Driving Exams Analyzer — Project Report

1. Introduction

This project consists of developing a desktop application with **PyQt6** to manage and analyze official driving exam data from the Spanish **DGT (Dirección General de Tráfico)**. The application imports real CSV/TXT files, stores them in **SQLite**, supports dynamic filtering and visualization, and generates **PDF reports**.

The implementation was aligned with the provided rubric, focusing on correct data ingestion, coherent filtering, SQL-based grouping, visualization, and formatted reporting.

2. CSV Import and Database

2.1 Data source

The dataset comes from DGT public listings and is provided as delimited text/CSV files.

2.2 Database schema

A local SQLite database is used with two tables:

- **exam_results**: stores the imported exam records.
- **imported_periods**: stores which **month/year** periods have already been imported to prevent duplicates.

Key fields stored in `exam_results` include:

- Province (`desc_provincia`)
- Exam center (`centro_examen`)
- Driving school code (`codigo_autoescuela`)
- Section code (`codigo_seccion`)
- Month/year (`mes`, `anyo`)
- Exam type (`tipo_examen`)
- License (`nombre_permiso`)
- Passed/failed counts (`num_aptos`, `num_no_aptos`)

2.3 Duplicate import prevention

To avoid duplicated data, `imported_periods` includes a **UNIQUE(anyo, mes)** constraint. Before importing a file, the application checks whether that period is already present. If the period exists, the import is blocked.

This provides consistent prevention of re-importing the same month/year.

3. Filters and Data Query

3.1 Filter controls

The UI provides a coherent filtering block with:

- **Province** filter
- **Year** filter
- **Limit** selector (maximum number of rows shown in the table)

3.2 QCompleter

The Province control includes **QCompleter** to enable fast searching of provinces. The completer is configured as case-insensitive for usability.

3.3 SQL-driven filtering

Filtering is implemented by building SQL queries dynamically with WHERE conditions depending on selected values. The **limit** is applied directly in SQL to ensure results are constrained consistently.

3.4 Grouping for analysis

For charts, data is grouped in SQL using `GROUP BY` and aggregated with `SUM(...)`. This ensures the chart reflects filtered results and database grouping, not ad-hoc Python grouping.

4. Table View

4.1 Table presentation

Filtered results are displayed in a **QTableView**. After applying filters, the table is refreshed to show the new result set.

4.2 Usability features

The table includes:

- Sorting enabled
- Alternating row colors

- Row selection behavior

5. Charts

5.1 Chart configuration

A Matplotlib chart is displayed in the “Chart” tab. The chart is built from grouped SQL results.

5.2 Meaningful visualization

The visualization uses a **stacked bar chart**:

- **Aptos (passed)**
- **No aptos (failed)**

This provides a clearer analytical view than a simple single-series bar chart.

5.3 Filter synchronization

When filters change, the chart is rebuilt so it always reflects the current selection.

6. PDF Reports

6.1 Table report

The table can be exported to PDF using ReportLab’s **Platypus** table system, which generates:

- A real table grid layout
- Styled header row
- Automatic pagination
- Metadata (generated date and applied filters)

6.2 Chart report

The chart can be exported to PDF by saving the Matplotlib figure to an image and embedding it into a PDF page.

7. UI Organization

7.1 Layout

The UI is organized for clarity:

- Filters at the top
- Results shown in tabs (Table / Chart)

- Export buttons located close to their corresponding output

7.2 Programmatic UI decision

The UI is created in Python (not Qt Designer) to simplify dynamic updates of table and chart based on filters.

8. Code Structure and Modularity

The project follows a modular structure:

- `services/database.py` — database schema, queries, grouping, limit
- `services/csv_importer.py` — import logic and duplicate checks
- `services/charts.py` — Matplotlib chart creation
- `services/reports.py` — PDF export (table and chart)
- `main.py` — UI/controller logic (filters, refresh, export actions)

This separation improves maintainability and matches an MVC-style organization.

9. Completeness and Stability

The final application provides:

- Reliable CSV/TXT import
- SQLite persistence
- Month/year re-import protection
- Coherent filters + QCompleter
- SQL grouping and limit
- Table + chart visualization
- PDF export for table and chart

The application behaves consistently and produces results that match the user's current selection.

10. Conclusion

The Driving Exams Analyzer meets the main functional requirements by importing real DGT data, preventing duplicates, enabling SQL-driven filtering and grouping, visualizing results, and exporting formatted PDF reports. The code is modular and the interface is clear and usable.