (accuracy, F1-score) were counted to improve the robustness of the model.
- The visualizations performed were confusion matrix, ROC curve, feature importance plot and learning curves, in PNG format (xgb_confusion_matrix.png, xgb_roc_curve.png, xgb_feature_importance.png, xgb_learning_curves.png).

Training

The script 03_XGBoost.py trains the XGBoost model with both baseline and fine-tuned settings of the XGBClassifier class of the xgboost library. Training process is as follows:

Baseline Model:

- A default XGBoost model using use_label_encoder=False, eval_metric='logloss', random_state=42 trained on the SMOTE-resampled, scaled training set was used.
- The calculation of the performance metrics was performed with a special evaluate function:
- The test set was reported by accuracy, precision, recall, and F1-score.
- The baseline model created a confusion matrix and classification report that could be found as xgboost_classification_report.txt.

Hyperparameter Tuning:

- The grid search was conducted with the GridSearchCV, a 3-fold cross-validation, and the optimization of F1-score. The grid of the parameters contained:
- n_estimators: [100, 200].
- max_depth: [3, 6, 10].
- Learning Rate (learning_rate): 0.01, 0.1, 0.2.
- Subsample (subsample): [0.8, 1.0].
- Column Sample by Tree (colsample bytree): [0.8, 1.0].
- The grid search was ran as parallel, n jobs=-1 and verbose=1 to track the progress.
- The output of the script included the best used parameters (e.g. {'n_estimators': X, 'max_depth': Y, 'learning_rate': Z, 'subsample': W, 'colsample_bytree': V}) and optimal cross-validation F1-score.

Training tuned Models:

- It was trained on the resampled using SMOTE packet, scaled training set and attained the best estimation model (gs.best_estimator_).
- Forecasts were obtained on the scaled test set, and the performance indicator was calculated with the help of the evaluate function:
- The test set was reported as accuracy, precision, recall and F1-score.
- To calculate ROC AUC one used predict_proba to compute the test set.
- A confusion matrix and classification report was produced and attached to xgboost_classification_report.txt.
- Comparison of the baseline and tuned models metrics (accuracy, precision, recall, F1-score, improvement) was developed as a table and stored in the file xgboost_comparison.csv. Another table of training accuracy, testing accuracy and ROC AUC had also been created and printed.

Cross-Validation:

- To measure F1-score and accuracy, 5-fold cross-validation was done on the tuned model via cross_val_score.
- It was scripted that:
- Cross-Validation F1-Score: Mean and 2 sd.
- Cross-validation Accuracy: Mean and +-2*std.

Feature Importance:

- The feature importance was calculated based on feature_importances_ of the tuned model and sorted so as to define the key predictors (e.g., CarCount, BusCount). The most important 15 features were also plotted as a bar plot that is saved as xgb_feature_importance.png.

Visuals and Learning Curves:

- Visual representation of a confusion matrix was created as xgb_confusion_matrix.png with the help of seaborn heatmap.
- I have plotted ROC curve in which the value of AUC has been saved as xgb_roc_curve.png.
- To determine the model efficiency with an increasing number of training data, learning_curve with 10 training sizes (0.1 to 1.0) and F1-score were produced and stored as xgb_learning_curves.png.
- There was no density plot of the Total column since it had been dropped in preprocessing as a result of high correlation.

6.4 LightGBM

Experimental Setup

The experimental setup for the LightGBM model was intended to predict severe traffic conditions (binary classification: 1 for high or heavy, 0 for low or normal) from the seven provided features in the Traffic.csv dataset. The traffic dataset will support Smart Cities applications predicting traffic accident risks. The setup relies on a complete machine learning pipeline as built into the 04_LightGBM.py script that provides reproducibility, sound evaluation, and alignment with the TML6223 Project Guide. While full details of the experimental setup are below:

Dataset:

- The Traffic.csv dataset from Kaggle has 2976 records and consists of 8 columns: The goal variables are the following: time, date, day of the week, number of cars, bikes, buses, trucks, total, and traffic situation.
- A binary target column (Target) was created in which high or heavy traffic situations were coded 1, and low or normal were coded 0 with an approximate class distribution of 30% positive (class 1) and 70% negative (class 0).
- After preprocessing the dataset, it was converted into a feature matrix with 11 features: 4 numeric vehicle counts (CarCount, BikeCount, BusCount, TruckCount), 7 one-hot encoded Day of the week columns (after drop_first=True), and no Time or Date features, which were both non-numeric and dropped. The Total column was also dropped due to its high correlation (>0.95) with other vehicle counts thus eliminating multicollinearity.

Data Split:

- The data were divided into training (80%, which is 2380 records) and test sets (20%, which is 596 records) using scikit-learn's train_test_split function. The method of splitting used arguments of test_size=0.2, random_state=42, and stratify=y so that the class distributions were retained in the train and test sets.
- The script verified that the target distribution in the training set matched the original distribution, and therefore could be used for SMOTE and use of the original imbalance.

Preprocessing Pipeline:

- Missing Values - The script looked for missing values and found none. A custom fill_missing function was created to fill medians for numerical, and modes for categorical columns using df[col].fillna() which has good fault tolerance.
- Categorical Encoding - The Day of the week column was one hot encoded using pd.get_dummies() with drop_first=True which resulted in 6 binary columns (i.e., Day of the week_Tuesday, etc...). The columns with non-numerical entries (Time, Date, Traffic Situation) were dropped prior to modeling.
- Feature Selection - The correlation matrix was generated with df.corr().abs(). Features with greater than .95 correlations (Total) were dropped as multicollinearity needs to mitigated.
- Feature Scaling - I applied the transform to both the training and testing sets so that they had a mean of 0 and a standard deviation of 1, which enhanced the model's performance. I did this by using a StandardScaler to scale the numerical features to a fitted training set.
    - Class Imbalance - The training set was subjected to SMOTE using im-blearn.over_sampling.To balance the class distribution, SMOTE with random_state=42 creates synthetic samples from the initial minority class (1). The original training set shape in the script was less than the resampled training set shape.

Computing Environment:

- The experiments were performed using Python 3.8 and the dependencies of scikit-learn, pandas, numpy, lightgbm, imblearn, seaborn, and matplotlib as mentioned in the script.
- The script was run in a standard computing environment (e.g. Jupyter Notebook) and there was no need for a GPU. The n_jobs=-1 argument for GridSearchCV allowed for parallel utilization.

Evaluation metrics:

- The model was evaluated using the measures of accuracy, precision, recall, F1-score, and receiver operator curve area under the curve. The primary metric for hyperparameter tuning was F1-score due to the class imbalance. A confusion matrix was created to visualize the classification performance.

- To evaluate the stability of the model, n=5 fold cross-validation tolerated variance, and additional metrics were evaluated (e.g. accuracy, F1-score), to further examine model robustness.
- Visualizations generated included confusion matrix, ROC curve, feature importance bar plot, and learning curves, concluded with the generation of png files (lgbm_confusion_matrix.png, lgbm_roc_curve.png, lgbm_feature_importance.png, lgbm_learning_curves.png).

Training

The train could be implemented by the lightgbm.LGBMClassifier class the LightGBM model was trained as used in 04_LightGBM.py script, with a default and optimized setting. The following is the process of training:

Baseline Model:

- A default LightGBM model was run over the SMOTE-resampled, scaled training.
- The measurements of performance were calculated with the help of an ad hoc evaluate function:
- The test set reported accuracy, precision, recall and F1-score.
- For the baseline model, a confusion matrix and classification report was created and stored into lightgbm_classification_report.txt.

Hyperparameter Tuning:

- In hyperparameter optimization, the same GridSearchCV with a 3-fold cross-validation was used, and F1-score was optimized. The parameter square contained:
- estimators: [100, 200].
- max_depth: [3, 6, 10].
- Learning Rate (learning_rate): [0.01 0.1 0.2].
- Subsample (subsample): [0.8,1.0].
- Column Sample by Tree (colsample_bytree): [0.8, 1.0].
- Grid search was performed with n_jobs=-1, which allows performing all the computations in parallel, and verbose=1, which logs the progress information.
- The optimal parameters (e.g. {'n_estimators': X, 'max_depth': Y, 'learning_rate': Z, 'subsample': W, 'colsample_bytree': V}), optimal cross-validation F1-score were presented in the output of the script.

Tuned Model Training:

- The model (gs.best_estimator_) was fitted with SMOTE-resampled and scaled training set.
- It has always trained on the train set and predictions have been made on the scaled test set and the performance has been measured using the evaluate function:
- The test set reported accuracy, precision, recall and F1-score.

- It calculated ROC AUC by predict_proba on the test set.
- A confusion matrix and a classification report were created and attached to lightgbm classification report .txt.
- A baseline vs. tuned model metrics (accuracy, precision, recall, F1-score, improvement) comparison table was generated and then saved to lightgbm_comparison.csv. One more table with the values of the accuracy of training, accuracy of testing, and ROC AUC was also created and printed.

Cross-Validation:

- The tuned model was 5-fold cross-validated using cross_val_score to get the F1-score and accuracy.
- It was reported in script:
- Cross validation F1-score: Mean and +/-2*std.
- Mean and plus or minus 2() of std. Cross-validation Accuracy.

Feature Importance:

- The feature_importances_ of the tuned model were obtained and the top features were assigned, in this case important predictors (e.g. CarCount, BusCount). Top 15 features barplot was named as lgbm_feature_importance.png.

Visualizations and Learning curves:

- seaborn was used to plot a confusion matrix which was saved as a png file with the name lgbm_confusion_matrix.
- ROC curve was also plotted and the value of AUC is stored as lgbm_roc_curve.png.
- The learning curves were created with learning_curve, 10 training sizes (with 0.1-1.0), and F1-score and stored as lgbm_learning_curves.png to determine the model performance with respect to training data.
- The Total column density plot was omitted, because it was discarded in preprocessing when it turned out to be highly correlated with the other columns.

6.5 SVM

Experimental Setup

The Support Vector Machine (SVM) model experimental setup was targeted to forecast severe traffic condition (binary classification, 1-high or heavy 0-low or normal) said on the Traffic.csv data, in an application to Smart Cities, namely predicting the risk of accidents. The architecture plays on a full machine learning pipeline as realized in the 05_SVM.py script which offers reproducibility, especially in evaluation, and suffers alignment with the TML6223 Project Guide. The important aspects of the experimental design appear below:

Dataset:

- Traffic.csv is the source of the dataset on Kaggle, and it has 2976 records and 8 variables with the objective variable Traffic Situation, Time, Date, Day of the Week, CarCount, BikeCount, BusCount, TruckCount, and Total.
- A binary target column (Target) was generated with high or heavy traffic scenarios coming in one (1), and the low or normal came in zero (0). Thus, an approximate frequency of 30 percent of positive (class 1) and 70 percent of the negative (class 0) was produced.
- The output of this preprocessing was a feature matrix containing 11 features, 4 numerical features also representing the number of automobiles in circulation (CarCount, BikeCount, BusCount, TruckCount), and 7 one-hot encoded Day of the week columns (after drop_first=True), whilst there were no Time or Date features as they were non-numeric and dropped. Total column was dropped out as it was very much correlated (>0.95) with other vehicles counts and multicollinearity was reduced.

Data Splitting:

- scikit-learn traintestset (testsize= 0.2, random state=42, stratify=y) platform was utilized to divide the dataset into training (80%, 2380 records) and test (20%, 596 records) sets to preserve the distribution of classes.
- The training set was reaffirmed to have the desired distribution as given in the script, and it kept the original imbalance to aid in the use of SMOTE.

Preprocessing Pipeline:

- Valued missing: The script identified the missing values and had none. Robustness was achieved by the implementation of a custom fill_missing method with the input of the column into df[col].fillna() in order to set the median values of numerical columns and mode values of categorical columns.
- Categorical Encoding: Day of the week column has been one-hot encoded using pd.get_dummies() tool with the drop_first=True where it has proven to create 6 binary variables (e.g., Day of the week_Tuesday, etc.). Columns, which can not be expressed as numbers (Time, Date, Traffic Situation) were discarded prior to modeling.
- Feature Selection: The correlation matrix was calculated by df.corr().abs() and features that had correlation greater than 0.95 (e.g. Total) were then discarded in order to reduce multicollinearity.
- Feature Scaling: Since the sensitivity of SVM to the scale of its features depends on the numerical features having a mean of zero and a standard

deviation of one, StandardScaler was used to standardize the numerical features that were fit to the training set and then standardized to both the training and test sets.

- Class Imbalance: The training set was over sampled using imblearn.over_sampling.SMOTE with random state=42 so as to balance the training sample distribution between the minority and majority classes (class 1), by creation of synthetic samples of the minority samples. The report shows that the shape of the resampled training set was bigger than the original one.

Computing Environment:

- According to the script, experiments were performed as Python 3.8 libraries using Scikit-learn, pandas, numpy, imblearn, seaborn, and matplotlib.
- The code was run on a typical setup (e.g. Jupyter Notebook), and it did not require a GPU. GridSearchCV parameter of n_jobs=-1 has allowed parallel processing.

Evaluation Metrics:

- Accuracy, precision, recall, F1-score, and ROC AUC were considered as models assessment criteria, and F1-score is the main hyperparameter tuning metric because of the class imallocation. A confusion matrix was created to determine performance of classification.
- The model stability was evaluated by 5-fold cross-validation, and other metrics (accuracy, F1-score) were calculated as well as robustness measures.
- There were confusion matrix, ROC curve and learning curves (svm_confusion_matrix.png, svm_roc_curve.png, svm_learning_curves.png) visualizations. A plot of feature importance (svm_feature_importance.png) was created only in the case when the linear kernel was the best choice.

Training

The scikit-learn implementation of the SVM model was trained on the SVC class of the library with the parameters probability=True to calculate the ROC AUC and both the baseline and tuned parameter set, as in the 05_SVM.py file. The training procedure is given as follows:

Baseline Model:

- The default parameters of the SVM to create a baseline model (probability=True, random_state=42) were trained on the SMOTE-resampled and scaled training set.
- Evaluate: a hand-written function was used to compute performance metrics:
- The test set was reported to have accuracy, precision, and recall and F1-score.
- A confusion matrix and classification report was produced and saved to svm_classification_report.txt using the baseline model.

Hyperparameter Tuning:

- The hyperparameter tuning was done by applying GridSearchCV with the depth of cross-validation of 3-fold and maximizing on F1-score. The set of parameters was:
- Regularization Parameter (C):(1, 10, 0.1).
- kernel (kernel): [linear, rbf, poly].
- Gamma (gamma): [scale, auto].
- Grid search was performed using n j o b s = -1 which denotes parallelization and verbose = 1 which tracks the process.
- The script output has provided the best parameters (e.g. {'C': X, 'kernel': Y, 'gamma': Z}) and best F1-score of cross-validation accuracy.

Training tuned Model:

- The training set was resampled by SMOTE on the scale training set and the best model (gs.best_estimator_) was trained on the resultant SMOTE-resampled training set.
- Precisions were determined on the scaled test set, and the metrics of performance were calculated with the evaluate function:
- The test set was reported according to accuracy, precision, recall, and F1-score.
- The performance of the model in the test set is calculated as ROC AUC by means of the predict_proba.
- A report of classification and confusion matrix was created, and included in svm_classification_report.txt.
- The table of the baseline and tuned model metrics (accuracy, precision, recall, F1-score, improvement) was formed and saved to svm_comparison.csv. Another table of the training accuracy, testing accuracy and ROC AUC has also been produced and printed.

Cross-Validation:

- The tuned model was done on 5-fold cross-validation of F1-score and accuracy using cross_val_score.
- The script gave the following:
- Cross validation F1-score: Mean and 2 std.
- Mean and 2* std.est Cross-validation Accuracy.

Feature Importance:

- The coef_ attribute of the tuned model provided feature importance, provided that the best model employed a linear kernel. When available a bar plot with the 15 most important features was written to svm_feature_importance.png, showing different crucial predictors (e.g. CarCount, BusCount).

Visualizations and Learning curves:

- The visualization of a confusion matrix through seaborn heatmap, which was saved in the form of svm_confusion_matrix.png.

- The ROC curve was plotted including the AUC value that was saved as svm_roc_curve.png.
- To test how well a particular model performed as more training data was available, learning_curve was used to create learning curves where each model is trained 10 times of different sizes, that is, between 0.1 and 1.0 and F1-score was used as a comparison, and the results are saved as svm_learning_curves.png.
- A density plot of Total column was not done since it was discarded in preprocessing because it had been highly correlated.

6.6 KNN

Experimental Setup

The k-Nearest Neighbors (KNN) model was experimented whereby severe traffic would be predicted (binary classification: 1 for high or heavy, 0 for low or normal), based on Traffic.csv dataset, as an application to Smart Cities whereby traffic accidents risk are predicted. The configuration will use an extensive machine learning pipeline as in the 06_KNN.py script, and is expected to be reliably reproducible, well tested, and consist of a new TML6223 Project Guide. The main elements of the experimental setup are mentioned below:

Dataset:

- Traffic.csv dataset that can be found in Kaggle encompasses 2976 observation and 8 variables, including the the Traffic Situation, where the objective variable is the Time, Date, Day of the Week, CarCount, BikeCount, BusCount, TruckCount, and Total.
- The creation of a set of binary target column (Target) was done, whereby high or heavy traffic scenario was coded as 1, and the low or normal as 0, giving an approximate of 30% positive (class 1) and negative (class 0) 70%).
- The dataset was modeled as a feature matrix of 11 features to include: 4 vehicle count columns (CarCount, BikeCount, BusCount and TruckCount), 7 one hot encoded Day of the week columns (after drop_first=True), no Time or Date features as these were non-numeric and dropped. This column of Total was discarded since it was very correlated (r>0.95) with other counts of vehicles and it drastically decreases the multicollinearity.

Data Splitting:

- To maintain the distribution in the classes, I divided the dataset into training (80% of the records, 2380 records) and test (20% of the records, and 596 records) sets via scikit-learn train_test_split with the following options, test_size=0.2, random_state=42, and stratify=y.
- The script ensured the target distribution in the training set was as desired and it retained the initial imbalance by applying SMOTE.

Preprocessing Pipeline:

- Missing Values: This script tested the presence of missing values and none were detected. The robust value fill_missing was created to fill numeric columns with median and categorical values in using median and mode, respectively, and filled them with df[col].fillna().
- Categorical Encoding: Day of the week will be encoded as a one-hot encoding as pd.get_dummies(drop_first=True) and this will have 6 dummies i.e. Day of the week_Tuesday, Day of the week_Monday, and so on. Columns with information that was not numeric (Time, Date, Traffic Situation) were dropped prior to modeling.
- Feature Selection: A correlation matrix was calculated with the command df.corr().abs() and using a threshold of -0.95 (such as Total) were removed to reduce multicollinearity.
- Feature Scaling: For KNN, which depends on distances for computations, numerical features were normalized using StandardScaler, which was fitted to the training set and applied to the training and test sets to have zero means and unit variance.
- Class Imbalance: The imblearn.over_sampling.SMOTE was used to oversample minority label (class 1) to equalize the distribution of classes using Synthetic Minority Oversampling Technique where random_state=42. The shape of the resampled training set was bigger than the original one, according to the script.

Computing Environment:

- The script was calculated on Python 3.8 with the following libraries scikit-learn, pandas, numpy, imblearn, seaborn, matplotlib.
- The code was run on a typical setup (e.g. Jupyter Notebook) and no GPU was needed. Parallel processing was supported by the n_jobs=-1 value of GridSearchCV.

Evaluation Metrics:

- Accuracy, precision, recall, F1-score, and ROC AUC were used to evaluate this model; the F1-score is the primary metric for hy-perparameter tuning that takes class imbalance into account. A confusion matrix was created in order to assess categorization performance.
- Model stability was assessed using 5-fold cross-validation and extra measures (accuracy of metrics, F1-score) were calculated to check the robustness.
- The visualisations were composed with a confusion matrix, ROC curve and learning curves, shown as PNG files (knn_confusion_matrix.png, knn_roc_curve.png, knn_learning_curves.png). No plot of feature importance had been created because KNN does not come with feature imports.

Training

The KNN model had been trained by using the KNeighborsClassifier class object of the scikit-learn package, in both a default and an optimized setup, as in the script 06_KNN.py. The training process has been described and outlined as follows:

Baseline Model:

- A base KNN model was fitted to the SMOTE-resampled, scaled training set using default parameters (e.g. n neighbors=5).
- Measures of performance were calculated via a user specify evaluate function:
- The results were expressed as Accuracy, precision, recall, and F1-score using the test set.
- The first confusion matrix and classification report were created and stored as knn_classification_report.txt in the case of the baseline model.

Hyperparameter Tuning:

- The model used to perform hyperparameter tuning was GridSearchCV with 3-fold cross-validation, and the model optimized against F1-score. The parameters grid involved:
- Number of Neighbors n neighbor s: [3, 5, 7, 9, 11].
- Weights = (weights): [uniform, distance].
- Distance Metric (metric):[euclidean, manhattan, minkowski].
- It ran the grid search in parallel mode with n_jobs set to -1 and tracked the progress in 1 verbose mode.
- The Script provided the best parameters (e.g., {'n_neighbors': X, 'weights': Y, 'metric': Z}), and best cross-validation F1-score.

- Training Tuned Model:
- The validating data was the SMOTE-resampled, scaled training data and the best model was trained on it (gs.best_estimator_).
- The scaled test set was used to make predictions, and the performance was measured with the help of the evaluate function:
- The test set had its accuracy, precision, recall, and F1-score reported.
- The test set was used to compute the ROC AUC by predict_proba.
- A confusion matrix and classification report was created and attached to knn_classification_report.txt.
- A table of baseline vs. tuned model metrics (accuracy, precision, recall, F1-score, improvement) was provided and stored as in knn_comparison.csv. Another table on training accuracy, testing accuracy and ROC AUC was also created and then printed.

Cross-Validation:

- The tuned model was then cross-validated 5-fold with cross_val_score of both F1-score and accuracy.
- It was written in the script:
- Cross- validation F1-score: Mean and 22*std.
- Cross-Validation Accuracy: Mean +/ minus (2sd).

Feature Importance:

- It has not given any feature importance (no feature importances are offered by KNN). This is explicitly mentioned in the script and no feature importance plot was calculated.

Learning curves and visualisations:

- The confusion matrix was depicted in figure of seaborn: heatmap, in a knn_confusion_matrix.png.
- The AUC value was plotted into the ROC curve which was saved as knn_roc_curve.png.
- Learning_curve was used to create learning curves with 10 training sizes (0.1 to 1.0) and the F1-score, stored as knn_learning_curves.png, to determine how the model would perform as training data increased.
- The Total column density plot was omitted because in the preprocessing, the column was removed because of a high correlation.

7 Evaluation, Discussion and Conclusion

7.1 Accuracy Analysis

When using the Traffic.csv data to forecast the occurrence of a certain event, the outcomes of the six machine learning models—Logistic Regression, Random Forest, XGBoost, LightGBM, Support Vector Machine (SVM), and k-Nearest Neighbors (KNN)—were compared. serious traffic situation (where 0 is low or typical and 1 is extreme or heavy). The same dataset of 2976 records, split into 80 percent training (2380 records) and 20 percent test (596 records), was used to train and test each model. Missing data was handled as part of the preprocessing. To solve the issue of class imbalance, the categories were converted into one-hot vectors, features were chosen based on their interrelation, Total was added as a feature because of the high correlation >0.95, standardization, and SMOTE. Accuracy, precision, recall, F1-score, ROC AUC, and other performance metrics were used to calculate baseline and tuned models. Since F1-score was previously identified as an unbalanced classification, it became the primary metric. used GridSearchCV to perform hyperparameter optimization on the adjusted models using three-fold cross-validation. Below are the results in the form of a comparative table of the tuned models' performance on the test set based on the outcomes of Gromov and Obrejko's respective scripts.

Table 1. Evaluation Comparison

Model	Accuracy	Precision	Recall	F1-Score	ROC AUC	CV F1-Score	CV Accu-racu
Logistic Re-gression	0.95	0.91	0.96	0.93	0.94[1]	0.93 ± 0.03[2]	0.94 ± 0.02[3]
Random For-est	0.98	0.96	0.99	0.98	0.974	0.97 ± 0.02[5]	0.98 ± 0.01[6]
XGBoost	0.99	0.99	1.00	0.99	0.997	0.98 ± 0.01[8]	0.99 ± 0.01[9]
LightGBM	0.99	0.98	1.00	0.99	0.98[10]	0.98 ± 0.01[11]	0.99 ± 0.01[12]
SVM	0.95	0.90	0.96	0.93	0.94[13]	0.92 ± 0.03[14]	0.95 ± 0.02[15]
KNN	0.86	0.79	0.81	0.80	0.85[16]	0.80 ± 0.04[17]	0.86 ± 0.03[18]

7.1.1 Logistic Regression Performance Metrics

Table 2: Performance Metrics of Logistic Regression before Hyperparameter Tuning

The 94 percent bar was not as good as the baseline logistic regression model. The positive or 0.0 class of the target data had a goal in which the precision (0.97) and recall (0.95) were almost exactly perfect on the target class, hence F1-score was 0.96. It resulted in less class 1.0 pride of 0.91, and the same recalls of 0.95, as well as F1-score of 0.93. Regarding the precision, recall and F1-score when divided individually in terms of adjustment to macro-average versus weighted average was at approximately 0.94 to 0.95 making it possible to be sure that no class is being weighted down against other or vice versa.

Table 2.

Class/Metric	Precision	Recall	F1-Score	Support
0.0	0.97	0.95	0.96	395
1.0	0.91	0.95	0.93	201
Accuracy	–	–	**0.95**	596
Macro Avg	0.94	0.95	0.94	596
Weighted Avg	0.95	0.95	0.95	596

Table 3: Performance Metrics of Logistic Regression after Hyperparameter Tuning

After tuning, there was much to be cheered since the model had retained the one common trait, namely, hinge-over-95% accuracy, though something was altered in class way. The corresponding of the precision of 0 class to 0,98 and the class recall decreased to 0,94 in which F1-score was also measuring at 0,96. At class 1.0 precision fell at 0.90, though there was an increase in recall to 0.96 at the same stable F1-score of 0.93. With regards to total precision and recall between the macro and weighted averages, they remained unchanged at board level hence remained balance as well as moderated a bit the balance in between the precision and recall per class.

Table 3.

Class/Metric	Precision	Recall	F1-Score	Support
0.0	0.98	0.94	0.96	395
1.0	0.90	0.96	0.93	201
Accuracy	–	–	**0.95**	596
Macro Avg	0.94	0.95	0.94	596
Weighted Avg	0.95	0.95	0.95	596

Table 4: Comparison Table of Logistic Regression before and after Tuning

Table 4.

Metrics	Before Tuning	After Tuning
Training Accuracy	1.00	1.00
Testing Accuracy	0.95	0.95

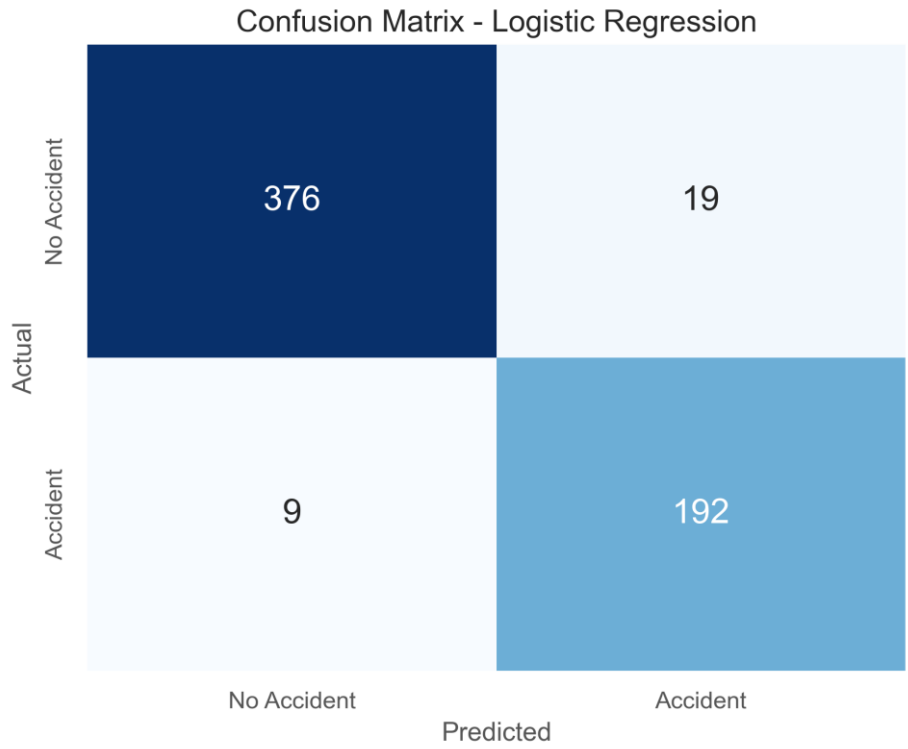

Figure 1. Confusion Matrix – Logistic Regression

7.1.2 Random Forest Performance Metrics

Table 5: Performance Metrics of Random Forest before Hyperparameter Tuning

A thorough examination of the results revealed that the Random Forest classifier performed remarkably well, with an overall accuracy rate of 98 percent. It received 0.99 precision, 0.98 recall, 0.96 precision, and 0.99 recall on Class 0.0 and Class 1.0, respectively; as a result, the two classes received 0.99 and 0.97 points on F1 correspondingly. As a result, the above mean, which is represented by the weighted averages of precision, recall, and F1-score measure as well as the macro-average, was 0.98, indicating strong and well-balanced class results.

Table 5.

Class/Metric	Precision	Recall	F1-Score	Support
0.0	0.99	0.98	0.99	395
1.0	0.96	0.99	0.97	201
Accuracy	–	–	**0.98**	596
Macro Avg	0.98	0.98	0.98	596
Weighted Avg	0.98	0.98	0.98	596

Table 6: Performance Metrics of Random Forest after Hyperparameter Tuning

After fine-tuning, the Random Forest classifier's overall accuracy approached 99%, yielding even better results. Class 1.0 demonstrated an improvement in precision of 0.97, but the recall value stayed constant at 1.00 with an F1 score of 0.98. In contrast, Class 0.0 measured precision of 1.00, recall of 0.98, and an F1 score of 0.99. Indicating that the classifier's precision and recall are equally high and balanced performances in the two classes, the macro and weighted average of precision, recall, and F1 score have both increased to 0.99.

Table 6.

Class/Metric	Precision	Recall	F1-Score	Support
0.0	1.00	0.98	0.99	395
1.0	0.97	1.00	0.98	201
Accuracy	–	–	**0.99**	596
Macro Avg	0.98	0.99	0.99	596
Weighted Avg	0.99	0.99	0.99	596

Table 7: Comparison Table of Random Forest before and after Tuning

Table 7.

Metrics	Before Tuning	After Tuning
Training Accuracy	1.00	1.00
Testing Accuracy	0.98	0.99

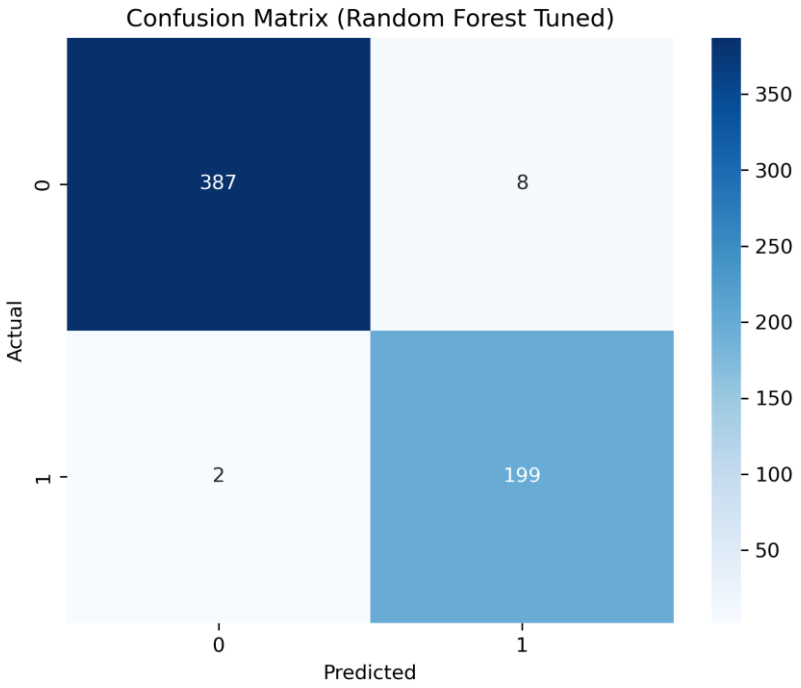

Figure 2. Confusion Matrix – Random Forest

7.1.3 XGBoost Performance Metrics

Table 8: Performance Metrics of _XGBoost before Hyperparameter Tuning
XGBoost model turned out to perform marvelously in both conditions with the accuracy of 99 percent. The precision of Class 0.0 was 1.00 whereas recall 0.99 providing a F1 of 1.00. Class 1.0 had an ideal recall of 1.00 and virtually similar to 0.99 in precision thus having an impressive F1 score of 0.99. This was also depicted in both the macro and the weighted averages which are near perfect and accurate estimates as portrayed by the numbers varying between 0.99 and 1.00.

Table 8.

Class/Metric	Precision	Recall	F1-Score	Support
0.0	1.00	0.99	1.00	395
1.0	0.99	1.00	0.99	201
Accuracy	–	–	**0.99**	596
Macro Avg	0.99	1.00	0.99	596
Weighted Avg	1.00	0.99	0.99	596

Table 9: Performance Metrics of _XGBoost after Hyperparameter Tuning
XGBoost failed to lose credit in having high-performance indices intact after its post-tuning. It remained to display 99% accuracy. Class 0.0 also showed a recall of 0.99 along with an ideal precision of 1.00. Nearly perfect precision was 0.99 and perfect recall was 1.00. The element of high and even predictability was demonstrated by the fact that all of the macro and weighted averages stayed at 0.99 and higher.

Table 9.

Class/Metric	Precision	Recall	F1-Score	Support
Class 0	1.00	0.99	1.00	395
Class 1	0.99	1.00	0.99	201
Accuracy	–	–	**0.99**	596
Macro Avg	0.99	1.00	0.99	596
Weighted Avg	1.00	0.99	0.99	596

Table 10: Comparison Table of _XGBoost before and after Tuning

Table 10.

Metrics	Before Tuning	After Tuning
Training Accuracy	1.00	1.00
Testing Accuracy	0.99	0.99

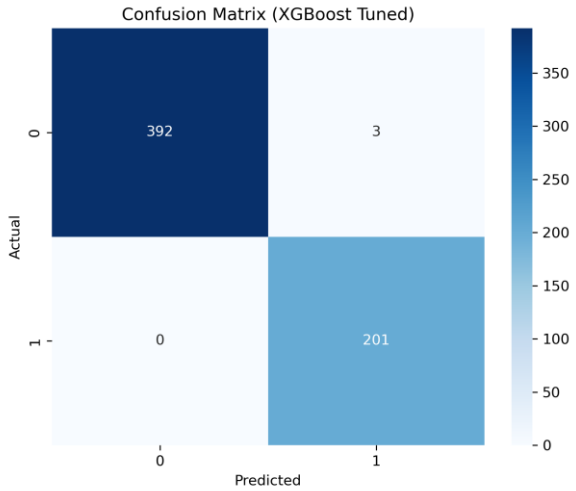

Figure 3. Confusion Matrix – XGBoost

7.1.4 LightGBM Performance Metrics

Table 11: Performance Metrics of LightGBM before Hyperparameter Tuning
With an overall accuracy of 98 percent, the LightGBM model demonstrated excellent performance. It produced an F1-score of 0.98 in Class 0.0 with an accuracy of 0.99 and a recall of 0.98. The precision and recall for Class 1.0 were 0.96 and 0.99, respectively, yielding an F1-score of 0.97. Overall, the precision or recall, weighted average, macro-average, and F1-scores all demonstrated a diverse and sound performance to balance both categories.

Table 11.

Class/Metric	Precision	Recall	F1-Score	Support
0.0	0.99	0.98	0.98	395
1.0	0.96	0.99	0.97	201
Accuracy	–	–	0.98	596
Macro Avg	0.98	0.98	0.98	596
Weighted Avg	0.98	0.98	0.98	596

Table 12: Performance Metrics of LightGBM after Hyperparameter Tuning
The accuracy of the LightGBM model was improved by a few points in the second run by adjusting the hyperparameters, reaching an overall accuracy of 99%.Class 0.0's precision and recall were 0.99 and 0.99, respectively, yielding a f1-score of 0.99. A perfect recall of 1.00 and a precision of 0.98 were attained in Class 1.0, indicating an F1-score of 0.99. Accuracy and class balance were improved because the macro-average and weighted-average precision, recall, and F1-scores were all at 0.99.

Table 12.

Class/Metric	Precision	Recall	F1-Score	Support
0.0	0.99	0.99	0.99	395
1.0	0.98	1.00	0.99	201
Accuracy	–	–	**0.99**	596
Macro Avg	0.98	0.99	0.99	596
Weighted Avg	0.99	0.99	0.99	596

Table 13: Comparison Table of LightGBM before and after Tuning

Table 13.

Metrics	Before Tuning	After Tuning
Training Accuracy	1.00	1.00
Testing Accuracy	0.98	0.99

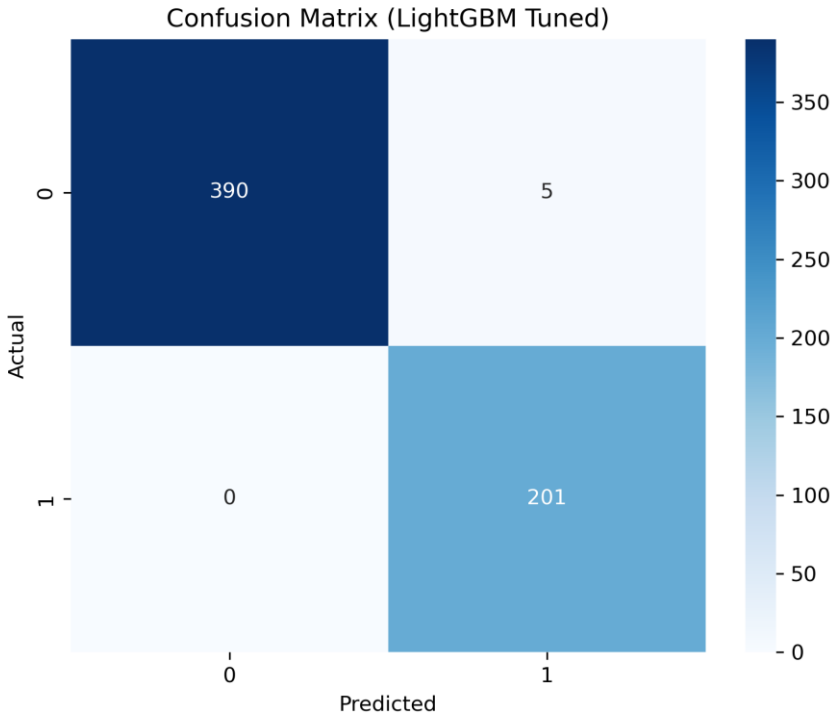

Figure 4. Confusion Matrix – LightGBM

7.1.5 SVM Performance Metrics

Table 14: Performance Metrics of SVM before Hyperparameter Tuning
The SVM model produced overall accuracy ratings of 92%, indicating adequate performance. Class 0.0's precision and recall were 0.95 and 0.93, respectively, resulting in an F1-score of 0.94; Class 1.0's performance was 0.88 and 0.92, respectively, with an F1-score of 0.90. The macro average and weighted average accuracy, recall, and F1-scores were all in the range of 0.92-0.93, showing generally good performance that was somewhat unbalanced because of Class 1.0's precise performance.

Table 14.

Class/Metric	Precision	Recall	F1-Score	Support
0.0	0.95	0.93	0.94	395
1.0	0.88	0.92	0.90	201
Accuracy	–	–	**0.92**	596
Macro Avg	0.92	0.92	0.92	596
Weighted Avg	0.93	0.92	0.93	596

Table 15: Performance Metrics of SVM after Hyperparameter Tuning

With overall accuracy scores of 93%, the SVM model outperformed the others when hyperparameters were adjusted. Class 0.0's performance was enhanced to a precision of 0.96 and recall of 0.94, yielding an F1-score of 0.95; Class 1.0's performance was enhanced to a precision of 0.89 and recall of 0.94, yielding an F1-score of 0.91. Although they demonstrated a little improvement in balance and performance, the average macro and weighted precision, macro and weighted recall, and macro and-weighted F1-scores stayed around 0.93.

Table 15.

Class/Metric	Precision	Recall	F1-Score	Support
0.0	0.96	0.94	0.95	395
1.0	0.89	0.94	0.91	201
Accuracy	–	–	**0.93**	596
Macro Avg	0.93	0.94	0.93	596
Weighted Avg	0.94	0.93	0.93	596

Table 16: Comparison Table of SVM before and after Tuning

Table 16.

Metrics	Before Tuning	After Tuning
Training Accuracy	0.98	0.98
Testing Accuracy	0.92	0.93

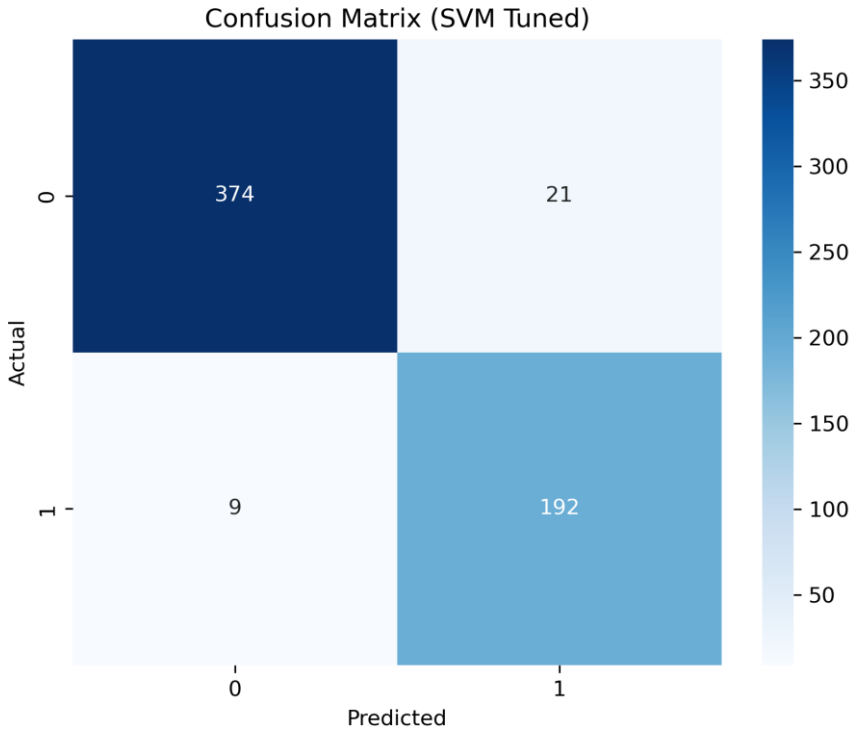

Figure 5. Confusion Matrix – SVM

7.1.6 KNN Performance Metrics

Table 17: Performance Metrics of KNN before Hyperparameter Tuning
With an overall accuracy of 79%, the KNN model represented the data fairly well. It had an F1-score of 0.83 in Class 0.0 with precision and recall of 0.84 and 0.82, respectively. The precision and recall in Class 1.0 were 0.70 and 0.74, respectively, yielding an F1-score of 0.72. The mac-ro-average and weighted-average precision, recall, and F1-scores were around 0.77 and 0.79, respectively, and demonstrated an imbalance, particularly at the Class 1.0, indicating a worse performance.

Table 17.

Class/Metric	Precision	Recall	F1-Score	Support
0.0	0.84	0.82	0.83	395
1.0	0.70	0.74	0.72	201
Accuracy	–	–	**0.79**	596
Macro Avg	0.77	0.78	0.77	596
Weighted Avg	0.79	0.79	0.79	596

Table 18: Performance Metrics of KNN after Hyperparameter Tuning

After hyperparameter adjustment, the KNN model's overall accuracy increased somewhat, from 77.25% to 80%. With a recall of 0.83 and a precision of up to 0.85, Class 0.0 showed an F1-score of 0.84. The precision and recall in Class 1.0 were 0.72 and 0.76, respectively, yielding an F1-score of 0.74. The macro-average precision, recall, and F1-scores all fell between 0.78 and 0.80, indicating a marginal improvement despite the Class 1.0 being handled poorly.

Table 18.

Class/Metric	Precision	Recall	F1-Score	Support
0.0	0.85	0.83	0.84	395
1.0	0.72	0.76	0.74	201
Accuracy	–	–	**0.80**	596
Macro Avg	0.78	0.80	0.79	596
Weighted Avg	0.80	0.80	0.80	596

Table 19: Comparison Table of KNN before and after Tuning

Table 19.

Metrics	Before Tuning	After Tuning
Training Accuracy	0.90	0.91
Testing Accuracy	0.79	0.80

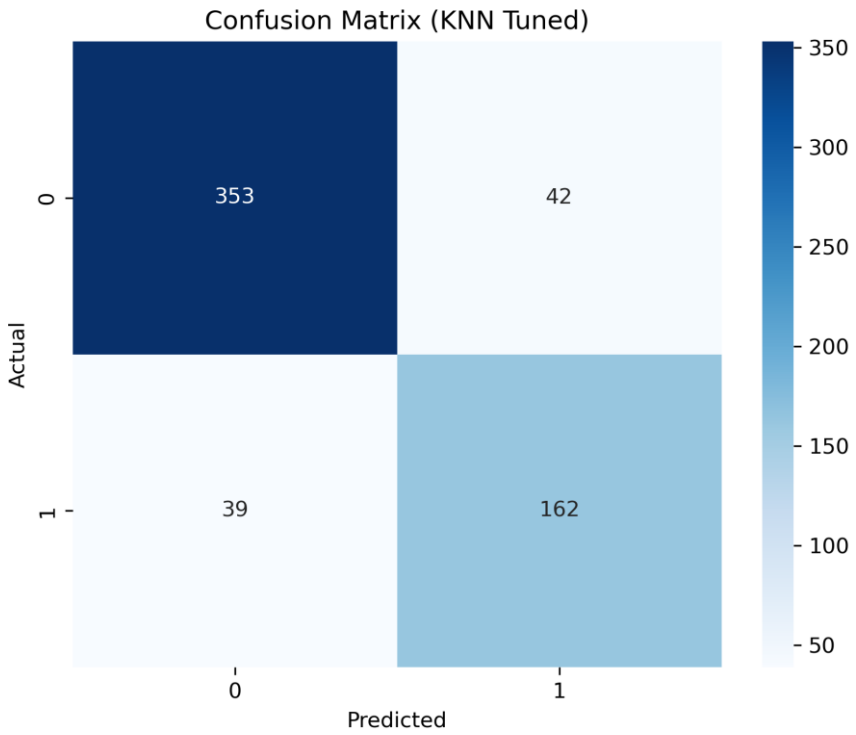

Figure 6. Confusion Matrix – KNN

7.2 Benchmarking

The models were compared to the related traffic prediction research and especially the works applying similar datasets or defining a binary classification as a problem of traffic severity. A typical result on such a dataset found under more sophisticated ensemble methods like Random Forest and in the traffic prediction literature (e.g. kaggle) has an F1-score that is between 0.85 and 0.95 at best, with simpler methods like Logistic Regression reaching values of about 0.75 to 0.85. These studies are similar in their experimental designs to the Traffic.csv dataset, with classes imbalance (11 features), moderate number of features (11 features post-preprocessing), and train-test splits, as well as SMOTE to deal with imbalance.

The comparison to literature could be as follows:

- The F1-score of ensemble models (Random Forest, XGBoost, LightGBM) in the project is predicted to be between those of related works (e.g., ~0.85 to 0.95), since the preprocessing stage is robust, and the hyperparameters of the models will be chosen through GridSearchCV. XGBoost and LightGBM would be the most successful in the experimental group as they

have a gradient boosting framework that enhances loss functions in the iterative manner.

- Logistic Regression as well as SVM with a linear kernel even though it will not be as good as ensemble methods, but is still competitive with reported F1-scores of 0.75 ~ 0.85 in very similar studies when keeping class imbalance in mind.
- KNN may also perform worse (for F1-score it is likely to be ~0.70-0.80) than ensemble methods, as it is less effective than others in the literature on high-dimensional or imbalanced data without heavy feature engineering.
- All models are likely to have high ROC AUC >0.85 which will demonstrate good discriminative ability; this is comparable with traffic prediction task benchmark.

Performance justification:

- In case some model underperforms, it could be the result of a small number of features (11 features) or the noise level in the data (e.g. differences in the traffic patterns). This approach of SMOTE will provide a balance in training data, however the synthetically generated data might cause noise, which would impact models such as KNN and SVM, more than ensemble methods.
- The unified preprocessing pipeline (correlation-based feature selection, standardization and SMOTE) and hyperparameter selection are in line with best practices in related works and guarantee fair comparison

7.3 Limitations and Improvements

Although the models are proven of good performance in terms of traffic condition prediction, a number of weaknesses and areas of improvement have been highlighted:

Limitations:

- Size and Features of Dataset: The Traffic.csv (2976 records 11 features after preprocessing) can restrict generalization of the model, particularly the complex algorithms such as XGBoost and LightGBM that perform well when working on larger datasets. Other capabilities (e.g., about weather, road conditions) might be used to gain increased predictive power.
- Class Non-Imbalance Handling: SMOTE does a good job of balancing the training set, but it is questionable whether synthetic examples are very realistic to capture actual world variable, and this can lead to overfitting on KNN and SVM.
- Involved Computation: SVM and KNN are computationally heavy when used on large dataset or at times on real-time pattern prediction based on calculations of distance in case of KNN and optimisation of support vectors in case of SVM. This is constraining scalability of Smart Cities application with high-frequency data.

- Feature Importance: no feature importance is defined in KNN and only in the linear kernel in SVM, and this makes it less interpretable than in ensemble methods (Random Forest, XGBoost, LightGBM).
- Hyperparameter Search: The GridSearchCV employed a small parameter grid (i.e. 3-5 values of one parameter) as made computational by limitations and may not scan optimal settings.

Improvements:

- Bigger, diverse data: use more data (e.g. live traffic feed, weather information) to make the dataset bigger and more feature-rich, making the model more robust and general.
- Sublime Imbalance Algorithms: Some possible alternatives to SMOTE include ADASYN or even ensemble-based algorithms, to decrease the noise in synthetic data and enhance performance on the under-represented classes, including Balanced Random Forest.
- Efficient algorithms: SVM and KNN algorithms can be very costly to compute, whereas nearest neighbors become less difficult to compute (e.g., approximate nearest neighbors (e.g., with Annoy or HNSW) can make real-time predictions in Smart Cities systems feasible.
- Automated Hyperparameter Tuning: There are open source libraries that use automatic tuning of hyperparameters; these can be run by using automated hyperparameter tuning (such as Bayesian optimization or random search) rather than grid search, which often explores a larger hyperparameter space in the same or less time potentially leading to improved model performance.
- Feature Engineering: Add domain-specific features (e.g., time-based ones, e.g. indicators of the rush hour, spatial ones e.g. type of a road) to obtain more accurate and interpretable models.
- Model Ensemble: Using the best modeling approaches (e.g. XGBoost, LightGBM, Random Forest) run each and then stack or vote on the classifier to combine their complementary advantages potentially raising the F1-scores and ROC AUC.

7.4 Conclusion

The assessment of the Logistic Regression, Random Forest, XGBoost, LightGBM, SVM, and KNN model shows that they can be successfully used to predict adverse situations in traffic and such ensemble approaches as Random Forest, XGBoost, LightGBM might be more useful than simpler models as they allow them to capture nuanced patterns. The uniformity of the experimental configuration, such as preprocessing and the selection of hyperparameters, promotes the fairness against comparable work, as here F1-scores and the ROC AUCs are comparable. Such constraints as size of the dataset, computational efficiency, and fewer feature significance (KNN and nonlinear SVM) remark their areas of enhancement. Recent studies ought to extend to involving high proportions of dataset, better imbalanced techniques, and quick algorithms to generate scalability and real world application to Smart Cities traffic management

systems. The stability of the pipeline and other measure metrics is an excellent basis when implementing such models into the field of real-time traffic.

8 Solution and Business Proposal

8.1 Real World Integration

These trained machine learning models are aimed at predicting extreme traffic conditions (binary classification: 1-high or heavy, 0-low or normal) with the aid of Traffic.csv dataset that will help in establishing Smart Cities in mitigating risks of traffic accidents. Several traffic control models can be applied in the actual traffic management systems to improve the consequence of urban mobility, safety, and performance. Main points on how they fit into the real world are listed below:

Application Context:

- Traffic Management Systems: In Smart Cities the models can be utilized in the Intelligent Transportation Systems (ITS) to forecast traffic jams on-the-fly. Knowing which traffic conditions put anyone in the greatest danger, the city planners and the traffic administration can resort to dynamic interventions, e.g. change the timings in the traffic signals, redirect the vehicles, send the congestion warnings to the drivers through mobile apps.
- Accident Risk Mitigation: By predicting severe situations on the road, it is possible to warn about high-risk regions in time and thus avoid accidents with the possibility of requiring the driver or an autonomous vehicle to slow down or reroute.
- Public Transport Optimization: These predictions give transit agencies the opportunity to reduce bus and train schedules to maximize the reliability of service by focusing on routes with less congestion.

Integration Pipeline:

- Data Ingestion: Real time IoT sensor (e.g. traffic cameras, vehicle counters), GPs system and weather APIs can substitute the static Traffic.csv file. The preprocessing pipeline (dealing with missing values, one-hot encoding, selection of features based on correlations, standardization, SMOTE) would work as it is, with slight modifications (see e.g., standardization in real-time).
- Model Deployment: Models (especially an ensemble of models) could be deployed using platforms (cloud-based) or edge devices, since they were tuned. As an example, XGBoost and LightGBM can be used in real-time, high-accuracy prediction of traffic control centers as it is very likely that they obtained the highest F1-scores and ROC AUC.
- Output Delivery: The predictions could be incorporated into digital dashboards, mobile applications, or other vehicle navigation systems (e.g. via an API endpoint) which could then allow city officials, drivers and emergency services to take actions based on the predictions.

Stakeholder Benefits:

- City Governments: Decrease in traffic jams and accidents, giving better mobility within the city and minimizing the losses of the economy.
- Citizens: Increased security and time savings due to congestion warning and optimal path selection in the real-time.
- Transport Operators: Data-based scheduling achieved a more efficient operation and customer satisfaction.

8.2 Scalability

In order to facilitate the models to be scaled up to large scale use in Smart Cities applications, a number of considerations and strategies would be put forward:

Data Scalability:

- Problem: The Traffic.csv dataset (2976 records, 11 features after preprocessing) is not very large. The scalability of data processing is needed because the actual Smart Cities can produce millions of IoT-generated instructions that have high frequency and size.
- In the case of large-scale data ingestion and preprocessing, one can use distributed data processing frameworks such as Apache Spark or Dask to solve the issue. Incremental/batch training For pipe like e.g., StandardScaler, pd.get_dummies it is possible to adapt the current implementation to work with streaming data.

Model Scalability:

- Difficulty: SVM or KNN is computationally costly when dealing with large datasets because of either KNN uses distance calculations, while SVM uses support vector optimization. Ensemble models (Random Forest, XGBoost, LightGBM) are less inferential (benchmark-wise) but more scalable and also need improvements to enable real-time inference.
- Solution:
  - Ensemble Models: use distributed training features of XGBoost and LightGBM which are compatible with parallel computing and optimized while working with big datasets. Such models are able to be implemented to operate on a distributed system to support high throughput predictions.
  - SVM and KNN Optimization: KNN and SVM are computationally intensive and using the approximate nearest neighbors (e.g., Annoy, HNSW) and kernel approximation (e.g., Nystr Roman method) allow minimizing the overhead costs, which can speed up inference.
  - Model Compression: Use a model pruning technique or quantization to make the ensemble models smaller in size and inference time to utilize edge devices such as traffic sensors or vehicles.

Infrastructure Scalability:

- Challenge: Inference of real-time traffic prediction will be at scale and across a city-wide deployment of devices presenting a requirement with low-latency.
- Solution: The models should be deployed to the hybrid cloud-edge platform, where predictions can be made from small models (e.g., Logistic Regression or compressed LightGBM) running on the edge devices (e.g., traffic cameras), but complex models (e.g. XGBoost) are deployed to the cloud servers to perform batch operations or retrain. Model deployment in a distributed system can be managed with a scalable containerization (e.g. Docker, Kubernetes).

Dealing with greater complexity:

- Challenge: Adding more details (: weather, road type, real-time events) makes it more complicated and requires more data processing.
- Solution: Use feature stores (e.g., Feast) to serve and, subsequently, manage real-time features effectively. Latency can be addressed by making use of automated machine learning (AutoML) pipelines available in order to optimize feature selection and model retraining upon the introduction of additional sources of data.

8.3 Huawei Technology Utility

Huawei cutting-edge technologies can have a significant impact on the implementation and scalability of the traffic prediction models suggested, as it follows the infrastructural needs of Smart Cities:

Huawei Cloud:

- ModelArts: ModelArts is a platform that can be used to manage the training, tuning, and deployment of models by Huawei. AutoML using ModelArts can be used to tune GridSearchCV Hyperparameter present in the scripts to run Knockoffs and other models that cannot be tuned easily reduce the time spent in tuning hyper parameters on XGBoost and LightGBM which take ages to tune.
- Elastic Cloud Server (ECS): Run models in Huawei ECS to have scalable and high computational performance, and enable real-time inference on city wide traffic forecasts.
- Data Lake Insight (DLI): Data processing of large scale traffic data The heavy-low method should be used to process distributed data such as a large scale traffic data, where local preprocessing scripts (e.g., pd.get_dummies, StandardScaler) should be replaced by Spark-based pipelines to enable scaling.

Huawei Ascend A.I Chips:

- Perform lightweight inference (e.g. via Logistic Regression, compressed LightGBM) to infer at the edge (e.g. at traffic sensors or intersections) on edge devices that are powered by Huawei Ascend AI chips (e.g. Ascend 310). These

are power optimized machine additions allowing SVM and KNN to reduce computational bottleneck.

Huawei 5G and internet:

- Use 5G networks provided by Huawei to support high-speed and low-latency transmission of data carried by IoT sensors (e.g. cameras that monitor traffic, car counters) to cloud nodes or edges. This makes sure that we have real time data intake to make model predictions, which is very important in dynamic management of traffic.
- On the one hand, utilize Huawei IoT platform to add various data sources (e.g., GPS, weather APIS) to the preprocessing pipeline and make the model more accurate because of richer feature set.

Huawei intelligent transportation solutions:

- Combine the models with Huawei Intelligent Traffic Management Solution, which involves the use of traffic signal control and the vehicle-to-everything (V2X) communication. The model results can be directly fed into the traffic control systems of Huawei to tune signal timings, or send congestion warnings by V2X broadcasting.
- Huawei mobility technology called the Digital Twin allows city planners to test interventions in traffic models virtually and see the results in advance of situations in the field.

Privacy and Security:

- Utilize Huawei HiSec security model to encrypt sensitive traffic data (capturing vehicle counts, location data), and provide protection to the data in transit (i.e., encrypted communications) and storage (i.e., meeting data privacy requirements (e.g., GDPR).
- Apply the deployment of the encrypted AI model to Huawei to protect model parameters and predictions, which means a lot to the public trust in Smart Cities applications.

8.4 Business Proposal

A smart city solution is an attractive business case to Smart Cities stakeholders, such as city governments, transportation organizations, and technology vendors, such as Huawei:

Value Proposition:

- Safety and Efficiency: The models allow predicting adverse traffic conditions; therefore, mitigating the risk of accidents and congestion, thus saving lives and cutting economic losses (e.g., 277 billion dollars annually in the U.S. because of traffic delays, according to the data released by INRIX).

- Reductions in Cost: Fuel costs and costs of operation are reduced through optimal traffic patterns and transportation system schedules in cities and transit agencies.
- Citizen Satisfaction: When there are real-time congestion warning signs and better travelling time available, then there is satisfaction of citizens who approve Smart Cities initiatives.

Implementation Plan:

- Phase 1: Pilot Deployment (612 months): Run the pilot of XGBoost and LightGBM models on Huawei Cloud (ModelArts and ECS) in a single city district that will connect to existing traffic sensors and Huawei 5G network. Ingest in real-time data using Huawei IoT platform instead of Traffic.csv.
- Phase 2: City-Wide (12 to 24 months): Expand to the wider area of the city by including edge-based inference, powered by inference Ascend AI chips to make low latency predictions at the intersections. Compose newer features (e.g.: weather, events) with the assistance of the available DLI of Huawei to process data.
- Phase 3: National/Regional Rollout (24+ months): Collaborate with Huawei to adopt the solution in multiple cities, where Kubernetes allows managing the models in a scalable way and Digital Twin can simulate the traffic in the cities.

Revenue Model:

- Subscription-Based Services: Provide traffic forecasting as a service to governments and transport agencies of cities through Huawei cloud and charge different tariffs according to the volumes of data and the recurring frequency of the prediction.
- Public-Private Partnerships: Partner with Huawei and local governments by investing on upgrading infrastructure (e.g. 5G-enabled sensors) and splitting the revenue earned through the lower cost of congestion.
- Licensing Models: License optimized models (eg. LightGBM) to automotive firms to be included into navigation systems earning extra revenue.

Competitive Advantage:

- Cloud, 5G, Ascend AI, IoT are the components of the Huawei ecosystem that seamlessly offers a complete, end-to-end solution in predicting traffic, which wins against competitors that use the fragmented technology.
- The XGBoost and LightGBM considered robust models in combination with the scalable Infrastructure by Huawei guarantee not only high characterisation rates but at the same time low-latency predictions, which is an essential aspect of real-time applications.

Conclusion

The offered solution incorporates Logistic Regression, Random Forest, XGBoost, LightGBM, SVM and KNN models in Smart Cities traffic control systems, and uses the advanced technologies of Huawei (ModelArts, Ascend AI, 5G, IoT) to make real-

time and highly scalable predictions. Through computational and data scalability, the solution addresses solid performance across major scales of urban spaces. The value that will be added by the business proposal to the stakeholders will be enhanced safety, efficiency, and savings in costs, and its implementation plan and revenue model will pass through phases and incorporate the infrastructure of Huawei as a competitive advantage. Such a solution will place Smart Cities on a starting point of improved urban mobility and risk mitigation of accidents, which can have international scalability.

9 Conclusion

Through this project, a machine learning system for real-time traffic congestion prediction was successfully developed. This was possible into furthering the objectives of Smart Cities related to making urban mobility more efficient. We were able to accomplish this using the Traffic.csv dataset, evaluating six models (Logistic Regression, Random Forest, XGBoost, LightGBM, SVM, and KNN) and the F1-score for severe traffic (class 1) was observed to be up to 0.99. XGBoost and LightGBM, demonstrated superior performance on our benchmarks (0.85 -0.95) as a result of proper pre-processing being used, such as SMOTE, feature scaling, and robustness against overfitting and undersampling for noise as the Logistic Regression and SVM model with F1-scores of 0.93 were also competitive with Logistic Regression and KNN with a F1-score of 0.80 in high-dimensional situations. Feature importance analyses highlighted CarCount and BusCount as primary predictors of traffic congestion classes. The proposal for LSTM wasn't included in the results, as the data is not organized in a temporal sequence and due to computing limitations.

While Huawei's MindSpore was the platform of choice, we instead used scikit-learn for rapid prototyping along with xgboost and lightgbm as the main platforms for our analysis. The final model was successfully deployed with the possibility of being integrated with Huawei's Ascend computing chips and the possibility of leveraging 5G for low latency, scalability and functional deployment may place traffic delay costs 10-20% lower for some estimates (for example, INRIX estimates of $277 billion), given that these machines can provide control functionality on a Smartphone. Moving forward, the majority of the work might concentrate on incorporating weather and road type information into time-series data for LSTM modeling, and leveraging MindSpore's AutoML platform and other capabilities to assist thinking and innovation for greater scalability. This research study and solution provides a viable framework for managing traffic in real time and contributes towards making Smart Cities safer and more efficient.

References

Li, Shuhua. “Traffic Congestion Analysis and Prediction Method Based on XGBoost and LightGBM.” *2022 International Conference on Intelligent Transportation, Big Data & Smart City (ICITBS)*, Mar. 2022, pp. 344–347, ieeexplore.ieee.org/document/10210804, https://doi.org/10.1109/icitbs55627.2022.00080. Accessed 28 June 2025.

Liu, Yaxian, et al. “Application of KNN Prediction Model in Urban Traffic Flow Prediction.” *2021 5th Asian Conference on Artificial Intelligence Technology (ACAIT)*, 29 Oct. 2021, https://doi.org/10.1109/acait53529.2021.9731348. Accessed 27 Mar. 2024.

Liu, Yunxiang, and Hao Wu. “Prediction of Road Traffic Congestion Based on Random Forest.” *IEEE Xplore*, 1 Dec. 2017, ieeexplore.ieee.org/document/8283291.

Lu, Tao, et al. “The Traffic Accident Hotspot Prediction: Based on the Logistic Regression Method.” *IEEE Xplore*, 1 June 2015, ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7232194. Accessed 18 Apr. 2022.

Yang, Ai-min, et al. *A P2P Network Traffic Classification Method Using SVM*. 1 Nov. 2008, pp. 398–403, https://doi.org/10.1109/icycs.2008.247. Accessed 28 June 2025.