EV Charging Station Usage Prediction

Tiago Ferreira da Silva

Ironhack Data Science & Machine Learning Bootcamp

Nov 2024

Introduction

Challenge
As more people switch to Electric Vehicles, companies installing Charging Stations need to ensure they're located where demand is highest.

Project Goal
Develop a machine learning model to predict the usage of a new EV charging station based on its location.

Why It Matters
An overcrowded or underused station means unhappy customers or lost revenue.

Expected Outcome
A data-driven tool that helps companies plan better, strategically placing chargers where they'll be used most effectively.

DATA SOURCES

PAUA Dataset

Available Features & Target

Dataset Source

Paua Tech

Paua is a UK-based company that provides a unified EV charging solution for businesses, enabling fleet drivers to access multiple networks with a single app and card.

Dataset Source

Paua Tech

15 Locations x 7 Weekdays x 24 Hours = **2,520 datapoints**

LATITUDE	LONGITUDE	CITY	WEEKDAY	HOUR	NR_CONNECTORS	TOTAL_DURATION	CHARGING_CAPACITY	CHARGE_PERCENTAGE
50.918783	-0.454201	Storrington	Sunday	14	1	29.63	3420	0.0087
52.013641	4.375962	Delft	Friday	5	1	60.98	3360	0.0181
52.013641	4.375962	Delft	Saturday	14	1	0.0	3360	0.0
50.918783	-0.454201	Storrington	Friday	7	1	15.0	3360	0.0045
52.013641	4.375962	Delft	Friday	7	1	375.0	3360	0.1116
52.013641	4.375962	Delft	Sunday	17	1	0.0	3420	0.0
50.918783	-0.454201	Storrington	Friday	14	1	0.0	3360	0.0
52.013641	4.375962	Delft	Sunday	7	1	0.0	3420	0.0
52.013641	4.375962	Delft	Sunday	15	1	0.0	3420	0.0
50.918783	-0.454201	Storrington	Friday	17	1	0.0	3360	0.0
52.013641	4.375962	Delft	Saturday	10	1	0.0	3360	0.0
50.918783	-0.454201	Storrington	Monday	9	1	61.0	3420	0.0178
52.013641	4.375962	Delft	Monday	9	1	674.0	3420	0.1971
50.918783	-0.454201	Storrington	Saturday	18	1	12.0	3360	0.0036
52.013641	4.375962	Delft	Saturday	18	1	0.0	3360	0.0
50.918783	-0.454201	Storrington	Saturday	23	1	0.0	3360	0.0
50.918783	-0.454201	Storrington	Monday	14	1	136.98	3420	0.0401

*one datapoint indicates the yearly consumption of 1 Charger for specific Weekday and hour

Dataset Source

Paua Tech

LATITUDE	LONGITUDE	CITY	WEEKDAY	HOUR	NR_CONNECTORS	TOTAL_DURATION	CHARGING_CAPACITY	CHARGE_PERCENTAGE
50.918783	-0.454201	Storrington	Sunday	14	1	29.63	3420	0.0087
52.013641	4.375962	Delft	Friday	5	1	60.98	3360	0.0181
52.013641	4.375962	Delft	Saturday	14	1	0.0	3360	0.0
50.918783	-0.454201	Storrington	Friday	7	1	15.0	3360	0.0045
52.013641	4.375962	Delft	Friday	7	1	375.0	3360	0.1116
52.013641	4.375962	Delft	Sunday	17	1	0.0	3420	0.0
50.918783	-0.454201	Storrington	Friday	14	1	0.0	3360	0.0
52.013641	4.375962	Delft	Sunday	7	1	0.0	3420	0.0
52.013641	4.375962	Delft	Sunday	15	1	0.0	3420	0.0
50.918783	-0.454201	Storrington	Friday	17	1	0.0	3360	0.0
52.013641	4.375962	Delft	Saturday	10	1	0.0	3360	0.0
50.918783	-0.454201	Storrington	Monday	9	1	61.0	3420	0.0178
52.013641	4.375962	Delft	Monday	9	1	674.0	3420	0.1971
50.918783	-0.454201	Storrington	Saturday	18	1	12.0	3360	0.0036
52.013641	4.375962	Delft	Saturday	18	1	0.0	3360	0.0
50.918783	-0.454201	Storrington	Saturday	23	1	0.0	3360	0.0
50.918783	-0.454201	Storrington	Monday	14	1	136.98	3420	0.0401

Target variable

Dataset

Avaiable Features & Target

Features:

- ❑ Latitude
- ❑ Longitude
- ❑ Weekday
- ❑ Hour
- ❑ Charging_Capacity

Target Variable:

- ❑ Charge_Percentage

Dataset

Avaiable Features & Target

Features:

- ❑ Latitude
- ❑ Longitude
- ❑ Weekday
- ❑ Hour
- ❑ Charging_Capacity

!!

Problem:

Not enough features to build a robust model !!

Target Variable:

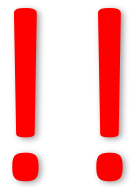
- ❑ Charge_Percentage

Dataset

Avaiable Features & Target

Features:

- ❑ Latitude
- ❑ Longitude
- ❑ Weekday
- ❑ Hour
- ❑ Charging_Capacity



Problem:

Not enough features to build a robust model !!

Solution:

Data Enrichment

Target Variable:

- ❑ Charge_Percentage

DATA ENRICHMENT

Demographics

Nearby Landmarks

Major Roads

Avg Traffic Volume

Dataset Enrichment

How do we add relevant info to our Data?

- **Demographics:**
  - Population, Population Density, Age distribution
  - Average Income, Cost of Living
  - Cars per 100 people
  - Percentage of EV
- **Nearby landmarks**:
  - How many Food & Beverage in the surrounding?
  - Retail & Convenience?
  - Leisure & Entertainment?
  - Health & Services?
- **Promity to major roads**:
  - Motorways, trunk, primary, secondary
- **Average Traffic Volume**
- **Nearby Competitors**

Dataset Enrichment

How do we add relevant info to our Data?

- **Demographics:**
  - Population, Population Density, Age distribution
  - Average Income, Cost of Living
  - Cars per 100 people
  - Percentage of EV

- **Nearby landmarks**:
  - How many Food & Beverage in the surrounding?
  - Retail & Convenience?
  - Leisure & Entertainment?
  - Health & Services?

- **Promity to major roads**:
  - Motorways, trunk, primary, secondary

- **Average Traffic Volume**

- **Nearby Competitors**

Dataset Enrichment

How do we add relevant info to our Data?

- **Demographics:**
  - Population, Population Density, Age distribution
  - Average Income, Cost of Living
  - Cars per 100 people
  - Percentage of EV

- **Nearby landmarks**:
  - How many Food & Beverage in the surrounding?
  - Retail & Convenience?
  - Leisure & Entertainment?
  - Health & Services?

- **Promity to major roads**:
  - Motorways, trunk, primary, secondary

- **Average Traffic Volume**

- **Nearby Competitors**

Dataset Enrichment

How do we add relevant info to our Data?

- ❑ **Demographics:**
  - ❑ Population, Population Density, Age distribution
  - ❑ Average Income, Cost of Living
  - ❑ Cars per 100 people
  - ❑ Percentage of EV
- ❑ **Nearby landmarks**:
  - ❑ How many Food & Beverage in the surrounding?
  - ❑ Retail & Convenience?
  - ❑ Leisure & Entertainment?
  - ❑ Health & Services?
- ❑ **Promity to major roads**:
  - ❑ Motorways, trunk, primary, secondary
- ❑ **Average Traffic Volume**
- ❑ **Nearby Competitors**

Dataset Enrichment

How do we add relevant info to our Data?

- **Demographics:**
  - Population, Population Density, Age distribution
  - Average Income, Cost of Living
  - Cars per 100 people
  - Percentage of EV
- **Nearby landmarks**:
  - How many Food & Beverage in the surrounding?
  - Retail & Convenience?
  - Leisure & Entertainment?
  - Health & Services?
- **Promity to major roads**:
  - Motorways, trunk, primary, secondary
- **Average Traffic Volume**
- **Nearby Competitors**

Dataset Enrichment

How do we add relevant info to our Data?

- **Demographics:**
  - Population, Population Density, Age distribution
  - Average Income, Cost of Living
  - Cars per 100 people
  - Percentage of EV
- **Nearby landmarks**:
  - How many Food & Beverage in the surrounding?
  - Retail & Convenience?
  - Leisure & Entertainment?
  - Health & Services?
- **Promity to major roads**:
  - Motorways, trunk, primary, secondary
- **Average Traffic Volume**
- **Nearby Competitors**

Dataset Enrichment

Nearby Landmarks

Approach:

- ❑ Use **OpenStreetMap (OSM)** API to retrieve geographic data
- ❑ Finds **nearby landmarks within** a **1 km radius** of each station location

Tags Searched:

- ❑ Amenities: cafe, restaurant, pub, library, parking, hospital, etc.
- ❑ Shops: supermarket, convenience, retail
- ❑ Leisure: park, playground, sports center
- ❑ Tourism: museum, zoo, attraction
- ❑ Competitors: charging_station

Output:

- ❑ DataFrame with **counts of nearby landmarks** categorized by type (amenity, shop, leisure, tourism)

```python
# Function to find nearby landmarks
def landmarks(lat_long):
    tags = {
        'amenity': ['cafe', 'restaurant', 'pub', 'library', 'parking', 'charging_s
        'shop': ['supermarket', 'convenience'],
        'leisure': ['park', 'playground'],
        'tourism': ['museum', 'attraction']
    }
    results = pd.DataFrame()

    for _, row in lat_long.iterrows():
        try:
            location = (row['LATITUDE'], row['LONGITUDE'])
            data = ox.features_from_point(location, tags=tags, dist=1000)
            counts = data.apply(pd.Series.value_counts).sum(axis=1).to_dict()
            results = pd.concat([results, pd.DataFrame([counts])], ignore_index=Tr
        except:
            continue

    return results

# Example usage
landmarks = landmarks(data[['LATITUDE', 'LONGITUDE']])
```

Dataset Enrichment

Proximity to major roads

Approach:

- ❑ Use **OpenStreetMap (OSM)** API to retrieve geographic data
- ❑ Finds **nearby major roads within** a **2 km radius** of each station location

Tags Searched:

- ❑ motorway
- ❑ Trunks
- ❑ Primary
- ❑ secondary

Output:

- ❑ DataFrame with **counts of nearby major roads**

```python
# Function to count major roads
def major_roads_count(lat_long):
    major_road_types = ['motorway', 'trunk', 'primary', 'secondary']
    results = pd.DataFrame()

    for _, row in lat_long.iterrows():
        try:
            location = (row['LATITUDE'], row['LONGITUDE'])
            G = ox.graph_from_point(location, dist=2000, network_type='drive')
            road_counts = {road: 0 for road in major_road_types}

            # Count each major road type
            for _, _, _, data in G.edges(data=True):
                if data['highway'] in major_road_types:
                    road_counts[data['highway']] += 1

            # Store counts in a DataFrame
            count_df = pd.DataFrame([{**{'LATITUDE': row['LATITUDE'], 'LONGITUDE':
            results = pd.concat([results, count_df], ignore_index=True)

        except:
            continue

    return results

# Example usage
lat_long_df = pd.DataFrame({
    'LATITUDE': [37.7749, 34.0522],  # Example latitudes
    'LONGITUDE': [-122.4194, -118.2437]  # Example longitudes
})

major_roads_results = major_roads_count(lat_long_df)
print(major_roads_results)
```

Dataset Enrichment

Average Traffic Volume

Approach:

- Use Google Maps **Distance Matrix API** to retrieve traffic-based travel duration.
- Finds the **average traffic duration to 8 nearby points within a 1 km radius around a specific location at a specified day and hour.**

Parameters

- Location: Center point (latitude, longitude)
- target_day, target_hour

Output:

- Single value representing the average duration in traffic around the specified location.

```
def get_avg_traffic_duration(location, day, hour, radius=1000, points=8):
    lat, lon = location
    step = 360 / points
    durations = []

    # Calculate target time
    today = datetime.datetime.now()
    days_ahead = (day - today.weekday() + 7) % 7
    target_time = (today + datetime.timedelta(days=days_ahead)).replace(hour=hour,
    departure_time = int(target_time.timestamp())

    for i in range(points):
        # Calculate nearby coordinates
        angle = i * step
        dest_lat = lat + (radius / 111320) * math.cos(math.radians(angle))
        dest_lon = lon + (radius / (111320 * math.cos(math.radians(lat)))) * math.s

        # API request
        url = f"https://maps.googleapis.com/maps/api/distancematrix/json?origins={
        response = requests.get(url).json()

        # Extract duration in traffic
        if response['status'] == 'OK' and response['rows'][0]['elements'][0]['statu
            duration = response['rows'][0]['elements'][0].get('duration_in_traffic
            if duration: durations.append(duration)

    return sum(durations) / len(durations) if durations else None
```

Dataset Enrichment

Population & Demographics

Current Challenges

- ❑ While demographic APIs are available, they can be challenging to use.
- ❑ Data quality and reliability vary significantly between countries.

Manual Input Requirement

- ❑ For our model, the user currently needs to research and manually input demographic data.

Future Direction

- ❑ Integrate APIs to automatically extract demographic data based on location.

Dataset Enrichment

Population & Demographics

COUNTRY	TOWN	AVG_ICOME	COST_OF_LIVING	NET_INCOME	CARS_PER_100	EV_RATE	POPULATION	POP_DENSITY	AGE_20	AGE_40	AGE_60	AGE_80	AGE_100
UK	Banbury	36000	1200	21600	57,5	2,5	52052	3766	23,28	30,16	26,52	15,92	4,12
Belgium	Bilzen	35000	1000	23000	52,5	1,75	32782	431,5	20,15	22,51	27,01	24,18	6,15
Belgium	Diepenbeek	35000	1000	23000	52,5	1,75	19607	473,5	19,23	23,38	27,42	23,92	6,04
Germany	Rodgau	47500	1000	35500	57,5	3,75	44501	684,2	18,68	23,13	28,98	22,56	6,66
Netherlands	Den Hoorn	42000	1100	28800	47,5	5,5	8250	2806	18,24	39,73	20,36	18,03	3,74
UK	Barnard Castle	36000	1200	21600	62,5	1,75	5784	2667	18,96	17,35	26,03	28,05	9,61
Netherlands	Staphorst	40000	900	29200	57,5	3,5	17739	132,4	27,92	24,22	22,44	17,36	8,06
UK	Chester	41000	1500	23000	52,5	3,5	92742	3760	20,97	28,94	25,78	18,89	5,42
Netherlands	Oud Gastel	40000	1100	26800	57,5	4,5	5495	2907	19,59	21,46	26,77	25,99	6,2
Netherlands	Oosterhout	40000	1100	26800	52,5	4,5	57924	810,9	20,39	22,54	26,17	25,36	5,53
UK	Storrington	36000	1200	21600	62,5	3,5	8709	1936	17,71	15,71	25,25	31,65	9,68
UK	Whitley Bay	36000	1200	21600	52,5	2,75	36866	5045	22,29	17,95	29,66	23,5	6,61
Germany	Bersteland	35000	650	27200	62,5	2,25	844	28,59	18,39	14,59	32,62	25,62	8,78
Netherlands	Nuis	40000	900	29200	60	3,5	375	2500	22,64	21,46	26,61	23,92	5,36

```
# Read the Dataset with demographics
demographic = pd.read_csv("./dataset/demographic.csv", delimiter=';')

# Merge demographic features with our main dataset on 'TOWN'
data = pd.merge(data, demographic, on='TOWN', how='left')
```

DATA PREPROCESS

Grouping Amenities

Weekday names to int

Data Preprocess

Amenities Grouping

Simplify amenity data by grouping specific locations into broader, meaningful categories.

Broader Categories:

- ❑ **Food & Beverage**: fast food, cafes, restaurants, bars, pubs, bakeries
- ❑ **Retail & Convenience**: supermarkets, convenience stores
- ❑ **Competitors**: other charging stations
- ❑ **Entertainment**: parks, sports centers, museums, cinemas, zoos
- ❑ **Health & Services**: hospitals, hotels, libraries, parking facilities, public toilets

Implementation:

- ❑ Summing individual columns within each group to create a single category feature.
- ❑ Dropping the original amenity columns to simplify the dataset.

```
# Group amenities into broader categories
data['FOOD_&_BEVERAGE'] = data[['fast_food', 'cafe', 'restaurant', 'bar', 'pub', 'b
data['RETAIL_&_CONVENIENCE'] = data[['supermarket', 'convenience']].sum(axis=1)
data['COMPETITORS'] = data['charging_station']
data['ENTERTAINMENT'] = data[['playground', 'park', 'community_centre', 'sports_cen
data['HEALTH_&_SERVICES'] = data[['hospital', 'hotel', 'library', 'parking', 'toile

# Drop original columns
data = data.drop(['toilets', 'fast_food', 'cafe', 'restaurant', 'bar', 'pub', 'bake
                  'charging_station', 'playground', 'park', 'community_centre', 'sp
                  'theatre', 'cinema', 'zoo', 'hospital', 'hotel', 'library', 'park
```

Data Preprocess

Weekday names to Int

Convert weekday names (e.g., "Monday") to integers for easier data handling and analysis.

Mapping:

- ❑ **0** = Monday, **1** = Tuesday, ..., **6** = Sunday

Implementation:

- ❑ A function weekday_to_int checks if the input matches a weekday name.
- ❑ Returns the corresponding integer if valid, otherwise returns None.

```python
# Function to convert a weekday name (e.g., "Monday", "Tuesday") into the correspo
def weekday_to_int(weekday_name):
    """
    Convert a weekday name to an integer.

    Args:
    - weekday_name (str): The name of the weekday (e.g., "Monday", "Tuesday").

    Returns:
    - int: Integer corresponding to the weekday (0=Monday, 1=Tuesday, ..., 6=Sunda
           Returns None if the input is not a valid weekday.
    """
    days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "S

    try:
        return days.index(weekday_name.capitalize())
    except ValueError:
        return None
```

FEATURE ENGINEERING

Hourly Categorization

Weekday Grouping

Encoding

Feature Engineering

Hourly Categorization and Encoding

Group hours into time slots

Time Slots:

- **Morning**: 6 AM - 10 AM
- **Midday**: 10 AM - 4 PM
- **Evening**: 4 PM - 8 PM
- **Night**: 8 PM - 6 AM

Implementation:

- A function time_slot categorizes each hour into a slot.
- One-hot encoding is applied to create separate columns for each time slot.

```
# Function to categorize hour into time slots
def time_slot(hour):
    if 6 <= hour < 10:
        return "MORNING"
    elif 10 <= hour < 16:
        return "MIDDAY"
    elif 16 <= hour < 20:
        return "EVENING"
    else:
        return "NIGHT"

# Apply time slots to 'HOUR' and one-hot encode
data['TIME_SLOT'] = data['HOUR'].apply(time_slot)
data = pd.get_dummies(data, columns=['TIME_SLOT']).drop(['HOUR'], axis=1)
```

Feature Engineering

Weekday Grouping and Encoding

Group days into **Workday** (Mon-Fri) and **Weekend** (Sat-Sun)

Implementation:

- ❑ A function day_type categorizes each day as either WORKDAY or WEEKEND.
- ❑ The function is applied to create a new column DAY_TYPE.
- ❑ One-hot encoding is used to split DAY_TYPE into separate columns, and the original weekday column is removed.

```
# Function to group weekdays
def day_type(weekday):
    return "WORKDAY" if weekday <= 4 else "WEEKEND"

# Apply day type categorization and one-hot encode
data['DAY_TYPE'] = data['WEEKDAY'].apply(day_type)
data = pd.get_dummies(data, columns=['DAY_TYPE']).drop(['WEEKDAY'], axis=1)
```

Correlation

Correlation Matrix

Correlation

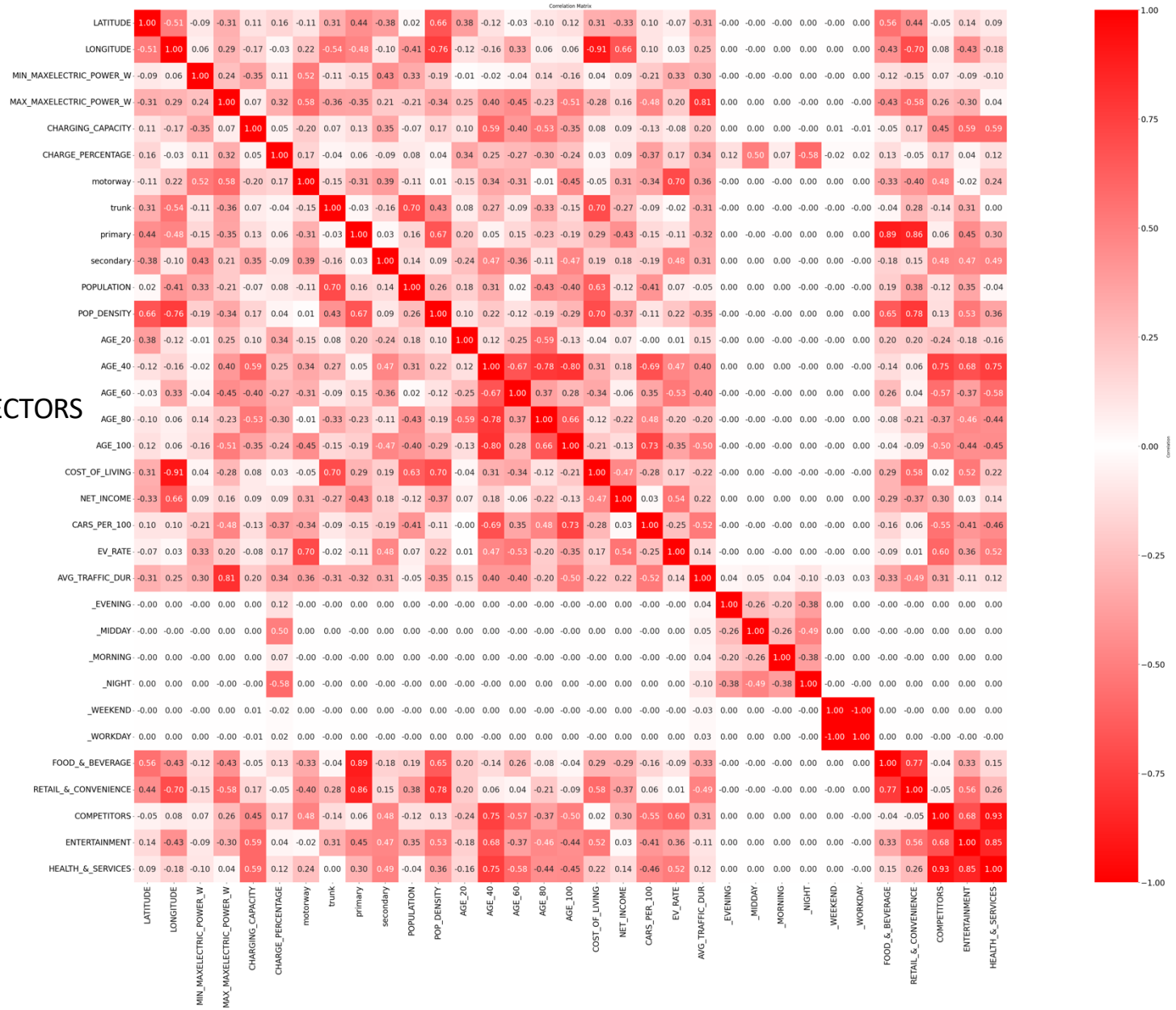
Correlation Matrix

High correlation between:

- ❑ AVG_INCOME X NET_INCOME
- ❑ CHARGING_CAPACITY X NR_OF_CONNECTORS

DROP:

- ❑ AVG_INCOME
- ❑ NR_OF_CONNECTORS



Final Dataset

Enriched and preprocessed

Final Dataset

Features

Category	Columns
Location Data	`LATITUDE`, `LONGITUDE`
Charger Specifications	`MIN_MAXELECTRIC_POWER_W`, `MAX_MAXELECTRIC_POWER_W`, `CHARGING_CAPACITY`
Road Proximity	`motorway`, `trunk`, `primary`, `secondary`
Population and Demographics	`POPULATION`, `POP_DENSITY`, `AGE_20`, `AGE_40`, `AGE_60`, `AGE_80`, `AGE_100`
Economic Factors	`COST_OF_LIVING`, `NET_INCOME`
Vehicle Statistics	`CARS_PER_100`, `EV_RATE`
Traffic Data	`AVG_TRAFFIC_DUR`
Time-Based Variables	`_EVENING`, `_MIDDAY`, `_MORNING`, `_NIGHT`, `_WEEKEND`, `_WORKDAY`
Nearby Amenities	`FOOD_&_BEVERAGE`, `RETAIL_&_CONVENIENCE`, `COMPETITORS`, `ENTERTAINMENT`, `HEALTH_&_SERVICES`

Target

Category	Columns
Utilization Rate	`CHARGE_PERCENTAGE`

Model

Train-test split

Features and Labels split

Feature Scaling

Model Selection

Grid Search

Test and Evaluate

Model

Train-Test split

to ensure the target variable's distribution is consistent between the training and test sets.

use stratify= ['POP_DENSITY_bin’] to ensure each split has a similar distribution of population density, as defined by the bins.

```
# Parameters
n_bins = 10  # Number of bins for stratification
test_size = 0.2  # Proportion of the dataset to include in the test split
random_state = 42  # Random seed for reproducibility

# Step 1: Create bins for population density
data['POP_DENSITY_bin'] = pd.qcut(data['POP_DENSITY'], q=n_bins, labels=False)

# Step 2: Perform stratified train-test split based on population density bins
train, test = train_test_split(
    data,
    test_size=test_size,
    stratify=data['POP_DENSITY_bin'],
    random_state=random_state
)

# Step 3: Drop the bin column after the split
train = train.drop(columns=['POP_DENSITY_bin'])
test = test.drop(columns=['POP_DENSITY_bin'])
```

Model

Features and Label split

```
# Features
X_train = train.drop(columns=['CHARGE_PERCENTAGE'])
X_test = test.drop(columns=['CHARGE_PERCENTAGE'])

# Label
y_train = train['CHARGE_PERCENTAGE']
y_test = test['CHARGE_PERCENTAGE']
```

Model

Feature Scaling

Fitting and Transforming the Trainin set

Transforming the Test Set

```
# Initialize the scaler
scaler = StandardScaler()

# Standardize train features (fit_transform)
X_train_scaled = scaler.fit_transform(X_train)

# Standardize test features using stats of train features (transform)
X_test_scaled = scaler.transform(X_test)

# Convert back into pandas DataFrame
X_train_scaled = pd.DataFrame(X_train_scaled, columns=X_train.columns)
X_test_scaled = pd.DataFrame(X_test_scaled, columns=X_test.columns)
```

Model

Model Selection

Preliminar Evaluation of multiple Regressors

```python
# Dictionary for models
models = {
    "Linear Regression": LinearRegression(),
    "Decision Tree Regressor": DecisionTreeRegressor(max_depth=10),
    "Random Forest Regressor": RandomForestRegressor(n_estimators=100),
    "Gradient Boosting Regressor": GradientBoostingRegressor(n_estimators=100),
    "Support Vector Regressor": SVR(C=1.0),
    "k-Nearest Neighbors Regressor": KNeighborsRegressor(n_neighbors=5)
}
```

Cross Validation

```python
# Cross-validation
for name, model in models.items():
    score = cross_val_score(estimator=model, X=X_train_scaled, y=y_train, cv=5, sc
    print(f'{name} mean score : {np.mean(score)}')
    print('-' * 40)
```

Top 3 Models

```
Linear Regression mean score : 0.7383190516185417
----------------------------------------
Decision Tree Regressor mean score : 0.837912183376478
----------------------------------------
Random Forest Regreesor mean score : 0.8701449315442973
----------------------------------------
Gradient Boosting Regressor mean score : 0.8705764346540885
----------------------------------------
Support Vector Regressor mean score : 0.8635486796875711
----------------------------------------
k-Nearest Neighbors Regressor mean score : 0.8794878901397076
----------------------------------------
```

Preliminar R2 score of ~ 88 %

Model

Grid Search : Hyperparameters Grid

Random Forest Grid

Gradient Boosting Grid

K-Nearest Neighbors

```
## Random Forest
rf_param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [10, 20, 30, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['auto', 'sqrt', 'log2'],
    'bootstrap': [True, False]
}

## Gradient Boosting
gb_param_grid = {
    'n_estimators': [100, 200, 300],
    'learning_rate': [0.01, 0.05, 0.1, 0.2],
    'max_depth': [3, 5, 7, 9],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'subsample': [0.6, 0.8, 1.0],
    'max_features': ['auto', 'sqrt', 'log2']
}

# k-Nearest Neighbors
knn_param_grid = {
    'n_neighbors': [3, 5, 7, 9, 11],
    'weights': ['uniform', 'distance'],
    'metric': ['euclidean', 'manhattan', 'minkowski']
}
```

Model

Grid Search

```
# Grid Search
from sklearn.model_selection import GridSearchCV

## Random Forest
rf_grid_search = GridSearchCV(estimator=RandomForestRegressor(), param_grid=rf_para
rf_grid_search.fit(X_train_scaled, y_train)

# Best parameters and score for Random Forest
rf_best_params = rf_grid_search.best_params_
rf_best_score = rf_grid_search.best_score_

## Gradient Boosting
gb_grid_search = GridSearchCV(estimator=GradientBoostingRegressor(), param_grid=gb_
gb_grid_search.fit(X_train_scaled, y_train)

# Best parameters and score for Gradient Boosting
gb_best_params = gb_grid_search.best_params_
gb_best_score = gb_grid_search.best_score_

## k-Nearest Neighbors
knn_grid_search = GridSearchCV(estimator=KNeighborsRegressor(), param_grid=knn_para
knn_grid_search.fit(X_train_scaled, y_train)

# Best parameters and score for k-Nearest Neighbors
knn_best_params = knn_grid_search.best_params_
knn_best_score = knn_grid_search.best_score_
```

```
Random Forest Regressor best Score: 0.8896460033268715
Random Forest Regressor best hyperparameters: {'bootstrap':
----------------------------------------
Gradient Boosting Regressor best Score: 0.8918444266538603
Gradient Boosting Regressor best hyperparameters: {'learnin
----------------------------------------
k-NN Regressor best Score: 0.8806962512918262
k-NN Regressor best hyperparameters: {'metric': 'manhattan'
```

Model

Test and Evaluate

```
# Test on test set
y_pred = trained_model.predict(X_test_scaled)

# Evaluate the model: performance on test set
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

# Print the results
print('Performance metrics of the Trained Model:')
print(f'Mean Squared Error: {mse}')
print(f'Mean Absolute Error: {mae}')
print(f'R^2 Score: {r2}')
```

```
Performance metrics of Trained model:
Mean Squared Error: 0.007143573669675724
Mean Absolute Error: 0.05941555105915787
r2 score: 0.9080634538045181
```

Deployment

Streamlit app

Future Work

Future Work

1. Integrate Real-Time Data Sources

❑ Leverage APIs to gather live population and demographic data for enhanced prediction accuracy.

2. Deploy the Model in a Production Environment

3. Expand to Regional and City-Wide Predictions

❑ Create a dynamic color map that displays usage predictions across a specific region, city, or town.

4. Enhance Prediction Accuracy

❑ Experiment with advanced models and algorithms, such as ensemble learning or neural networks, for improved forecasting.