Processing and Analysis of Belgian Car Data

Data Engineering Capstone Project

By Stef L.

2024

Table of Contents

Preface

As I reach the culmination of my degree in Big Data and Business Intelligence, alongside my Bootcamp in Data Engineering, I am proud to present my final capstone project: the development of an end-to-end data pipeline. This project is the result of months of rigorous study, practical application, and hands-on experience with cutting-edge tools and technologies that have shaped my understanding of the data engineering landscape.

The capstone project gave me the opportunity to integrate a range of tools and techniques I have learned throughout my academic journey. Utilizing Python for managing the ETL process, Mage-AI for orchestrating workflows, and Apache Spark for handling batch processing, I gained a comprehensive view of data engineering practices. My project also involved leveraging Google Cloud for hosting the data lake and data warehouse, automating cloud infrastructure with Terraform and Docker, and transforming data in BigQuery using dbt within the Kimball data modeling framework. To cap it off, I designed business reports using Power BI, delivering valuable insights from the data.

This project has been an extraordinary learning experience that solidified my understanding of modern data engineering practices, and I am eager to apply these skills in real-world projects moving forward.

Introduction

This project aims to analyze the interaction between various municipal-level characteristics and the degree of car ownership in Belgium. To achieve this, we have developed a comprehensive workflow that facilitates the extraction, transformation, and analysis of relevant data.
The workflow begins with the extraction of data from online sources. This data is subsequently ingested into our Data Lake.
Next, we perform the transformation and loading of this data. Using batch-processing and transformations, after which we load the data into our Data Warehouse.

Finally, we connect an interactive dashboard to the Data Warehouse. This allows us to generate insightful reports and conduct detailed analyses of the data.
The primary research question guiding this project is: How do different municipal-level characteristics interact with the degree of car ownership? Key elements of our analysis include population density, province, city area size, the count of sub-municipalities, and household types. Our data sources for this analysis include car ownership data from Statbel (statbel.fgov.be) and city demographic information from Wikipedia tables.

Project Results

The primary objective of this project is to apply the knowledge gained throughout my learning journey to build a complete end-to-end data pipeline, culminating in the development of a dashboard with two visual tiles. The task involves selecting a relevant dataset, creating a pipeline to process and store the data in a data lake, and then building another pipeline to move the data from the lake to a data warehouse. The data must be transformed within the warehouse to prepare it for visualization. Students will need to decide whether to implement a batch or streaming pipeline depending on their use case, with batch pipelines handling periodic data processing and streaming pipelines focusing on real-time data ingestion. A range of technologies is available for each phase of the pipeline, including cloud platforms (AWS, GCP), orchestration tools (Airflow, Prefect), and data warehouses (BigQuery, Redshift).

Tools Used

Overview

Languages: Python, SQL
IDE: Jupyter Notebook, Visual Studio Code
Workflow: Mage-AI
Transformations: PySpark, dbt Cloud
Infrastructure: Docker, Terraform
Storage: Google Cloud Storage (Data Lake), Google BigQuery (Data Warehouse)
Cloud environment: Google Cloud
Other tools: draw.io, Bash, MS Excel, MS PowerBi

Rationale for Tool Usage

In designing and implementing this data pipeline, I carefully selected tools that best aligned with the project’s objectives and requirements, balancing ease of use, scalability, and integration. Below is a rationale for each tool’s selection and role in the process.

Python and SQL form the core of this project due to their fundamental roles in data manipulation and transformation. Python offers powerful libraries for handling both structured and unstructured data, making it versatile across different data types. SQL, on the other hand, provides an efficient way to interact with databases, making it invaluable for data querying and manipulation, especially in structured environments like data warehouses.

To facilitate coding and development, I used Jupyter Notebook and Visual Studio Code as primary interfaces. Jupyter Notebook excels in testing Python code and data transformations due to its real-time feedback, making it easier to experiment and debug. Meanwhile, Visual Studio Code offers a more streamlined development environment with built-in GIT integration, support for various extensions, and a robust ecosystem for building scalable, production-level code.

For workflow orchestration, I chose Mage-AI, an emerging open-source tool, instead of the more traditional Apache Airflow. Mage-AI's simplicity and ease of use made it an ideal choice for smaller teams and projects that do not require complex orchestration capabilities. This helped streamline the pipeline building process, making it more accessible and easier to maintain.

For data transformation, I integrated PySpark and dbt Cloud. PySpark is well-suited for handling large-scale batch processing and supports user-defined functions (UDFs), making it highly flexible for both structured and unstructured data processing needs. dbt Cloud, on the other hand, was used for data modeling and SQL workflow scheduling. Its user-friendly interface and SQL-based transformation workflows made it perfect for managing the transformations required in Google BigQuery, our data warehouse.

Terraform played a key role in managing the cloud infrastructure. By automating infrastructure deployment, it eliminated the need for manual configuration through a GUI, reducing errors and ensuring consistency. Docker was used to containerize the pipeline, creating isolated environments to run the pipeline independently of local configurations, a best practice that Mage-AI also recommends for pipeline execution.

For cloud services, I selected Google Cloud as the backbone of our infrastructure. Google Cloud Storage served as the staging area for the Data Lake, efficiently handling unstructured data, while Google BigQuery was utilized for the Data Warehouse, storing structured data in analysis-ready tables, enabling smooth querying and reporting.

Additional tools complemented the pipeline. draw.io was used to design workflow diagrams, providing a clear visual representation of data flows. Bash scripts were employed for terminal-based tasks such as environment setup and automation. MS Excel proved useful for quick data exploration and validation, while MS Power BI was the tool of choice for final data visualization, offering intuitive and insightful business reporting dashboards.

In summary, the choice of tools was guided by the specific needs of the project, ranging from data transformation and workflow orchestration to cloud infrastructure management and data visualization. These technologies collectively formed a robust and scalable data pipeline that met the project's goals efficiently.

Project Activities

1. Identifying the Sources

Tools: Bash, Excel

The data collection phase began by gathering information from multiple online sources. First, municipality-level data was scraped directly from Wikipedia using a custom script. This data, available at https://nl.wikipedia.org/wiki/Tabel_van_Belgische_gemeenten, provided essential demographic and geographic information at the municipality level.

In addition, we sourced detailed data on car ownership by municipality and household type from Statbel, Belgium's official statistics agency. This dataset, available in an Excel format at https://statbel.fgov.be/sites/default/files/files/opendata/Aantal%20wagens%20volgens%20huishoudtype%20per%20gemeente/TF_CAR_HHTYPE_MUNTY.xlsx, provided key insights into vehicle distribution across regions and household categories.

Once the data was collected, we used Bash scripts for basic file handling tasks, such as downloading the datasets and performing simple row counting to ensure data integrity. For a more detailed exploration of the datasets, we utilized MS Excel, which allowed us to inspect the structure, identify any potential issues (such as missing values), and verify that the data was ready for further processing and analysis. This initial data exploration laid the foundation for the subsequent stages of transformation and integration into the data pipeline.

2. Designing the Workflow

Tools: draw.io

The workflow for the data pipeline follows a structured sequence of steps to ensure seamless data extraction, transformation, loading, and analysis.

- **Extraction:** Data is first gathered from two primary online sources: Wikipedia (for municipality-level data) and Statbel (for car ownership by municipality and household type). This data is then loaded into a Data Lake housed in Google Cloud Storage buckets.
- **Transformation & Ingestion:** The data is retrieved from the Data Lake and batch processed using Apache Spark to handle large-scale data transformations. Once processed, the transformed data is loaded into the Data Warehouse, which is hosted on Google BigQuery. This step ensures that the data is structured and ready for further transformations.
- **Planning:** Both the extraction and transformation/loading steps are orchestrated and scheduled using Mage-AI, which acts as the workflow manager. Mage automates and monitors the workflows, ensuring that the pipeline runs smoothly and efficiently.
- **Additional Transformations:** Once the data is stored in the Data Warehouse, further transformations are carried out using dbt Cloud. These transformations focus on refining the data and preparing it for more advanced analysis, using SQL-based modeling techniques.
- **Analysis:** The final stage connects MS Power BI to the Data Warehouse, allowing for data visualization and analysis. Power BI generates dynamic reports and dashboards, providing key insights into the data that can be used for business decision-making.

For a visual summary of the entire process, please refer to the *workflow_design.png*. This diagram highlights the key stages of the workflow and the tools used at each step.

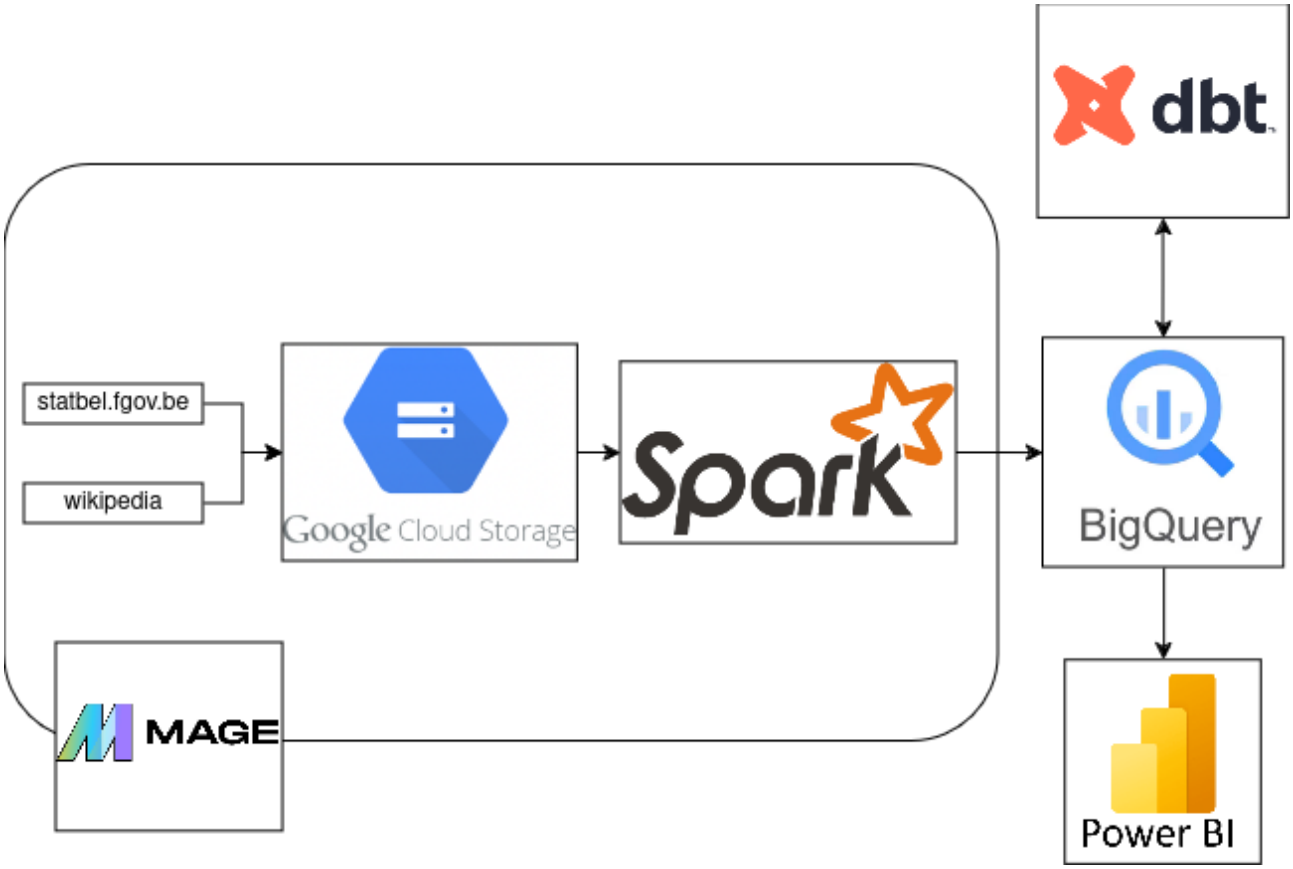


Figure 1: workflow_design.png

3. Managing the Data Workflow, Data Extraction, and Data Ingestion into Data Lake

Tools: Python, Docker, Mage-AI, Google Cloud Storage

The process begins with configuring the necessary infrastructure for data extraction and ingestion. First, a bucket is created in Google Cloud Storage, which will serve as the repository for the raw and transformed data. To manage access to this cloud storage, a service account is created in Google's IAM Admin console, with the credentials exported as a JSON file. This service account enables secure interactions between the pipeline and the cloud resources throughout the project.

With the infrastructure in place, the next step is to set up a Mage-AI instance locally. A template provided by Mage is used to initiate the setup, which is then customized by editing several configuration files, including the *Dockerfile*, *.env* file, and *requirements.txt.* These modifications tailor the environment to the specific needs of the project. The setup is finalized by running the commands *docker compose build and docker compose up* in the terminal. This launches the Mage environment, which can then be accessed through a local interface at http://localhost:6789.

Once the environment is live, a data pipeline is generated and the *IO_config.yaml* file is configured. This file is essential for integrating the pipeline with Google Cloud services. The configuration includes defining the path to the Google service account JSON file and removing any unnecessary Google Cloud settings to streamline the connection. With the pipeline ready, the next stage involves setting up blocks within Mage-AI to handle data extraction, transformation, and exportation.

The first block, a data loader script written in Python, is responsible for extracting data from Wikipedia. This script, named *wikipedia_api.py*, imports necessary libraries such as pandas, BeautifulSoup, and Mage's data loading functions. The script fetches the HTML content from the Dutch Wikipedia page titled "Tabel_van_Belgische_gemeenten" and parses it using BeautifulSoup for easier data extraction. A custom function is included to convert Belgian-style numeric formats (e.g., 1.234,56 to 1234.56), ensuring consistency across the dataset. After cleaning the data, the script converts it into a CSV file, which is stored in a pandas DataFrame for further processing.

Once the data is loaded, it undergoes transformation in the next block, handled by a script called *wikipedia_transform.py*. This transformation script begins by importing necessary libraries like pandas and io. It cleans the dataset by removing irrelevant columns, such as Index.1 through Index.4, and fixes any issues with the table headers. The cleaned data is then converted back into a CSV format and reloaded into a pandas DataFrame, ensuring it is ready for export.

The final step in processing the Wikipedia data involves exporting the transformed DataFrame to Google Cloud Storage. This is done using a data exporter script, *wikipedia_to_gcs.py*, which specifies the target storage path within the Google Cloud bucket. The script defines the path to the configuration file and uses GoogleCloudStorage libraries to upload the DataFrame to the bucket, storing it as a CSV file at raw/wp_municipality_data.csv.

In addition to the Wikipedia data, a separate data loader script, *statbelcar_api.py*, is used to retrieve data from Statbel, Belgium’s official statistics website. This script downloads an Excel file containing information on car ownership by household type for each municipality. The script reads the relevant data from the Excel file and stores it in a pandas DataFrame for processing.

Once the Statbel data is ready, it is also exported to Google Cloud Storage using another data exporter script, *statbelcar_to_gcs.py*. Similar to the Wikipedia exporter, this script specifies the storage location in the Google Cloud bucket (raw/statbelcar_data.csv) and uploads the cleaned data to the cloud.

Finally, with both data sources processed and transformed, the pipeline is executed, and the blocks are run. The result is a set of cleaned and structured CSV files, stored in Google Cloud Storage. This completes the data extraction and ingestion process, with the data now residing in the Data Lake, ready for further analysis and transformation. The entire process ensures that the data is accurately captured, transformed, and securely stored, forming the foundation for the subsequent stages of the data pipeline.

4. Transforming Data Using Batch Processing and Loading into Data Warehouse

Tools: Python, Spark, Google BigQuery, Google Cloud Storage, Mage-AI

First, the batch processing workflow is designed and tested in a Jupyter Notebook, documented in the file *batch_processing.ipynb*. This notebook outlines the steps required for the batch processing pipeline, enabling testing and refinement before full implementation. Once the workflow is verified, the notebook is converted into a Python script using the Bash command *jupyter nbconvert --to=script batch_processing.ipynb*. This command produces the *batch_processing.py* file, which is then further adapted to integrate with Google Cloud Storage (GCS) and BigQuery.

To connect the Python script with Google Cloud Storage and enable data writing to BigQuery, necessary modifications are made within the *batch_processing.py* file. This includes adding the appropriate authentication credentials and adjusting the script to use GCS and BigQuery as output targets for the transformed data.

Following this, we set up a Mage-AI instance with PySpark installed to handle distributed data processing. The required PySpark Dockerfile is sourced from Mage's GitHub repository, and the image is built using the command *docker build -t mage_spark .. Once the Docker image is successfully built, the container is run with the command docker run -it --name mage_spark -e SPARK_MASTER_HOST='local' -p 6789:6789 -v $(pwd):/home/src mage_spark /app/run_app.sh mage start spark_project*. This launches the Mage environment with PySpark enabled, allowing the batch process to operate at scale.

Within the *metadata.yaml* file of the project, we configure the necessary Spark session parameters. These configurations ensure the Spark session is properly connected to GCS and BigQuery. The *batch_processing.py* script is then integrated into Mage as a custom block to form part of the pipeline.

The custom block within Mage performs the following key tasks:

- **Imports Libraries:** Essential libraries such as PySpark for distributed data processing, pandas for data handling, and various PySpark functions are imported.
- **Conditionally Imports Decorators:** The block checks and imports Mage's custom and test decorators if they are not already present.
- I**nitialize Spark Session:** A new Spark session is created with configurations enabling the use of Google Cloud Storage connectors and authentication credentials for accessing GCS and BigQuery.
- **Read Input Data:** The script reads two CSV files, one for car data and one for municipality data, from GCS into Spark DataFrames.
- **Define Schemas:** Specific schemas for the car and municipality data are defined to ensure the correct structure is applied during the data processing.
- **Clean and Transform Data:** Unwanted characters, such as asterisks (*), are removed using regexp_replace. Additional calculated columns are created, such as pop_perc_increase_2024v2000 to measure population percentage increases. Translations for household types (e.g., from French to English) are also applied, and user-defined functions (UDFs) are used to add new columns, including has_car and no_car to indicate car ownership status.

- **Write Data:** The cleaned and transformed data is written back to Google Cloud Storage in Parquet format, ensuring efficiency and scalability for future use. The data is also exported to BigQuery tables for further analysis.
- **Configure GCS and BigQuery:** Proper authentication and project configurations are ensured to facilitate seamless data exchange between GCS and BigQuery.

Finally, the batch processing pipeline is integrated with the existing data ingestion pipeline within Mage, forming a unified system for handling data from extraction to processing. The entire setup is stored in the *mage_load_and_spark* folder, which consolidates both the data ingestion and the Spark-based batch processing workflows. For a visual representation of this combined pipeline, the diagram is provided in *mage_pipeline.png*.
This approach streamlines the batch processing workflow while leveraging the scalability of PySpark and the cloud infrastructure provided by Google Cloud. By integrating both processes in Mage, the system ensures an efficient end-to-end data pipeline.

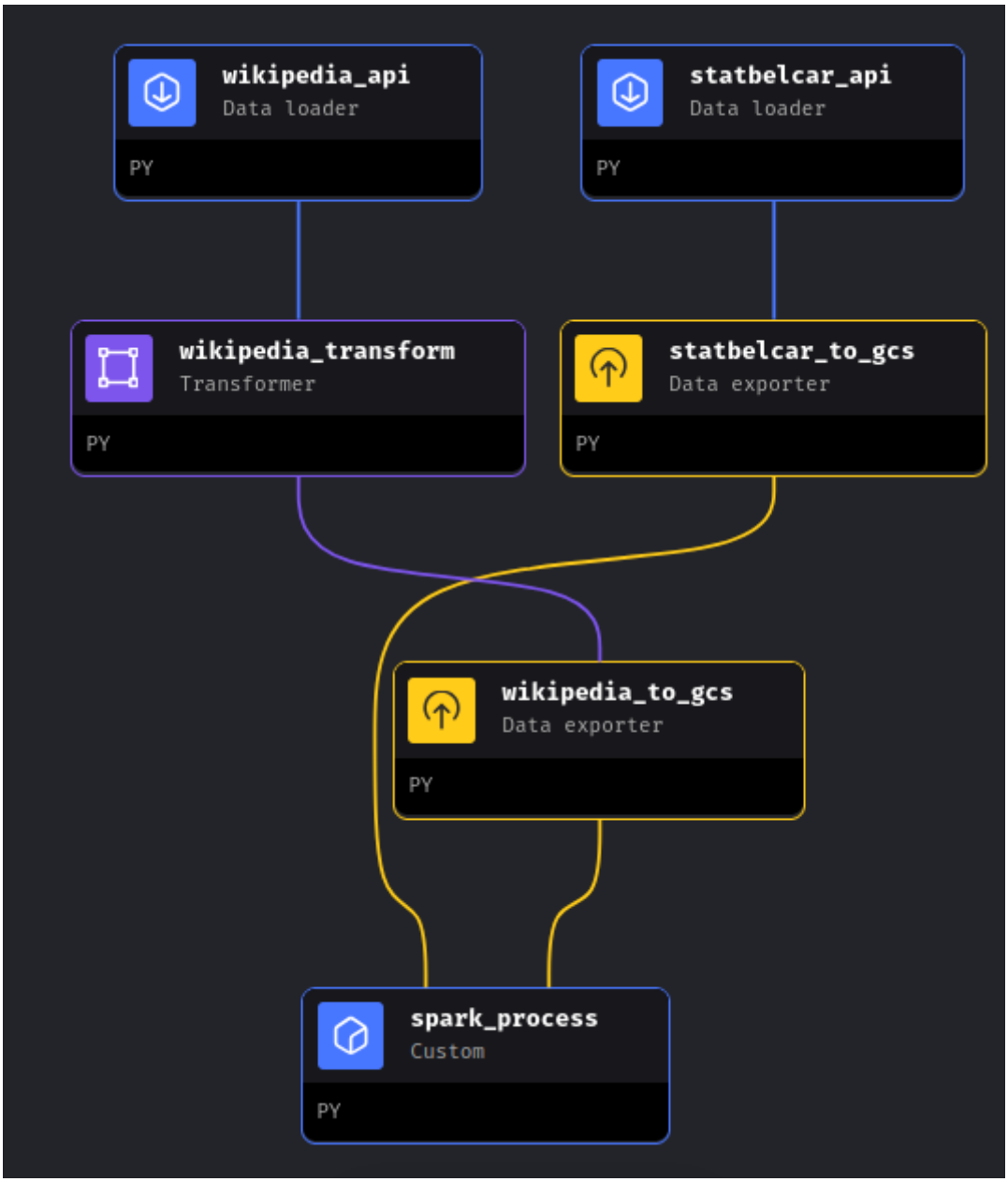


Figure 2: mage_pipeline.png

5. Deploying the Unified Mage Pipeline to the Cloud

Tools: Terraform, Google Cloud, GIT, Mage-AI, Docker

Deploying the Custom-Built Dockerfile to the Cloud

The deployment of the unified Mage pipeline to the cloud begins with pushing the custom-built Dockerfile for Mage to Google Artifact Registry using the GCloud CLI. To start this process, I first navigate to the directory containing the Dockerfile for the Mage pipeline. After locating the Dockerfile, I authenticate my Google Cloud account by running the command *gcloud auth login*. This step ensures that I have the necessary permissions to interact with Google Cloud resources.

Once authenticated, I set the project to the appropriate Google Cloud project using the command *gcloud config set project carprojectbelgium*. This command directs all subsequent operations to the specified project. Next, I create a new Docker repository in Google Artifact Registry. I do this by executing the command to create a repository called my-new-repo, specifying the repository format as Docker and the location as us-central1. This new repository will serve as the storage for my Docker images.

To enable Docker to interact with Google Artifact Registry, I authenticate Docker with the registry using an access token. I achieve this by running the command that prints the access token and then logs into Docker with it. After completing the authentication, I build the Docker image using the command *docker build -t mageai-customsparkfile:latest .*, which compiles the Dockerfile into an executable image.

Following the build, I tag the Docker image for the newly created repository. This is accomplished by using the docker tag command to label the image appropriately. Finally, I push the tagged image to the Google Artifact Registry with the command *docker push us-central1-docker.pkg.dev/carprojectbelgium/my-new-repo/mageai-customsparkfile:tag1*. This action uploads the image to the cloud, making it available for use.

Deploying the Mage Pipeline via Terraform

The next phase of the deployment involves using Terraform to provision the necessary infrastructure for the Mage pipeline. I begin this process by enabling the Service Usage API on the Google Cloud Platform (GCP). Once the API is enabled, I authenticate the application defaults by executing the command *gcloud auth application-default login*. After ensuring that my application defaults are set up correctly, I navigate to the directory that contains the Terraform files, specifically the *terraform_deploy* folder.

In the Terraform main file, I configure the Google Cloud provider using credentials from a local JSON file named *mage_planner_key.json*. This configuration specifies the GCP project as carprojectbelgium and sets the region to us-central1, where all resources will be created. Subsequently, I provision a Compute Engine instance named mageai-instance in the us-central1-a zone. This instance uses an e2-medium machine type and is initialized with a Debian 11 boot disk.

To manage resources effectively, I attach a Google service account with Cloud Platform access to the virtual machine. This service account, identified as mage-planner@carprojectbelgium.iam.gserviceaccount.com, allows the VM to interact with other Google Cloud services. I also configure the instance to use the default network with external access and tag it as http-server to facilitate firewall rules for incoming traffic.

To ensure the Docker environment is correctly set up, I include a startup script in the Terraform configuration that installs Docker and configures it to start automatically upon booting the VM. This script also sets up Docker to authenticate and pull images from Google Artifact Registry.

Once the Terraform configuration is complete, I initialize Terraform with the command terraform init. This step prepares Terraform to manage the resources specified in the configuration. After initialization, I apply the Terraform configuration using the command terraform apply, which creates the necessary resources in Google Cloud.

After the virtual machine is provisioned, I establish an SSH connection to it in order to add the service account JSON file, ensuring the VM can access the required Google Cloud services. I also set up Git synchronization with the *mage_load_and_spark* folder in my GitHub repository, following the guidelines provided in the Mage-AI documentation.

Once everything is configured, I utilize Mage-AI to plan and schedule jobs as needed. After executing the jobs, the data is processed and stored in the Google BigQuery Data Warehouse. Once the processing is complete, I clean up resources by destroying the virtual machine and associated infrastructure with the command terraform destroy. This final step helps in managing costs and maintaining a clean environment.

Through this comprehensive deployment process, the unified Mage pipeline is successfully established in the cloud, leveraging Google Cloud's capabilities for scalable and efficient data processing.

6. Designing the Data Warehouse and Executing Additional Transformation Jobs

Tools: SQL, DBT, Google BigQuery, GIT

Defining the Data Model and Setting Up dbt

The data modeling process begins with defining the grain of our data model and designing the fact and dimension tables using the Kimball model. The grain is established at the level of a unique combination of cardata_year, household_type_en and city_id. Each record in our tables corresponds to a distinct pairing of these three attributes. This means that for every type of household within a given municipality, for each year, there exists a single record representing that combination. To uniquely identify each instance, a surrogate key is generated through our dbt model. Please see *table_design.svg* for a visual depiction of the tables.

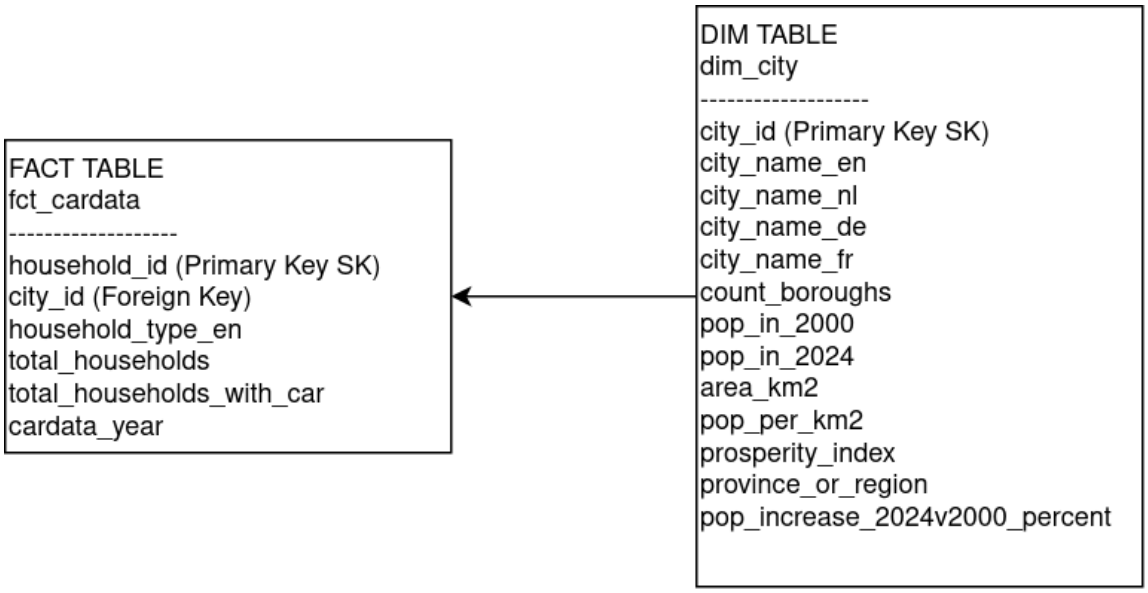


Figure 3: table_design.svg

Next, I set up the dbt Cloud project to manage our data transformation workflows effectively. To accomplish this, I create a subdirectory and initialize the dbt project within it. Following this, I configure the *dbt_project.yml* file to define essential project settings and specifications, ensuring that the project is organized and operates smoothly.

After establishing the project framework, I develop models and source files, which can be found in the dbt folder. One key file, *dbt/models/staging/src_carprojectbelgium.yml*, contains the source definition. This source is named carprojectbelgium, which references a GCP BigQuery dataset called belgium_cardata. Within this dataset, I specify two critical tables:

- spark_cardata2017-2023: This table includes car ownership data in Belgium from 2017 to 2023, processed using Spark.
- spark_municipalitydata: This table contains information related to towns and cities in Belgium, also processed with Spark.

I then focus on transforming the data by creating SQL files for staging the data. The file *dbt/models/staging/stg_carprojectbelgium__spark_cardata2017-2023.sql* is designed to select and transform specific columns from the car ownership data. This includes trimming city names in multiple languages by removing any text within parentheses and eliminating leading and trailing whitespace.

The file *dbt/models/staging/stg_carprojectbelgium__spark_municipalitydata.sql* follows a similar approach, retrieving and transforming several fields from the city source. For instance, a CASE statement is used to standardize municipality names for consistency, while the area of the municipality is rounded to two decimal places. Additionally, the population density is cast as an integer, and other demographic and geographic attributes are selected.

In the *dbt/models/intermediate/dim_city.sql* file, I configure a model to be materialized as a table, meaning the results will be physically stored in the database. This file retrieves various fields, such as a unique identifier for each city generated using ROW_NUMBER(), and municipality names in Dutch, English, French, and German. These attributes are gathered through a LEFT JOIN with car data to ensure that the city names align correctly with the relevant data.

Another important component is the *dbt/models/intermediate/fct_cardata.sql* file, which also materializes the results as a table. This file contains a Common Table Expression (CTE) that aggregates data from the car data by calculating total households and households with at least one car, grouped by year, municipality name in French, and household type. The main SELECT statement retrieves essential fields, including a unique identifier for each household, the city ID, and the year of the car data. It also calculates the percentage of households with cars, providing valuable insights into car ownership trends.

Additionally, I set up two data marts designed to facilitate user-friendly analysis of specific dimensions: one with a focus on car data by city from the most recent year recorded, and another for car data by household type from the same period. I create the file *dbt/models/mart/mart_latestyear_by_city.sql*, which materializes the results as a table and aggregates car data grouped by city for the most recent year. This file retrieves fields such as the most recent year of car data, total households, and the percentage of households with cars. A LEFT JOIN is performed to combine city demographics with car ownership statistics, ensuring clarity in the results.

Another model, *dbt/models/mart/mart_latestyear_by_householdtype.sql*, is designed to be materialized as a table as well. This file retrieves information about total households for each household type and the number of households that own at least one car. It also calculates the percentage of households with cars, filtering the data to include only records from the most recent year and grouping the results by household type for insightful analysis.

After setting up the models, I establish a production environment in dbt, to ensure a consistent data workflow. This step is crucial for managing how the data is transformed and loaded. Following this setup, I configure jobs and triggers using the production environment and dataset to automate the data processing tasks.

After configuring these jobs, I run them to execute the transformation processes. A visualization of the dbt job execution can be found in the *dbt_pipeline.png* file on the next page.

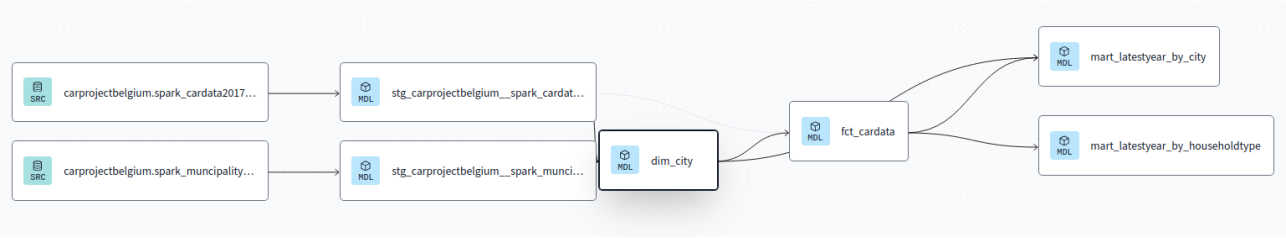


Figure 4: dbt_pipeline.png

Optimizing the Data Warehouse Tables

Finally, I focus on optimizing the Data Warehouse tables for better performance. To enhance storage efficiency, I implement partitioning and clustering strategies. **Partitioning** involves dividing a table into smaller segments based on a specified column, such as a date or an integer range. This technique reduces the amount of data scanned by queries, thereby improving performance and reducing costs.

Clustering, on the other hand, sorts data within partitions or across the entire table based on specific column values. This practice enhances query performance for filters or joins by grouping related rows together. For instance, partitioning is ideal for time-based data; in this case, I would partition the fct_cardata table by the cardata_year column. Meanwhile, clustering works effectively on columns frequently used for aggregation, such as city_id and household_type_en.

To put these optimization strategies into practice, I will execute a SQL query that creates a partitioned and clustered version of the fct_cardata table. The query will look like this:

```
CREATE TABLE dwh_belgium_cardata.fct_cardata_partitioned
PARTITION BY DATE(cardata_year)
CLUSTER BY city_id, household_type_en AS
SELECT * FROM dwh_belgium_cardata.fct_cardata;
```

This query will enhance the performance of our Data Warehouse by efficiently managing data storage and retrieval, ensuring that our analyses of Belgian car data are both effective and insightful.

7. Data Visualization and Reporting

Tools: MS PowerBi

To finish off our data process we create take advantage of dashboard tools to explore our data effectively. The first step in the data visualization and reporting process involves connecting the BigQuery data to Power BI. This integration allows us to access and analyze the car ownership data from our BigQuery warehouse directly in Power BI.

Once the connection is established, I begin designing the dashboard. The dashboard layout consists of four key panels, each providing unique insights into car ownership trends across Belgium. You can refer to the specific PowerBI design in the file *dashboard_cardata.pbix*.

- **First Panel:**
  The first panel focuses on car ownership levels by province and by household type. To visualize this, I create bar charts that display the number of households that own cars across different provinces, breaking down the data further by household type (e.g., single households, family households, etc.). This view helps provide a geographical and demographic perspective on car ownership patterns.
- **Second Panel:**
  The second panel uses scatter plots to explore the relationship between car ownership and two specific variables: municipality size and population density. The first scatter plot shows how car ownership varies with the size of municipalities, while the second plot examines the correlation between car ownership and population density. These visualizations help us identify whether certain municipal characteristics influence car ownership trends.
- **Third Panel:**
  The third panel presents a line graph that illustrates the relationship between car ownership and the prosperity index of different municipalities. By mapping car ownership levels against the prosperity index, we can investigate whether wealthier municipalities tend to have higher levels of car ownership.
- **Fourth Panel:**
  The fourth panel presents a line graph that illustrates the degree of car ownership over the years for different provinces. This visualization reveals that, while car ownership has remained relatively stagnant in most provinces and regions, Brussels stands out as an exception. The degree of car ownership in Brussels, which is already the lowest compared to other regions, has been actively declining over time. This trend suggests a shift in transportation preferences or policies in Brussels that contrasts with the rest of the country.

After finalizing the dashboard design, I export the dashboard as a PDF file for easier distribution and reporting purposes. This PDF version of the dashboard can be shared across teams and stakeholders to provide a snapshot of the insights gathered from the data.

You can refer to *dashboard_cardata_print.pdf* for the final, printable version of the dashboard, which is attached in the last four pages of this report.

Analysis

The analysis of car ownership across various municipalities reveals several significant patterns. As shown in the bar and line graphs, most provinces exhibit a relatively similar level of car ownership, with the notable exception of the Brussels region, which has a markedly lower degree of car ownership. This discrepancy can be attributed to the region's densely populated urban landscape, where the availability of public transport and limited space likely reduce the need for private vehicles.

Furthermore, the second bar graph highlights that car ownership is much higher among couples, both with and without children, compared to non-couples. In contrast, one-person households are the least likely to own a car, and even single-parent households exhibit significantly lower car ownership rates compared to couples.

The scatter plot analysis adds another dimension, demonstrating that larger towns and cities, in terms of area, tend to have higher rates of car ownership, whereas higher population density correlates with lower car ownership levels.

Lastly, the third panel suggests a socioeconomic factor in car ownership trends, showing that regions with lower prosperity indices also tend to have reduced car ownership rates. The combination of factors—urban density, household composition, town size, and economic prosperity—paints a comprehensive picture of the variables influencing car ownership across different regions.

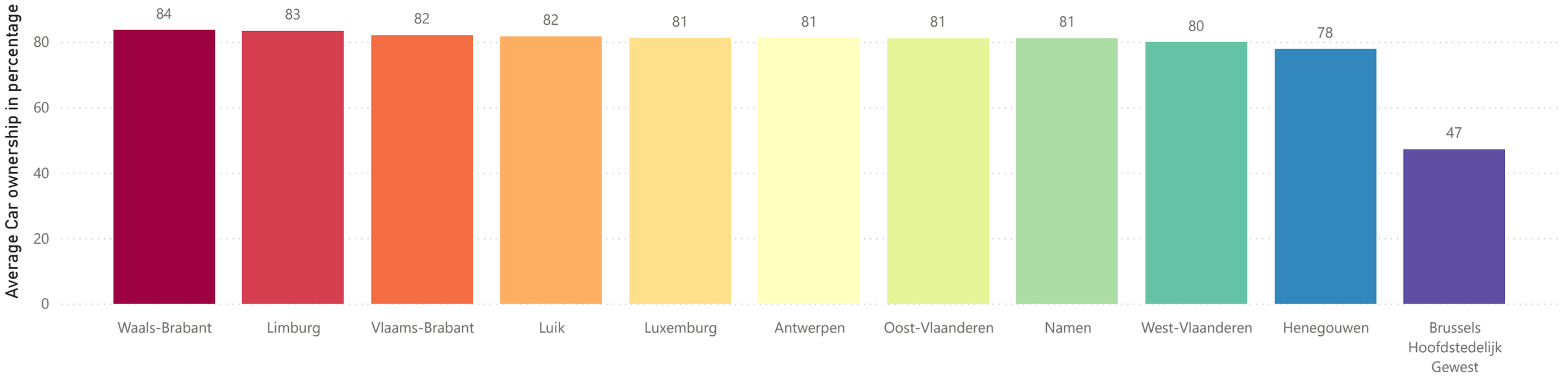
Conclusion

This project explored how various municipal characteristics relate to car ownership levels in Belgium, offering useful insights into regional transportation patterns. Using a data pipeline and a range of widely-used data processing tools, we analyzed the impact of factors like population density, household composition, and city size on car ownership. The findings show that car ownership tends to be lower in densely populated areas like Brussels and higher among couples, particularly in larger towns.

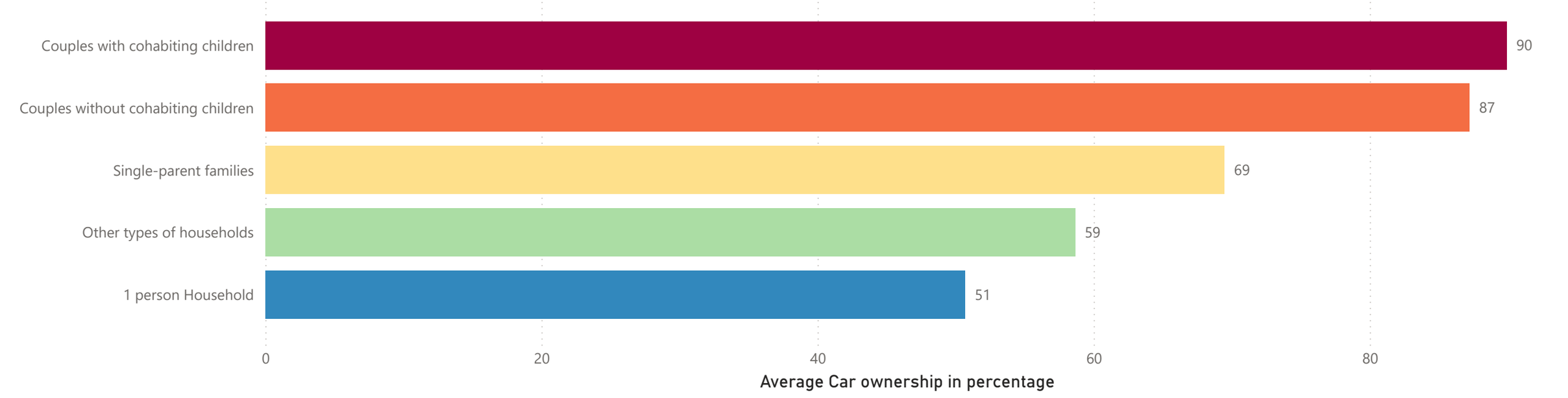
The data engineering tools employed in this project, known for their reliability and scalability, played a crucial role in managing, processing, and analyzing large and complex datasets. By automating data collection and transformation, these tools not only ensured the accuracy of the results but also enabled a more efficient and dynamic analysis. This approach allows for real-time insights and facilitates the identification of trends, ultimately providing data-driven evidence to support informed decision-making. In this case, the insights generated could help policymakers, urban planners, and researchers better understand how urban characteristics influence transportation needs, guiding strategies to reduce car dependency and promote sustainable mobility options.

Overall, the results provide a helpful overview of how different factors influence car ownership, which could inform future studies and policy decisions aimed at shaping more efficient and sustainable transportation systems.

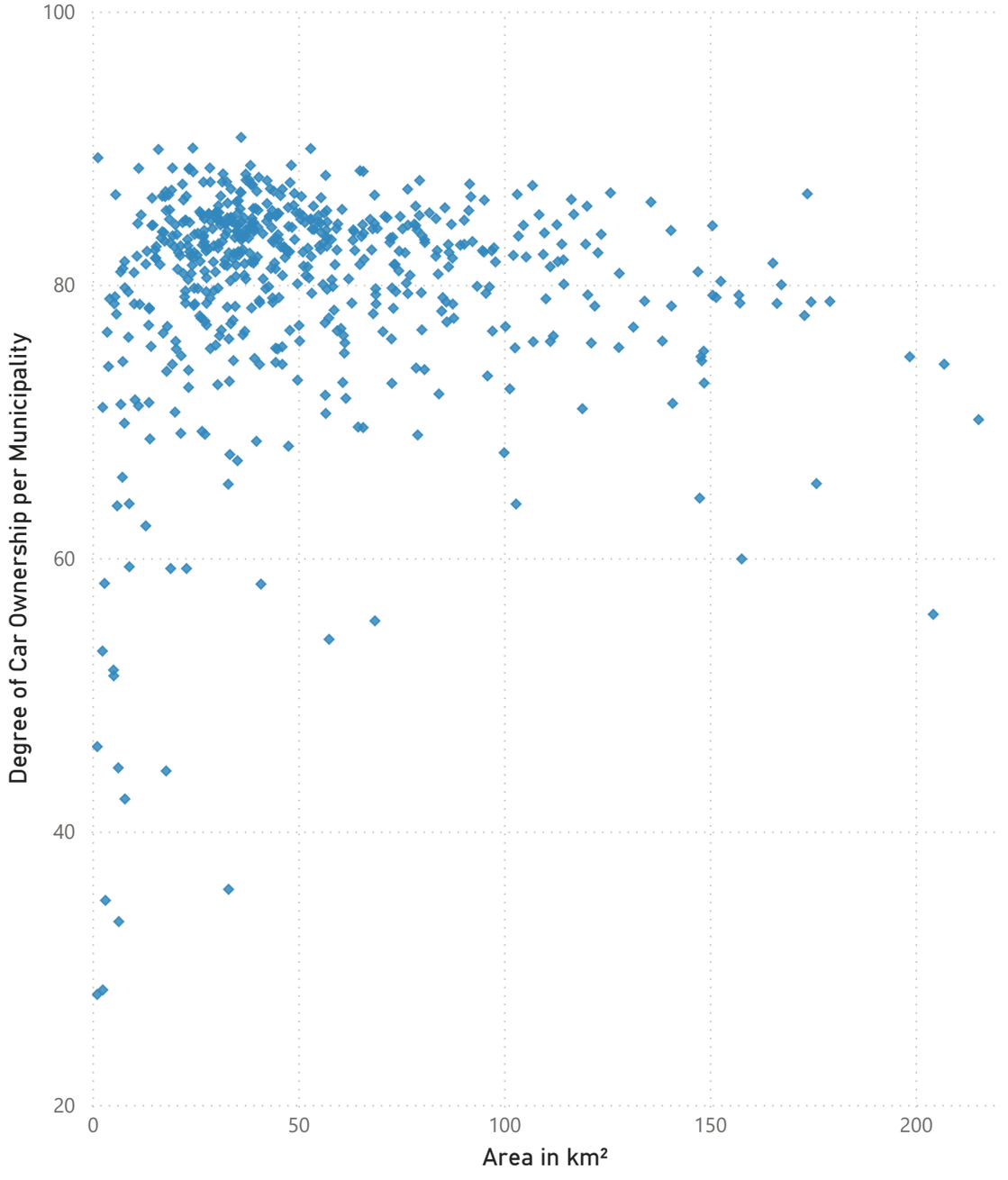
Degree of Car Ownership by Province



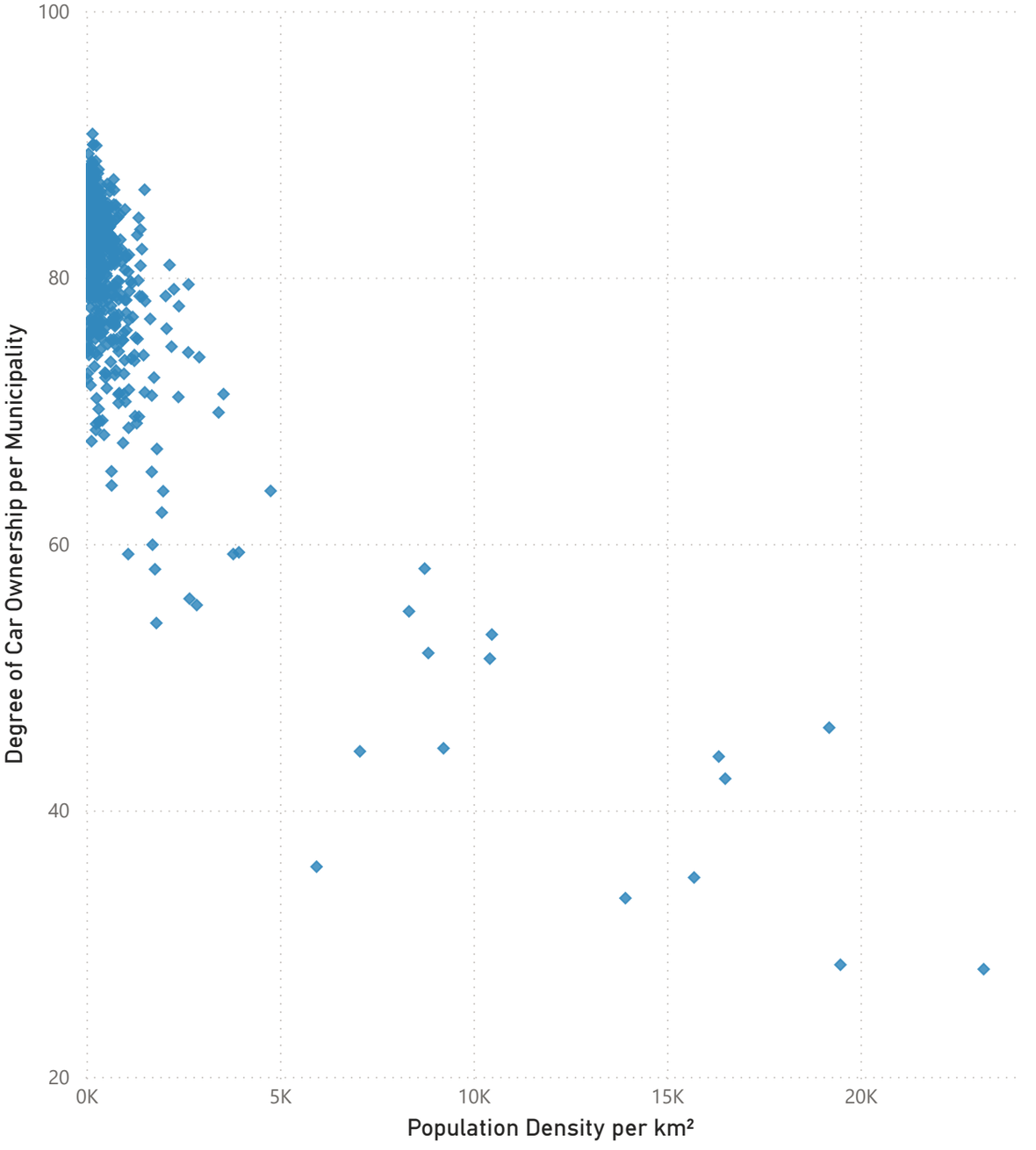
Degree of Car Ownership by Household Type

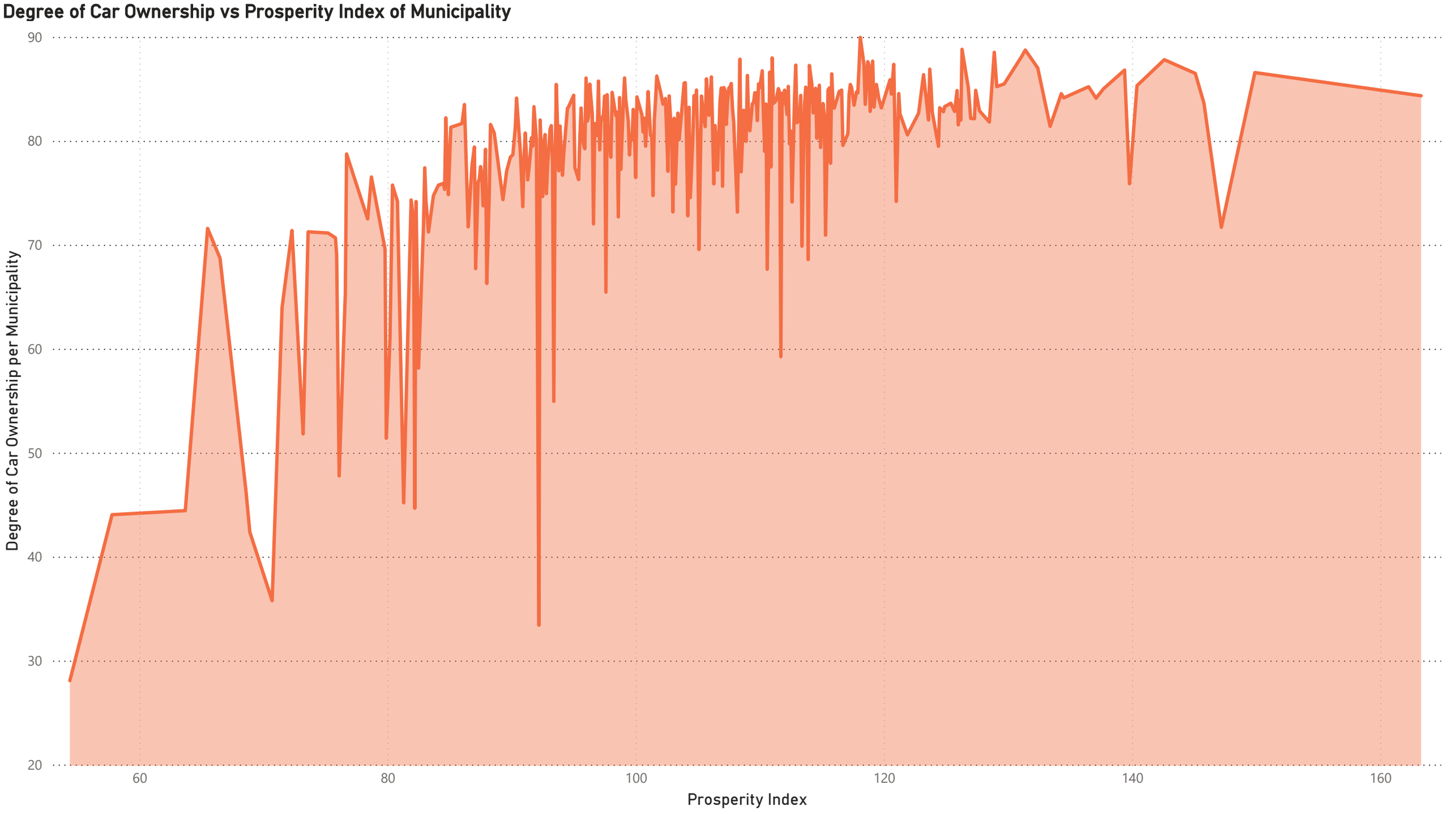


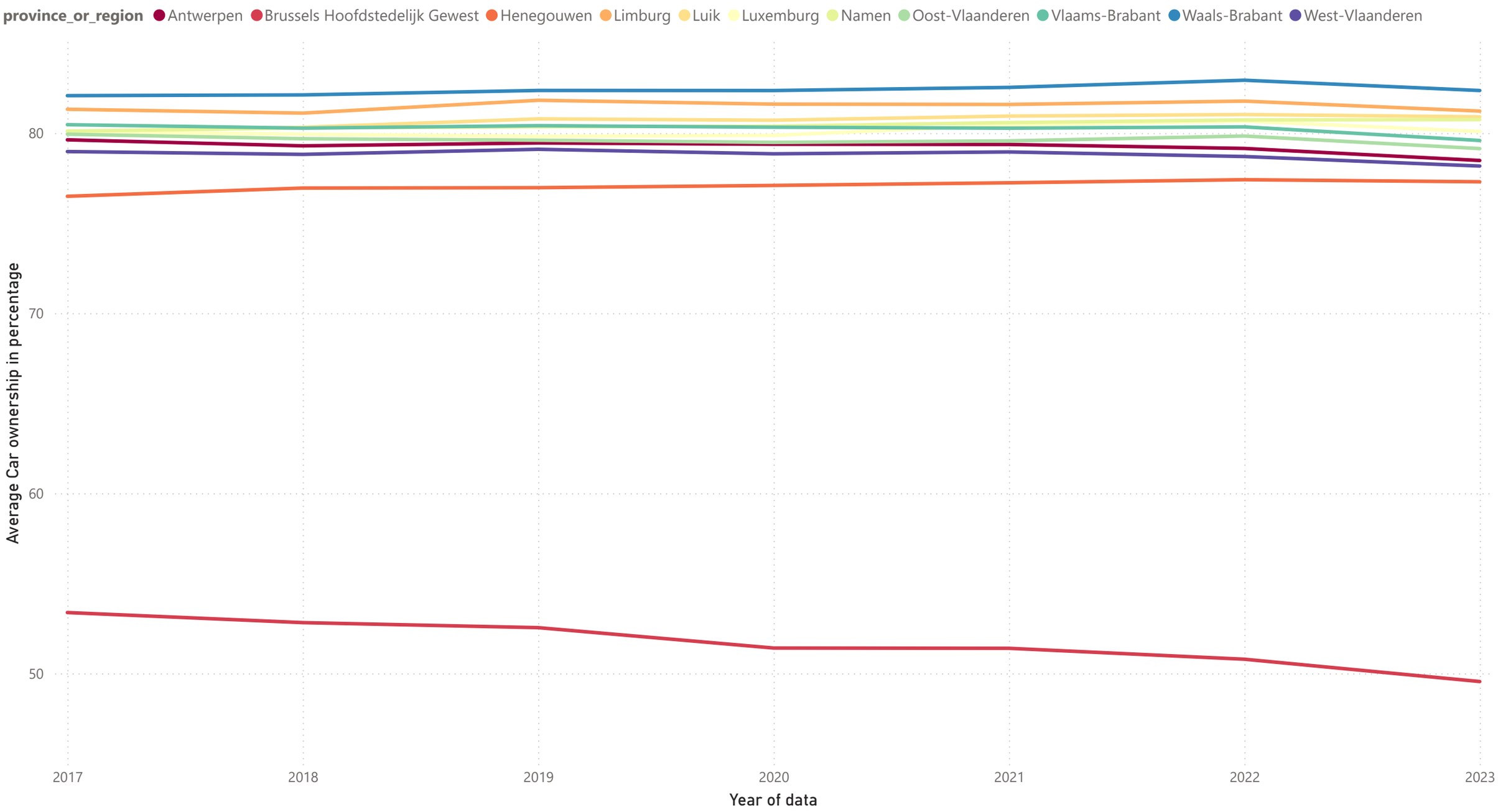
Degree of Car Ownership vs Municipality Size



Degree of Car ownership vs Population Density of Municipality




Degree of Car Ownership vs Prosperity Index of Municipality
Degree of Car Ownership per Municipality
90
80
70
60
50
40
30
20
60
80
100
120
140
160
Prosperity Index


Degree of Car Ownership by Province over the Years
province_or_region
Antwerpen
Brussels Hoofdstedelijk Gewest
Henegouwen
Limburg
Luik
Luxemburg
Namen
Oost-Vlaanderen
Vlaams-Brabant
Waals-Brabant
West-Vlaanderen
Average Car ownership in percentage
80
70
60
50
2017
2018
2019
2020
2021
2022
2023
Year of data