Uncovering Insights from UK Car Accidents

Abhi Zanzarukiya
MEng Electrical and Computer Engineering
University of Waterloo
Waterloo, Canada
azanzar@uwaterloo.ca
Student ID: 20892635

Heli Mistry
MEng Electrical and Computer Engineering
University of Waterloo
Waterloo, Canada
h23mistr@uwaterloo.ca
Student ID: 20985952

***Abstract*—Car accidents continue to be a major issue in the United Kingdom, causing loss of life and damage to property. With the increasing availability of data, there is an opportunity to analyze the underlying factors contributing to accidents and uncover insights that can help prevent them. This project aims to explore the huge UK Car Accidents 2005-2015 [1], translate it to a non-trivial relational form and provides a CLI to access the insights from the dataset. By cleaning and restructuring the database design for efficient usage, we present the following findings: the locations of accidents with severity, the count of accidents based on people's profiles, and the accident count by junction type and severity within the jurisdiction of a specific police force. The insights generated from this analysis can help inform policy decisions and interventions that can ultimately reduce the number of accidents on UK roads.**

***Index Terms*—Car accidents, Data analysis, Relational database design, United Kingdom**

I. Introduction and Project Scope

Road accidents in the UK cause significant loss of life and property damage yearly, and the government collects extensive data on these incidents. However, the data is not easily accessible or in a usable format for analysis, as it is stored in several large CSV files. To overcome this challenge, we aim to design a clean and efficient database to organize and make the data more accessible for analysis.

To achieve this goal, we have created tables for each data type, defined relationships between them, and optimized the structure for efficient querying. As part of the project, we identified three main findings: accident severity at different locations, high-risk junction types, and high-risk driver profiles by vehicle type. We have also provided a client application in the form of a CLI interface that facilitates CRUD operations of the database.

We have implemented several data structuring activities to ensure the final database design meets all use-case requirements. Although we did not go through all years of data, the process we used is generic enough to parse the complete database eventually. The following process was utilized to structure and clean the data:

1) Load the dataset into temporary raw tables in the database.
2) Design the database structure using an Entity-Relationship model and translate it to a relational schema.
3) Implement Stored Procedures as required.

We have also thoroughly tested the CLI interface provided for interacting with the database. To make it more user-friendly, we have included CLI shortcuts for testing the connection to the database, listing all tables, showing user grants, showing table schema, and selecting rows based on a condition with LIKE.

II. Implementation Overview

A. Architecture Design

The architecture of our application is based on a client-server model, where the client is a Python CLI that the user interacts with, and the server is a backend MySQL database that stores the data.

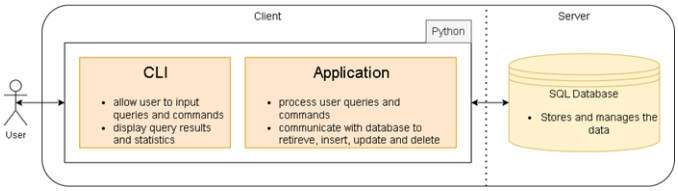


Fig. 1. Client-Server Architecture for the UK Accident Database Project

As shown in **Figure 1**, the user is the primary actor in the system. Users interact with the Python CLI to perform CRUD operations on the database and retrieve statistics and other information about road accidents in the UK.

The Python CLI handles user input and displays output to the user. It communicates with the backend SQL server through a database connector to perform CRUD operations on the database. In addition, the Python CLI includes the functionality to retrieve statistics from the database by executing pre-written queries in stored procedures.

The backend SQL server is responsible for storing and retrieving data from the database. It receives requests from the Python CLI through the database connector, processes them, and returns the requested data to the Python CLI.

Our application's architecture has been built to be both flexible and modular. Each component within the architecture has clearly defined responsibilities and interfaces, allowing us to incorporate new features and functionality into the CLI application easily. This design ensures that we can maintain a

robust database while expanding the application's capabilities in the future.

III. Data Readiness

A. Original Dataset Overview

The UK Car Accidents dataset was originally distributed among three large files: Accidents0515.csv, Vehicles0515.csv, and Casualties0515.csv. These files contained 31, 21, and 14 fields with relevant accident information. **Figure 2** presents an overview of the data contained in the downloaded CSV files.

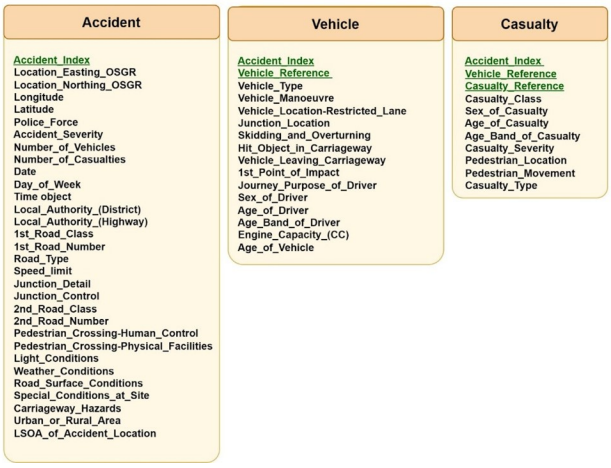


Fig. 2. Client-Server Architecture for the UK Accident Database Project

Most of the data in the dataset was collected from the STATS19 [2] form and stored in the form of categorical mapping. For instance, the "Light_Condition" field in the CSV has values like 1, 2, 3, etc., against which there was a pre-defined mapping (sample data shown below). To prepare the data for loading into the database, we joined the required columns with their mapping in the Jupyter Notebook and populated the fields with the relevant labels based on the code.

Data in CSV:

Light_Conditions
1
4
4
1
7
1
4

Corresponding Mapping:

Code	Label
1	Daylight
4	Darkness – lights lit
5	Darkness – lights unlit
6	Darkness – no lightning
7	Darkness – lightning unknown

B. Data Processing

After analyzing the fields in the datasets, we selected 24, 16, and 11 required fields from the Accidents.csv, Vehicle.csv, and Casualty.csv files, respectively. All these fields were processed and made ready for database insertion using the Python script. The approach we used for the data population in the database involved these steps:

1. Loading the CSV data into the RAW Temporary table Based on the selected fields, we created the following three schemas for the initial raw data population.

```
create table RawAccident(
 accident_id char(20),
 longitude decimal(8,6),
 latitude decimal(8,6),
 number_of_vehicles INT,
 number_of_casualties INT,
 datee DATE ,
 day_of_week INT,
 timee time,
 road_number INT ,
 speed_limit INT,
 police_force char(100) ,
 accident_severity char(50) ,
 local_authority_district char(100) ,
 local_authority_highway char(100) ,
 road_class char(50) ,
 road_type char(50) ,
 junction_detail char(100) ,
 junction_control char(100),
 pedestrian_crossing_facility char(100),
 weather_condition char(50),
 light_condition char(50),
 road_condition char(50),
 urban_or_rural char(10),
 special_condition char(50),

 primary key(accident_id)
 );
```

```
create table RawVehicle(
 accident_id char(20),
 vehicle_id INT,
 age INT,
 engine_capacity INT,
 age_of_vehicle INT,
 vehicle_location_restricted_lane char(100),
 junction_location char(100),
 hit_object_in_carriageway char(100),
 point_of_impact char(100),
 journey_purpose char(100),
 sex char(10),
 age_group char(10),
 vehicle_type char(50),
 vehicle_manoeuvre char(100),
 skidding_and_overturning char(100),
 vehicle_leaving_carriageway char(100),

 primary key(accident_id,vehicle_id)
 );
```

```
create table RawCasualty (
 accident_id char(20),
 vehicle_id INT,
 casualty_id INT,
 age INT,
 casualty_class char(50),
 sex char(10),
 age_group char(10),
 casualty_severity char(50),
 pedestrian_movement char(100),
 pedestrian_location char(100),
 casualty_type char(50),

 primary key (accident_id, vehicle_id, casualty_id)
 );
```

2. Data Cleaning After populating the data, the next step was to perform necessary clean-up activities to ensure the data was accurate and consistent. This included removing any duplicates and checking for NULL constraints. Some fields in the data contained "Data missing or out of range," which were replaced with NULL values to ensure consistency in the final relational schema. Additionally, fields like engine_capacity, age_of_vehicle had values of -1, which were also replaced with NULL.

3. Relation Schema Creation and Insertion Process The designed schema was then filled with the required data. The data is organized into various entities as shown in the database class diagram in **Section V**. The ERD to Class separation process is explained in detail in the following sections. To separate out the relevant fields from the Raw Temporary tables and populate it in the Relational Schema, we used the views shown in **Table I**.

IV. Entity-Relationship (ER) Model

To improve data organization and management, we have split the dataset into separate entities and relations based on the information contained in each file. The resulting ER-Diagram is shown in **Figure 3**.

Specifically, we split the data from Accidents.csv into Accidents, Roads, and Police Authority entities, while Casualties.csv was split into Casualty and Person Profile entities. Additionally, both Accident and Vehicles entities have a total specialization based on the junction information related to the accident. The resulting entities are interconnected through relevant relations, which are depicted in **Figure 3**.

Table II summarizes the relationship types for the entities in this ERD, while **Table III** explains the rationale behind our design choices.

TABLE I
VIEWS DEFINED

Views	Description
police_authority_view	Used to extract police_force, local_authority_district, and local_authority_highway from the RawAccident table and populate the PoliceAuthority table with this data. The unique authority_id is generated as an AUTO INCREMENT field.
Raw_road_View	Used to extracts road details such as road_class, road_number, road_type, speed_limit, and urban_or_rural from the RawAccident table and populate the Road table with this data. The unique road_id is generated as an AUTO INCREMENT field.
road_id_authority_id_view	Joins the RawAccident table with the Road and PoliceAuthority tables to retrieve the corresponding road_id and authority_id for each accident and is used to populate the Road_Authority table with this data.
accident_road_id_authority_id_view	Retrieves data needed for the Accident table from the RawAccident table. Additionally, it joins the RawAccident table with the Road and PoliceAuthority tables to retrieve the road_id and authority_id associated with each accident. This data is then used to populate the Accident table.
junction_accident_view	Retrieves data related to Junction_Accident from the RawAccident tables and is used to populate the Junction_Accident table with this data.
Junction_vehicle_view	Retrieves data related to Junction_Vehicle from the RawVehicle tables and is used to populate the Junction_Vehicle table with this data.
raw_casualty_view	Used to populates the Casualty table with data from the RawCasualty table, after joining it with the PersonProfile table to retrieve the profile_id.
raw_vehicle_view	Used to populate the Vehicle table with data from the RawVehicle table, after joining it with the PersonProfile table to retrieve the profile_id.

TABLE II
ERD RELATIONSHIP SUMMARY

Relationship	Relationship Type	Relationship Summary
Accident $\rightarrow$ Junction_Accident	Inheritance	Total, Disjoint
Accident $\rightarrow$ Non_Junction_Accident	Inheritance	Total, Disjoint
Vehicle $\rightarrow$ Junction_Vehicle	Inheritance	Total, Disjoint
Vehicle $\rightarrow$ Non_Junction_Vehicle	Inheritance	Total, Disjoint
Caused	Weak Entity Set	w.r.t Accident
Involved	Weak Entity Set	w.r.t Accident
Was_Engaged_In	Weak Entity Set	w.r.t Accident and Vehicle

TABLE III
ERD DESIGN CHOICES

Relationship Set	Entities Involved	Design Choice
Person_Profile	Casualty, Vehicle	Prepopulating a separate table to store profiles of persons involved in accidents and drivers (age, sex, and age_group) to avoid redundant data, improve data integrity, and reduce the risk of data inconsistency.
Junction_Accidents	Accident	Creating a specialized table for Junction-Specific Accidents to accommodate additional fields that are not relevant for accidents that did not occur at a junction, improving data consistency and reducing the risk of data duplication.
Weak Entity Sets	Vehicle, Casualty	Making the casualty and vehicle tables weak entities with respect to the Accident table to link each casualty and vehicle to the specific accident they were involved in without duplicating their Accident ID.
Police_Authority		Each accident occurs on a given road that can be managed by multiple authorities. To describe the authority that manages the current accident, we created a relation with the Accidents table. This way, we can link each accident to the relevant authority without repeating their data.
Occurred_On	Accident, Road	Including relevant attributes in the relationship set 'occurred_on' between the Accidents and Road tables to capture environmental conditions at the time of the accident.

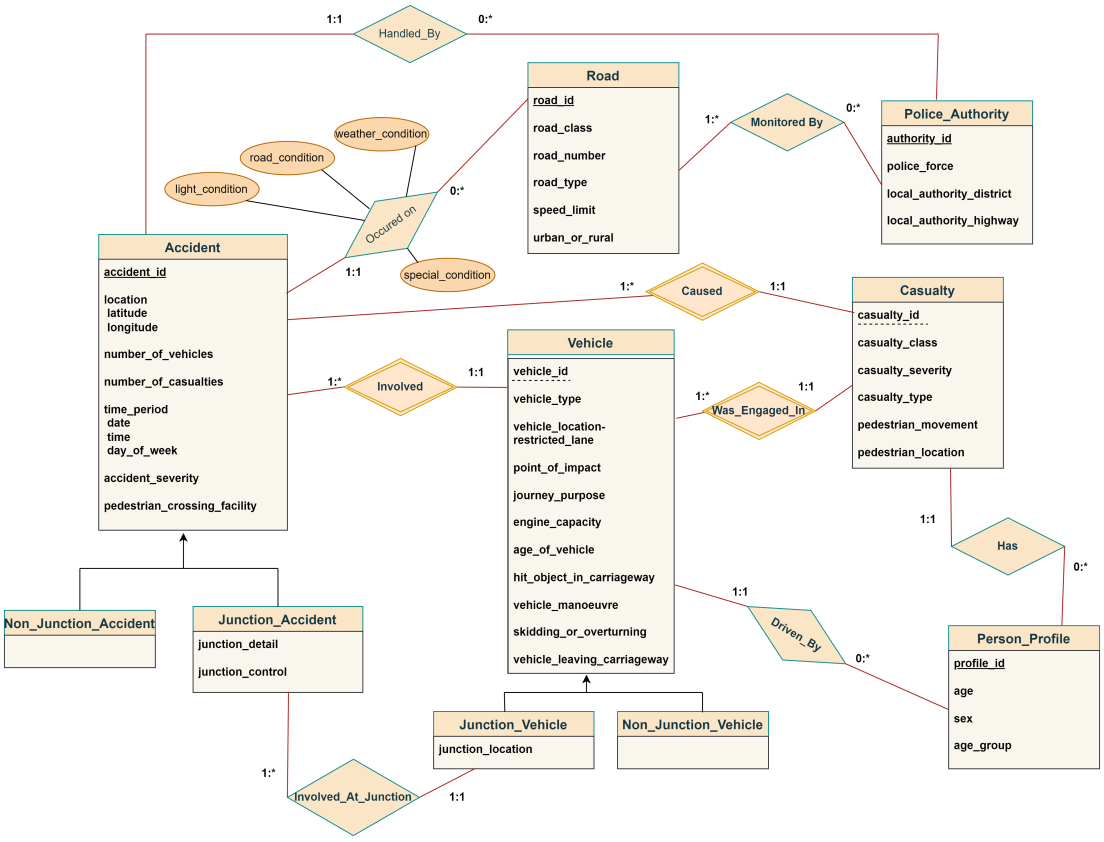


Fig. 3. ER Diagram for UK Accidents Database

V. Relational Schema

To represent the data in a way that aligns with our database design, we have created a relational schema based on the **ERD 3** we previously designed. This involved converting the ER diagram to a Relational Diagram by adhering to the appropriate rules and ensuring primary and foreign key constraints were added where necessary. The resulting schema can be seen in **Figure 4**.

A. Key Considerations in drafting Class Diagram from ERD

During the development of the class diagram from the ERD, we made the following key considerations:

1. Road and PoliceAuthority in many-to-many relation
To address the scenario where one road was managed by multiple police authorities and one police authority was responsible for handling many roads, a separate table called Road_Authority was created. This table holds the primary key of both tables to enable the necessary associations.

2. Accidents and Vehicles of Junction and Non-Junction type
There were two types of accidents: Non-Junction Accidents and Junction Accidents. To accommodate this, a single table called Junction_Accident was created, and a separate table for Non_Junction_Accident was not required. The same approach was taken for Vehicles, where information about the junction location was included in cases where the accident occurred near a junction.

3. Use of ENUM for a substantial number of attributes
The UK accident data collection process uses a sheet that includes checkboxes for the necessary attributes to be filled out. After analysing the data, it was noticed that the available options for certain fields hardly changed. Therefore, ENUM was used for these attributes, which allowed for more efficient data entry by providing predefined options. This approach minimized the need to store special descriptions separately since the available options were consistent throughout the data entries.

B. Field Constraints

Following **Table IV** lists the constraints on the fields we have in the database.

C. Stored Procedures

To facilitate the operations by the user via the CLI, we have implemented stored procedures as described in the **Table V** that handle the insertion and deletion of records into/from the appropriate tables. These stored procedures were designed to avoid the need for the user to enter unique identifying fields, such as authority_id. Instead, the identifying attributes of the police_authority are provided as input, and the stored

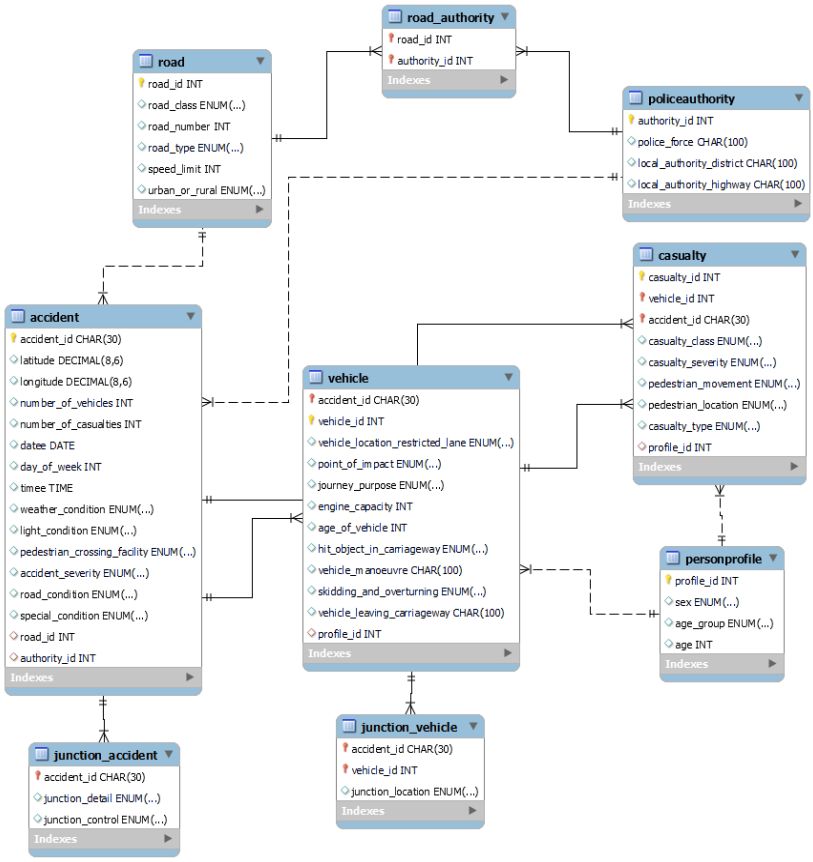


Fig. 4. Relational Schema in MySQL (Reverse-Engineered)

TABLE IV
CONSTRAINTS FOR EACH FIELD IN THE DATASET

No.	Field	Constraint
1	Accident_id	A unique alphanumeric string
2	Latitude, longitude	Valid coordinates Latitude: -90 to 90, Longitude: -180 to 180
3	Date	Date from a certain start 01/01/1900 to current date
4	Speed_limit	A positive integer
5	Police_force	A valid alphabetical string
6	Engine_capacity	A positive integer
7	Age_of_vehicle	A positive integer
8	Age_group and age	The age must lie within the specified age group

procedure correctly inserts the appropriate foreign key in the referring tables. Similarly, deletion of an accident record needs to subsequently delete the relevant data from the corresponding tables. The delete stored procedure helps facilitate this.

D. Indices

Our application has several queries that extract information from our database. Most of these operations already had the necessary indices in place due to the relevant primary and foreign keys.

However, additional indices were required to enhance their performance for some of the queries used for **statistical inferences VII**. For query 1, we created a secondary index on policeauthority(police_force, local_authority_highway, local_authority_district), as well as Accident(authority_id, datee, accident_severity, latitude, longitude) tables.

Similarly, for another query, we created secondary indices on vehicle(profile_id) and personprofile(sex, age_group) to be used with a primary index extension to profile_id.

These additional indices significantly improved the query performance and helped expedite extracting information from the database.

TABLE V
STORED PROCEDURES

Stored Procedure	Description
Insert_Accident_Data_SP	Handles insertion of a record into the Accident table. Based on the identifying attributes for road and police authority provided by the user, it fetches the corresponding primary keys for the Road and Authority details passed during creation and populates the table with road_id and authority_id.
Insert_Vehicle_Data_SP	Handles insertion of a record into the Vehicle table. It fetches the corresponding primary key for the Person Profile information passed from the CLI and populates the table with profile_id.
Insert_Casualty_Data_SP	Handles insertion of a record into the Casualty table. It fetches the corresponding primary key for the casualty's profile passed from the CLI and populates the table with profile_id.
Insert_Junction_Accident_Data_SP	Handles insertion of records into the specialized table, Junction_Accident.
Insert_Junction_Vehicle_Data_SP	Handles insertion of records into the specialized table, Junction_Vehicle.
Delete_Accident_by_ID_SP	Handles deletion of records from the Accident, Vehicle, Casualty, Junction_Accident, and Junction_Vehicle tables based on the accident_id passed during the call. If an exception occurs, the operation is rolled back.

VI. CLIENT INTERFACE (CLI)

Our Python CLI for the UK Accidents Database offers a range of functionalities to assist users in managing and analyzing accident data. Users can query the database, insert or delete data, plot accident locations, and obtain statistical information by selecting from a menu of options.

When designing the CLI, we considered the ideal client requirements, including the ability to query and input data. We also aimed to present statistical data in a visual format, such as graphs or maps, for easy interpretation by users.

It also provides additional functionalities, such as testing the connection to the database, listing all tables, showing user grants, showing table schema, and selecting rows based on a condition with LIKE. We incorporated these proposed functionalities into the final implementation of the CLI and the original functionalities.

List of CLI Operations:

- Testing the connection to the database
- Connecting to the database
- Listing all the tables in a database
- Showing current user grants
- Showing table schema
- Selecting all rows from a table
- Selecting rows from a table based on a condition
- Selecting rows from a table based on a condition with LIKE
- Inserting UK accident data
- Deleting UK accident data
- Plotting the location of accidents
- Getting the profile-wise count of accidents
- Getting the junction-wise count of accidents

To ensure the correctness of the CLI functions, we exhaustively tested all of them using the Python unit-test framework. This included testing functions to query and insert data, delete data, plot accident locations, and obtain statistical information.

Overall, the Python CLI for the UK Accidents Database provides a comprehensive set of tools for users to manage and analyze accident data. The CLI design considers both the ideal client requirements and the actual client-proposed functionalities, resulting in a robust and user-friendly implementation.

VII. STATISTICAL INFERENCES

Once we had created the relational schema for the database and populated it with relevant data, our next step was to perform static analysis to gain insights into certain characteristics and features of the database. To accomplish this, we wrote specific queries as part of a stored procedure utilized by the CLI tool to provide a user-friendly interface for interacting with the database. These static analysis queries helped us better understand the data within the database, including things like the distribution of data across tables, data types, and data ranges.

A. Locations of Accidents with its severity

Along with the accidents, we stored the details of the location where the accidents occurred. It might be important for the authorities to know which region in their patrol area had accidents and of what severity. This helps them to do the investigation at those places, and if certain things from the perspective of safety or sign boards are missing, then they can repair those in priority. Moreover, this data can be used to generate maps and visualizations that illustrate accident hotspots, which can help authorities plan and allocate resources more effectively. It can also be shared with the public to raise awareness about road safety issues and encourage safer driving behaviours.

Listing 1. SQL Query for retrieving locations of Accidents with its severity

```
with authorities_id as (
select authority_id from PoliceAuthority
where police_force
LIKE in_police_force and
local_authority_district
LIKE local_authority_district and
local_authority_highway
LIKE in_local_authority_highway)

SELECT latitude, longitude, accident_severity
FROM Accident
where authority_id in (select * from authorities_id)
and CAST(Month(datee) as char) Like in_month
and Year(datee)= in_year
and accident_severity LIKE accident_severity_option;
```

The CLI allows the user to plot accident locations based on the following six filter fields: police_force, local_authority_district, local_authority_highway, year, month, and severity type as shown in **Figure 5**.

The locations shown in the **Figure 6** below are displayed on the map based on the options entered in the CLI using this query.

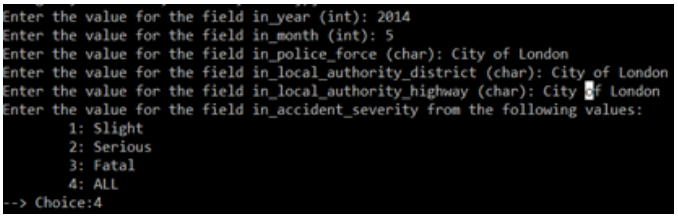


Fig. 5. CLI options to plot accident locations

Fig. 6. Output for Locations of Accidents with their severity

B. *Accidents count based on the profiles of the people*

We also show statistics of the profile of the person involved in the accident (as a casualty or as a driver of the vehicle). This information based on the grouping factor of sex and age group can help gain insights into the individuals' characteristics. Based on this, authorities might make changes to the assessments of driving tests. Also, based on the casualty type "Taxi/Private hire car occupant, Cyclist, Motorcycle 125cc and under rider or passenger, Pedestrian, Car occupant", we plot the count based on sex and age group.

Listing 2. SQL Query for retrieving accident count based on the profiles

```
with profile_wise_drivers as (
SELECT *
FROM ukfinal.vehicle as t1
inner join personprofile as t2 using (profile_id))

SELECT sex, age_group, count(*) as count
        from profile_wise_drivers
        group by sex,age_group
        having sex<>'Not_known';
```

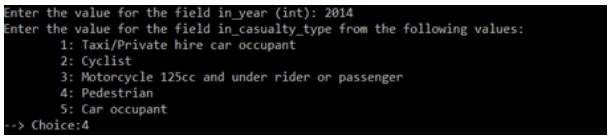


Fig. 7. CLI options to show accidents count based on the profiles

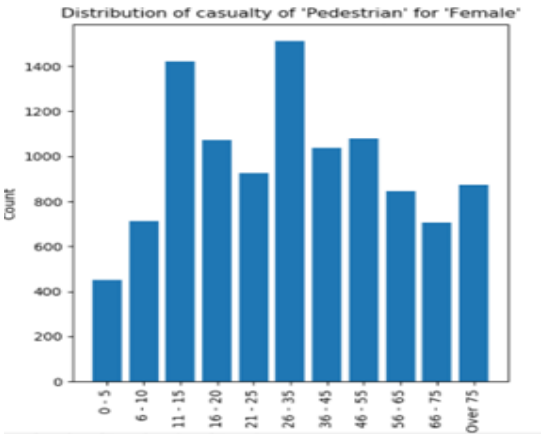


Fig. 8. Output of accidents count based on the profile

C. *Accident Counts by Junction Type and Severity under Jurisdiction of a particular police force*

We did this analysis to provide important insights into the distribution of accidents under the jurisdiction of a given Police Force. The predominance of T or Staggered junction as the most common junction type for accidents suggests the need for targeted interventions to improve safety at these locations. This may involve improvements in road design, such as better signage and markings, or increased enforcement of traffic rules and regulations.

Similarly, the high number of accidents categorized as slight severity underscores the need for continued efforts to improve road safety and reduce the number of accidents. This may involve a combination of measures, including improved infrastructure, education and awareness campaigns, and stricter enforcement of traffic laws.

We used an SQL query to retrieve data from the database for accidents that occurred under the jurisdiction of a given police force. The query selected accidents based on their severity and the authority ID of the police force, which was obtained from the policeauthority table in the database. We then joined the resulting dataset with the junction_accident table to obtain information on the junction type where the accidents occurred. Finally, we used the retrieved data to create a table showing the counts of accidents by junction and severity.

Listing 3. SQL Query for retrieving count of accidents by junction type and severity

```
with accidents_of_police_region as
(select accident_id, accident_severity
from accident where authority_id in
        (select authority_id
        from policeauthority
        where police_force LIKE in_police_force)),
junction_accident_of_police_region as
(select * from junction_accident
inner join accidents_of_police_region
using (accident_id))

select junction_detail as junction,
    accident_severity, count(*)
from junction_accident_of_police_region
group by junction_detail,accident_severity
order by junction_detail,accident_severity;
```

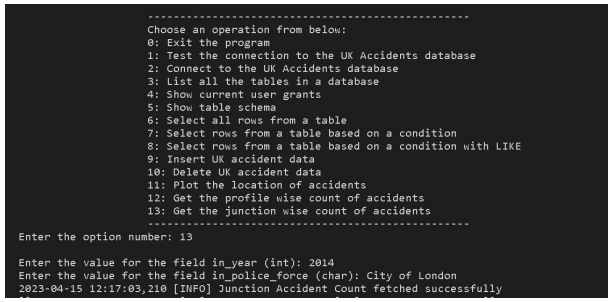


Fig. 9. CLI options to show accidents count of accidents by junction type and severity

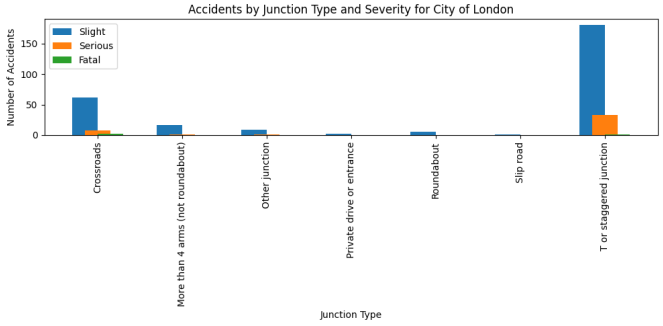


Fig. 10. Output of accidents count by junction type and severity

In conclusion, the clear database design of this dataset plays a vital role in facilitating the inference of numerous other insights that can be derived from it.

VIII. Testing

A. *Server-Side Testing*

For the server-side testing, we manually tested all the CRUD operations on the database. We used a cloned database to insert test data and tested for all possible scenarios that could cause data integrity issues, such as adding age to an age group where it does not belong. We aimed to ensure the correctness and integrity of the data stored in the backend MySQL database. By performing these tests, we could verify that the database functions as expected and can handle various data manipulation operations.

B. *Client-Side Testing*

For the client-side testing, we added unit tests using the Python Unit test module for all the functions of the Python CLI. We majorly used mock testing to separate client-side testing from server-side testing. Our focus was to ensure that the Python CLI performs all the CRUD operations as expected, with proper validation and error handling. Additionally, we conducted usability testing to check the ease of use of the CLI, including the command syntax, help menus, and error prompts.

IX. Data Mining

Accurate prediction of the severity of accidents is essential for efficient emergency response and traffic management. For the data mining part of this project, we investigated how the Random Forest algorithm could be utilized to predict accident severity using a set of carefully selected features. [4]

After a thorough analysis of the data fields in the Accident and Vehicles files, we have identified the following set of features that could play a crucial role in predicting the severity of accidents categorized as Slight (3), Serious (2), and Fatal (1):

- Did_Police_Officer_Attend_Scene_of_Accident
- Age_of_Drive
- Vehicle_Type
- Weather_Conditions
- Road_Surface_Conditions
- Light_Conditions
- Speed_limit
- Number_of_Vehicles
- Number_of_Casualties
- Engine_Capacity_(CC)
- Sex of Driver

During the preprocessing phase, we cleaned the dataset by removing entries with null values and those with entries marked as -1. This ensured that the data used for training the Random Forest model was high quality and free from missing or invalid values.

Furthermore, we observed that the dataset's distribution of accident severity categories was not balanced, as shown in **Figure 11**. We addressed the class imbalance by undersampling the majority class. This critical step helped us build a robust predictive model that is unbiased towards the majority class.

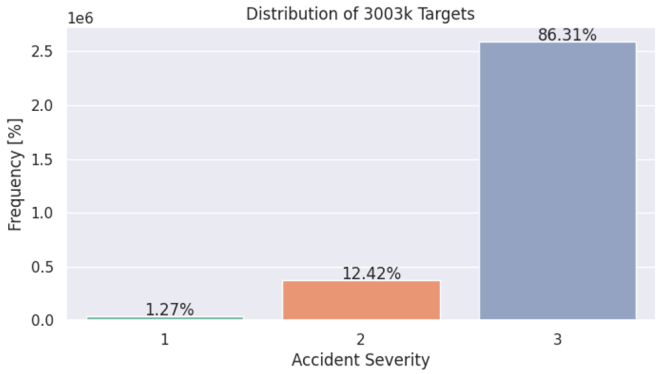


Fig. 11. Distribution of Accident Severity Categories in the Dataset

To account for the high number of Slight severity accidents in the dataset, we employed a sampling technique that assigned different weights to the data points. Specifically, we assigned weights of 0.2 to the Slight accidents and 0.8 to the Fatal and Serious accidents.

After sampling 40% of the data, the distribution was shown in the **Figure 12** below, which was uniformly distributed and almost balanced.

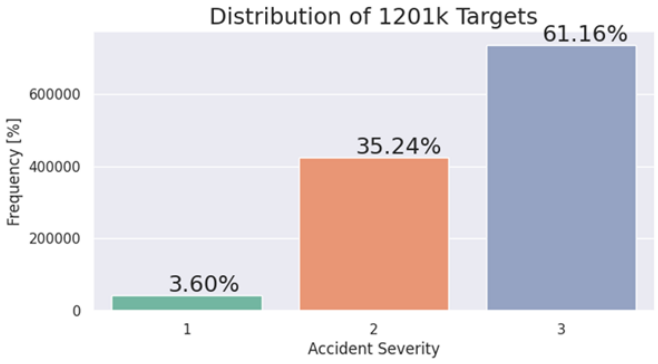
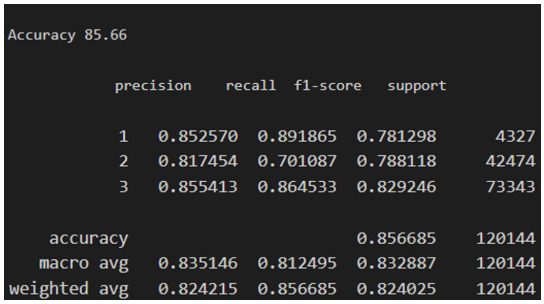


Fig. 12. Dataset distribution after sampling

Then, we utilized the Random Forest Classifier with 100 decision trees as the parameter for this task, as the features were already in categorical form and did not require encoding. The dataset was split into a training set comprising 90% of the data and a test set comprising 10% of the data.

After training the model, we evaluated its performance using various metrics. The accuracy of the model was found to be 85.66%, indicating that it was able to correctly predict the severity of accidents in 85.66% of cases (as shown in the **Figure 13**).

The model's performance is relatively better in predicting slight and fatal accidents than serious accidents, as reflected by the precision, recall, and F1-score values. The confusion matrix in the **Figure 14** shows how the model worked on predicting the test data into different classes.



Accuracy 85.66

	precision	recall	f1-score	support
1	0.852570	0.891865	0.781298	4327
2	0.817454	0.701087	0.788118	42474
3	0.855413	0.864533	0.829246	73343
accuracy			0.856685	120144
macro avg	0.835146	0.812495	0.832887	120144
weighted avg	0.824215	0.856685	0.824025	120144

Fig. 13. Model Accuracy

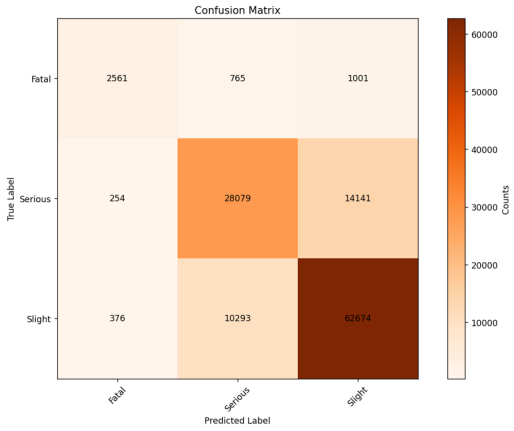


Fig. 14. Confusion Matrix

X. Submission

All project submissions have been uploaded on Gitlab in the repository here: https://git.uwaterloo.ca/arzanzar/db-656-uk-accident-data/-/tree/main/

We have organized all the code and relevant materials for our project into the following directories:

1) `CLASS_RELATIONAL_SCRIPTS/`: contains SQL scripts to create the relational model, populate data, and a script to pre-populate the `profile_data` table.
2) `Class Diagram/`: contains the image of the class diagram.
3) `ERD_Diagram/`: contains the image and draw.io file of the ERD diagram.
4) `RAWDATASCRIPTS/`: contains the following:
   a) `Data_Cleanup.ipynb`: a Jupyter notebook to preprocess the downloaded CSV files.
   b) `RawDataClean.sql`: files to create Raw data tables.
   c) `Raw_Data_Clean_Script.sql`: a script with commands to handle NULL and missing values in the RAW tables.
5) `StoredProcedure/`: contains all the stored procedures written.
6) `UK_ACCIDENTDATA_CLI/`:
   a) `accident_cli/`: contains the Python CLI code

b) `models/`: contains generic queries for each table in the database.
c) `queries/`: contains queries for statistical inferences.
d) `tests/`: contains unittests for the Python CLI.

7) `Accident_Severity_Analysis`:
a) `UK_Accident_Severity_Analysis.ipynb`: contains Random Forest algorithm for the accident severity prediction along with preprocessing steps.

REFERENCES

[1] Silicon99. (n.d.). *Dft-accident-data*. Kaggle. https://www.kaggle.com/datasets/silicon99/dft-accident-data.

[2] UK Department for Transport, *STATS19 Data Collection*, [Online]. Available: https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/995422/stats19.pdf.

[3] Department for Transport. (n.d.). *Reported road accidents, vehicles and casualties tables for Great Britain*. https://www.gov.uk/government/statistical-data-sets/reported-road-accidents-vehicles-and-casualties-tables-for-great-britain.

[4] DataCamp. (n.d.). *Random Forests Classifier in Python*. [Online]. Available: https://www.datacamp.com/tutorial/random-forests-classifier-python.