Documentation for recreating ”Two-dimensional Pareto frontier forecasting for technology planning and roadmapping”

Arshia Feizmohammady

August 8, 2025

Introduction

This document provides thorough documentation for recreating the results of the article written by Ilya Yuskevich,Rob Vingerhoeds, and Alessandro Golkar. It includes an overview of the methodology, code snippets, and explanations to ensure reproducibility and understanding.

Data Gathering

This section provides an overview of the data gathering and cleaning process. It includes detailed steps on how data was collected, and what sources were used.

The authors of the article utilized the following website to gather their data:

Car Specs Database — cars-data.com [WWW Document], n.d., URL: `https://www.cars-data.com/` (accessed 2.9.20), 2020,

This website hosts a comprehensive dataset of all car models. However, it does not provide an option to directly download the data, necessitating web scraping to obtain the required information.

Due to technical challenges associated with web scraping, an alternative approach was taken. A publicly available dataset from Kaggle was used. This dataset can be found at:

Car Specification Dataset 1945-2020, URL: `https://www.kaggle.com/datasets/jahaidulislam/car-specification-dataset-1945-2020`.

It is important to acknowledge the differences between these two datasets, as they can impact the recreated figures. The dataset used by the original authors consists of 3,289 datapoints and encompasses car models from 32 car brands representing all major automotive companies. In contrast, the dataset obtained from Kaggle contains over 50,000 datapoints, providing a broader range of information.

Data Visualization

This section visualizes the total max power vs. fuel consumption tradeoff of cars between 1975 and 2015. Below is the Python code used for data visualization:

```python
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.colors import LinearSegmentedColormap

# The dataset has the input Fuel Consumption, but we need to convert that to the output Miles Per Gallon (MPG)
KM_PER_100KM_TO_MPG = 235.214583

# Load and read the data
file_path = 'Car Dataset 1945-2020.csv'
car_data = pd.read_csv(file_path, low_memory=False)
```

Listing 1: Data Visualization Code

First, we import the `pandas` and `matplotlib.pyplot` libraries for data manipulation and plotting, respectively. Additionally, we import `LinearSegmentedColormap` for creating custom colormaps. We then define a conversion factor from kilometers per 100 kilometers to miles per gallon (MPG) and load the dataset from a CSV file.

```python
# Exclude rows with NaN or non-numeric data points in critical columns - not excluding these will create problems with
# the code, possibly not allowing it to run
car_data = car_data.dropna(subset=['engine_hp', 'mixed_fuel_consumption_per_100_km_l'])
car_data = car_data[pd.to_numeric(car_data['engine_hp'], errors='coerce').notnull()]
car_data = car_data[pd.to_numeric(car_data['mixed_fuel_consumption_per_100_km_l'], errors='coerce').notnull()]
```

Next, we exclude rows with NaN or non-numeric data points in critical columns such as `engine_hp` and `mixed_fuel_consumption_per_100_km_l` to prevent errors during analysis.

```python
# Remove duplicate data points - In the article they stated that we should remove duplicate data points
car_data = car_data.drop_duplicates(subset=['engine_hp', 'mixed_fuel_consumption_per_100_km_l', 'fuel_grade', 'transmission', 'Year_from'])
```

We then remove duplicate data points based on several columns (`engine_hp`, `mixed_fuel_consumption_per_100_km_l`, `fuel_grade`, `transmission`, `Year_from`).

```python
# Define a custom colormap - this custom colormap is to visualize the year for each dataset - I followed the gradient they used in the article
cmap = LinearSegmentedColormap.from_list('custom_cmap', ['darkblue', 'lightblue', 'blue','lightgreen', 'green', 'orange', 'yellow'])
```

A custom colormap is defined to visualize the data according to the year, following the gradient used in the article.

```python
# Filter data to include only petrol and manual transmission cars - In the article, they filtered for manual transmission cars only
# Some petrol car models in the data set aren't defined as petrol, but some other keywords, we therefore need to filter for those keywords
petrol_keywords = ['95', '98', '92', '80', 'Gasoline', 'Gas', 'Ethanol']
filtered_data = car_data[
    (car_data['fuel_grade'].str.strip().str.lower().isin([kw.lower() for kw in petrol_keywords])) &
    (car_data['transmission'].str.strip().str.lower() == 'manual'.lower()) &
    (car_data['Year_from'] >= 1970) &
    (car_data['Year_from'] <= 2015)
].copy()
```

The data is filtered to include only petrol cars with manual transmissions, produced between 1970 and 2015. Specific keywords are used to identify petrol cars.

```python
# Print first few rows of filtered data to check filtering - this is an optional step, mainly for debugging, I have commented it our for  now

##print(filtered_data.head())
```

An optional step to print the first few rows of filtered data for debugging purposes is included but commented out.

```python
# Function to preprocess data and convert fuel consumption - converting input to output
def preprocess_data(data):
    data['mixed_fuel_consumption_mpg'] = KM_PER_100KM_TO_MPG / data['mixed_fuel_consumption_per_100_km_l']
    return data

# Preprocess data for filtered data
filtered_data = preprocess_data(filtered_data)
```

A function `preprocess_data` is defined to convert fuel consumption to MPG, and the filtered data is preprocessed accordingly.

```python
# Print the number of data points - the article had around 5000 datapoints, the dataset I'm using has around 9000
# This could result in some discrepencies later on
print(f"Number of data points: {len(filtered_data)}")
```

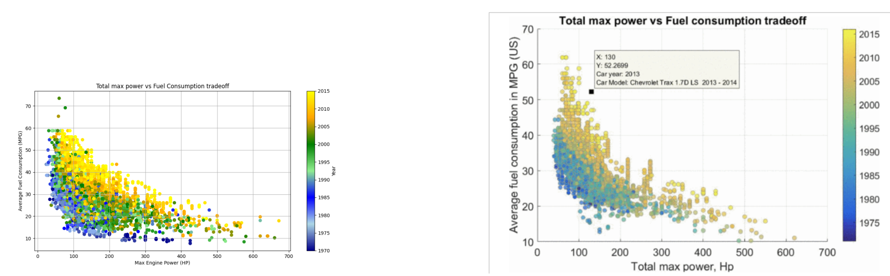
The number of data points in the filtered dataset is printed for comparison with the original article.

```python
# Function to create scatter plots with actual release years
def plot_car_data(data, title):
    plt.figure(figsize=(12, 6))
    scatter = plt.scatter(data['engine_hp'], data['
        mixed_fuel_consumption_mpg'], c=data['Year_from'], cmap=
        cmap)
    cbar = plt.colorbar(scatter, label='Year')
    plt.title(title)
    plt.xlabel('Max Engine Power (HP)')
    plt.ylabel('Average Fuel Consumption (MPG)')
    plt.grid(True)
    plt.show()

# Plot the car data for all models
plot_car_data(filtered_data, 'Total max power vs Fuel Consumption
    tradeoff')
```

Finally, a function `plot_car_data` is defined to create scatter plots of engine power vs. fuel consumption, with the color representing the year. The plot is displayed using `matplotlib`.

Comparison of Results - Total Max Power vs. Fuel Consumption Tradeoff

In this subsection, we will compare the generated plot with the corresponding plot from the original article. The comparison will help in understanding the discrepancies and similarities between the recreated results and the article's results.



(a) Obtained Plot (b) Article Plot

Figure 1: Comparison of Total Max Power vs. Fuel Consumption Tradeoff

By comparing the two images, we can see that while they are not identical, they are quite similar, suggesting that we chose a good dataset to use for recreating the results of this article.

Pareto Frontier

In this section, we will try to find the best responses of the data and estimate a Pareto frontier based on that. Below is the Python code used to achieve this:

```python
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.colors import LinearSegmentedColormap
import numpy as np
from scipy.optimize import curve_fit

# The dataset has the input Fuel Consumption, but we need to convert that to the output Miles Per Gallon (MPG)
KM_PER_100KM_TO_MPG = 235.214583

# Load and read the data
file_path = 'Car Dataset 1945-2020.csv'
car_data = pd.read_csv(file_path, low_memory=False)
```

Listing 2: Pareto Frontier Code

First, we import the necessary libraries: `pandas` for data manipulation, `matplotlib.pyplot` for plotting, `LinearSegmentedColormap` for creating custom colormaps, `numpy` for numerical operations, and `curve_fit` from `scipy.optimize` for fitting a logarithmic curve. We define a conversion factor from kilometers per 100 kilometers to miles per gallon (MPG) and load the dataset from a CSV file.

```python
# Exclude rows with NaN or non-numeric data points in critical columns
car_data = car_data.dropna(subset=['engine_hp', 'mixed_fuel_consumption_per_100_km_l'])
car_data = car_data[pd.to_numeric(car_data['engine_hp'], errors='coerce').notnull()]
car_data = car_data[pd.to_numeric(car_data['mixed_fuel_consumption_per_100_km_l'], errors='coerce').notnull()]

# Remove duplicate data points
car_data = car_data.drop_duplicates(subset=['engine_hp', 'mixed_fuel_consumption_per_100_km_l', 'fuel_grade', 'transmission', 'Year_from'])
```

Next, we exclude rows with NaN or non-numeric data points in critical columns such as `engine_hp` and `mixed_fuel_consumption_per_100_km_l` to prevent errors during analysis. We also remove duplicate data points based on several columns (`engine_hp`, `mixed_fuel_consumption_per_100_km_l`, `fuel_grade`, `transmission`, `Year_from`).

```python
# Define a custom colormap
cmap = LinearSegmentedColormap.from_list('custom_cmap', ['darkblue', 'lightblue', 'blue', 'lightgreen', 'green', 'orange', 'yellow'])

# Filter data to include only petrol and manual transmission cars from 1970
petrol_keywords = ['95', '98', '92', '80', 'Gasoline', 'Gas', 'Ethanol']
```

```python
filtered_data = car_data[
    (car_data['fuel_grade'].str.strip().str.lower().isin([kw.lower() for kw in petrol_keywords])) &
    (car_data['transmission'].str.strip().str.lower() == 'manual'.lower()) &
    (car_data['Year_from'] == 1970)
].copy()
```

We define a custom colormap to visualize the data according to the year and filter the data to include only petrol cars with manual transmissions from the year 1970. Specific keywords are used to identify petrol cars.

```python
# Function to preprocess data and convert fuel consumption
def preprocess_data(data):
    data['mixed_fuel_consumption_mpg'] = KM_PER_100KM_TO_MPG / data['mixed_fuel_consumption_per_100_km_l']
    return data

# Preprocess data for filtered data
filtered_data = preprocess_data(filtered_data)
```

A function `preprocess_data` is defined to convert fuel consumption to MPG, and the filtered data is preprocessed accordingly.

```python
# Print the number of data points for 1970
print(f"Number of data points in 1970: {len(filtered_data)}")
```

The number of data points in the filtered dataset for the year 1970 is printed.

```python
# Function to identify maximum points along the x-axis
def identify_max_points_along_x(df, step=1, max_x=800):
    x_values = np.arange(0, max_x+1, step)
    max_points = []
    for x in x_values:
        subset = df[df['engine_hp'] == x]
        if not subset.empty:
            max_y = subset['mixed_fuel_consumption_mpg'].max()
            max_points.append((x, max_y))
    return pd.DataFrame(max_points, columns=['engine_hp', 'mixed_fuel_consumption_mpg'])

# Identify maximum points along the x-axis for 1970
max_points_1970 = identify_max_points_along_x(filtered_data)
```

We define a function `identify_max_points_along_x` to identify the maximum points along the x-axis (engine horsepower). This function helps in determining the Pareto frontier. The maximum points for the year 1970 are identified using this function.

```python
# Function to fit a logarithmic curve
def logarithmic_fit(x, a, b, c):
    return a * np.log(b * (x + 1)) + c

def fit_logarithmic_curve(max_points):
    x = max_points['engine_hp'].values
    y = max_points['mixed_fuel_consumption_mpg'].values

    # Fit the logarithmic curve
```

```
    popt, _ = curve_fit(logarithmic_fit, x, y, maxfev=10000)

    return popt

# Fit the logarithmic curve for the maximum points
log_params = fit_logarithmic_curve(max_points_1970)
```

A logarithmic function `logarithmic_fit` is defined, and another function `fit_logarithmic_curve` is used to fit this curve to the maximum points identified. The parameters of the fitted logarithmic curve are obtained.

```
# Determine the adjusted limits for the plot
max_y_pareto = max_points_1970['mixed_fuel_consumption_mpg'].max()
max_x_pareto = max_points_1970['engine_hp'].max()
y_limit = np.ceil(max_y_pareto / 10) * 10
x_limit = np.ceil(max_x_pareto / 50) * 50
```

The adjusted limits for the plot are determined based on the maximum points along the y-axis and x-axis.

```
# Function to create scatter plots with actual release years and the Pareto frontier
def plot_car_data_with_pareto_frontier(data, max_points, log_params, title):
    plt.figure(figsize=(12, 6))
    scatter = plt.scatter(data['engine_hp'], data['mixed_fuel_consumption_mpg'], c=data['Year_from'], cmap=cmap)
    cbar = plt.colorbar(scatter, label='Year')
    plt.title(title)
    plt.xlabel('Max Engine Power (HP)')
    plt.ylabel('Average Fuel Consumption (MPG)')
    plt.grid(True)

    # Plot the Pareto frontier
    plt.plot(max_points['engine_hp'], max_points['mixed_fuel_consumption_mpg'], 'ro-', label='Pareto Frontier')

    # Plot the fitted logarithmic curve within the specified limits
    x_vals = np.linspace(0, x_limit, 1000)
    y_vals = logarithmic_fit(x_vals, *log_params)
    y_vals = np.minimum(y_vals, y_limit)
    plt.plot(x_vals, y_vals, 'g--', label='Logarithmic Fit')

    plt.legend()
    plt.xlim(0, x_limit)
    plt.ylim(0, y_limit)
    plt.show()

# Plot the car data for 1970 with Pareto frontier and logarithmic fit
plot_car_data_with_pareto_frontier(filtered_data, max_points_1970, log_params, '1970: Max Engine Power vs Fuel Consumption tradeoff with Pareto Frontier')
```

Finally, a function `plot_car_data_with_pareto_frontier` is defined to create scatter plots of engine power vs. fuel consumption with the actual release years and the Pareto frontier. The fitted logarithmic curve is also plotted within the specified limits. The car data for 1970 is plotted with the Pareto frontier and the logarithmic fit.

Result

In this subsection, we present the results of the Pareto frontier analysis. The following image shows the Pareto frontier for the year 1970 along with the fitted logarithmic curve.

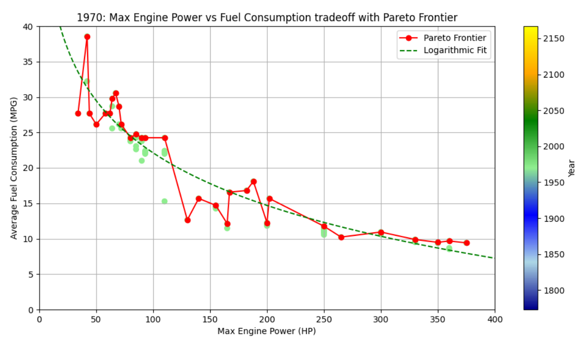


Figure 2: 1970: Max Engine Power vs Fuel Consumption Tradeoff with Pareto Frontier

We can see from the result that this is a good start. However, the logarithmic fit will need to be adjusted in our future code so that no data points are above it.

Pareto Frontiers Backtesting

In this section, we will utilize our Pareto frontier estimation method to estimate the Pareto frontiers until 2050, using both a complete (1970-2015) and incomplete (1970-1995) dataset. We will then compare those two results to backtest our method.

```
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.colors import LinearSegmentedColormap, Normalize
import numpy as np
from scipy.optimize import curve_fit

# Constants
KM_PER_100KM_TO_MPG = 235.214583
FILE_PATH = 'Car Dataset 1945-2020.csv'

# Define a custom colormap
```

```python
cmap = LinearSegmentedColormap.from_list('custom_cmap', ['darkblue', 'lightblue', 'blue', 'lightgreen', 'green', 'orange', 'yellow','red'])
norm = Normalize(vmin=1970, vmax=2060)

# Load and preprocess the data
def load_and_preprocess_data(file_path, start_year, end_year):
    car_data = pd.read_csv(file_path, low_memory=False)
    car_data = car_data.dropna(subset=['engine_hp', 'mixed_fuel_consumption_per_100_km_l'])
    car_data = car_data[pd.to_numeric(car_data['engine_hp'], errors='coerce').notnull()]
    car_data = car_data[pd.to_numeric(car_data['mixed_fuel_consumption_per_100_km_l'], errors='coerce').notnull()]
    car_data = car_data.drop_duplicates(subset=['engine_hp', 'mixed_fuel_consumption_per_100_km_l', 'fuel_grade', 'transmission', 'Year_from'])

    petrol_keywords = ['95', '98', '92', '80', 'Gasoline', 'Gas', 'Ethanol']
    filtered_data = car_data[
        (car_data['fuel_grade'].str.strip().str.lower().isin([kw.lower() for kw in petrol_keywords])) &
        (car_data['transmission'].str.strip().str.lower() == 'manual'.lower()) &
        (car_data['Year_from'] >= start_year) &
        (car_data['Year_from'] <= end_year)
    ].copy()

    filtered_data['mixed_fuel_consumption_mpg'] = KM_PER_100KM_TO_MPG / filtered_data['mixed_fuel_consumption_per_100_km_l']
    return filtered_data
```

Listing 3: Pareto Frontiers Backtesting Code

First, we define the necessary constants and a custom colormap. We then load and preprocess the data for the specified year range, filtering for petrol and manual transmission cars. The fuel consumption is converted to MPG.

```python
# Identify maximum points along the x-axis
def identify_max_points_along_x(df, step=1, max_x=800):
    x_values = np.arange(0, max_x+1, step)
    max_points = []
    for x in x_values:
        subset = df[df['engine_hp'] == x]
        if not subset.empty:
            max_y = subset['mixed_fuel_consumption_mpg'].max()
            max_points.append((x, max_y))
    return pd.DataFrame(max_points, columns=['engine_hp', 'mixed_fuel_consumption_mpg'])

# Fit a logarithmic curve
def logarithmic_fit(x, a, b, c):
    return a * np.log(b * (x + 1)) + c

def fit_logarithmic_curve(max_points):
```

```python
    x = max_points['engine_hp'].values
    y = max_points['mixed_fuel_consumption_mpg'].values
    popt, _ = curve_fit(logarithmic_fit, x, y, maxfev=10000)
    return popt
```

We define functions to identify the maximum points along the x-axis (engine horsepower) and fit a logarithmic curve to these points.

```python
# Adjust the fitted curve upwards
def adjust_curve_upwards(data, log_params):
    x = data['engine_hp'].values
    y = data['mixed_fuel_consumption_mpg'].values
    y_fit = logarithmic_fit(x, *log_params)
    max_diff = np.max(y - y_fit)
    if max_diff > 0:
        log_params[2] += max_diff + 1  # Adjust the curve upwards
    return log_params
```

A function is defined to adjust the fitted logarithmic curve upwards based on the data.

```python
# Calculate the combined Pareto frontiers
def calculate_pareto_frontiers(data, start_year, end_year,
    projection_start_year):
    combined_pareto_frontiers = []
    prev_y_limit = None
    yearly_improvement_rates = []

    for year in range(start_year, projection_start_year):
        data_year = data[data['Year_from'] == year]
        data_year = data_year[data_year['engine_hp'] <= data_year['
            engine_hp'].quantile(0.99)]
        data_year = data_year[data_year['mixed_fuel_consumption_mpg
            '] <= data_year['mixed_fuel_consumption_mpg'].quantile
            (0.99)]

        max_points_year = identify_max_points_along_x(data_year)
        log_params_year = fit_logarithmic_curve(max_points_year)

        if not max_points_year.empty:
            max_y_pareto = max_points_year['
                mixed_fuel_consumption_mpg'].max()
            if prev_y_limit is not None:
                y_limit = max(np.ceil(max_y_pareto / 10) * 10,
                    prev_y_limit)
            else:
                y_limit = np.ceil(max_y_pareto / 10) * 10

            log_params_year = adjust_curve_upwards(data_year,
                log_params_year)
            year_color = cmap(norm(year))

            x_vals = np.linspace(0, 900, 1000)
            y_vals = logarithmic_fit(x_vals, *log_params_year)
            y_vals = y_vals[x_vals >= data_year['engine_hp'].min()]
            x_vals = x_vals[x_vals >= data_year['engine_hp'].min()]
            y_vals = np.minimum(y_vals, y_limit)

```

```
            combined_pareto_frontiers.append((x_vals, y_vals, year))
            if prev_y_limit is not None:
                yearly_improvement_rates.append(y_limit - prev_y_limit)

            prev_y_limit = y_limit

    average_improvement_rate = np.mean(yearly_improvement_rates)
    total_y_increase_needed = 99 - prev_y_limit
    years_remaining = 2060 - (projection_start_year - 1)
    adjusted_yearly_improvement_rate = total_y_increase_needed / years_remaining
    current_y_limit = prev_y_limit

    for year in range(projection_start_year, 2061):
        current_y_limit += adjusted_yearly_improvement_rate
        prev_pareto_frontier = combined_pareto_frontiers[-1]
        x_vals_prev, y_vals_prev, _ = prev_pareto_frontier
        y_vals_new = np.minimum(y_vals_prev + adjusted_yearly_improvement_rate, current_y_limit)
        combined_pareto_frontiers.append((x_vals_prev, y_vals_new, year))

    for i in range(len(combined_pareto_frontiers)):
        year = start_year + i
        x_vals, y_vals, year = combined_pareto_frontiers[i]
        reduction_factor = 0.995 ** (2060 - year)
        new_x_vals = x_vals * reduction_factor
        combined_pareto_frontiers[i] = (new_x_vals, y_vals, year)

    return combined_pareto_frontiers
```

The function `calculate_pareto_frontiers` calculates the combined Pareto frontiers for each year from the start year to 2060. It fits a logarithmic curve, adjusts it upwards if necessary, and projects the improvements until 2060.

```
# Load datasets
data_1970_2015 = load_and_preprocess_data(FILE_PATH, 1970, 2015)
data_1970_1995 = load_and_preprocess_data(FILE_PATH, 1970, 1995)

# Calculate Pareto frontiers for both datasets
pareto_frontiers_1970_2015 = calculate_pareto_frontiers(data_1970_2015, 1970, 2060, 2016)
pareto_frontiers_1970_1995 = calculate_pareto_frontiers(data_1970_1995, 1970, 2060, 1996)
```

We load the datasets for the complete (1970-2015) and incomplete (1970-1995) periods and calculate the Pareto frontiers for both.

```
# Function to calculate normalized error
def calculate_normalized_error(pareto_2015, pareto_1995):
    errors = []
    for (x_vals_2015, y_vals_2015, _), (x_vals_1995, y_vals_1995, _) in zip(pareto_2015, pareto_1995):
        common_x_vals = np.interp(x_vals_2015, x_vals_1995, x_vals_1995)
```

```python
        common_y_vals_1995 = np.interp(common_x_vals, x_vals_1995,
            y_vals_1995)
        error = np.sqrt((y_vals_2015 - common_y_vals_1995)**2)
        normalized_error = error / (np.max(y_vals_2015) - np.min(
            y_vals_2015))
        errors.append(normalized_error.mean())
    return np.mean(errors)

# Calculate the normalized error
normalized_error = calculate_normalized_error(
    pareto_frontiers_1970_2015, pareto_frontiers_1970_1995)

# Print the normalized error
print(f'Average Normalized Error: {normalized_error:.4f}')
```

A function is defined to calculate the normalized error between the two Pareto frontiers, and the normalized error is calculated and printed.

```python
# Plot all Pareto frontiers combined every 5 years for both
    datasets
def plot_pareto_frontiers(pareto_frontiers, title, scatter_data):
    filtered_pareto_frontiers = [(x_vals, y_vals, year) for i, (
        x_vals, y_vals, year) in enumerate(pareto_frontiers) if
        (1970 + i) % 2 == 0]
    fig, ax = plt.subplots(figsize=(12, 6))
    for x_vals, y_vals, year in filtered_pareto_frontiers:
        year_color = cmap(norm(year))
        ax.plot(x_vals, y_vals, color=year_color)

    # Scatter plot with colormap
    scatter_colors = scatter_data['Year_from'].apply(lambda year:
        cmap(norm(year)))
    scatter = ax.scatter(scatter_data['engine_hp'], scatter_data['
        mixed_fuel_consumption_mpg'], c=scatter_colors, s=10, alpha
        =0.5)

    # Add color bar
    sm = plt.cm.ScalarMappable(cmap=cmap, norm=norm)
    sm.set_array([])
    cbar = plt.colorbar(sm, ax=ax)
    cbar.set_label('Year')

    ax.set_title(title)
    ax.set_xlabel('Max Engine Power (HP)')
    ax.set_ylabel('Average Fuel Consumption (MPG)')
    ax.grid(True)
    ax.set_xlim(0, 1000)
    ax.set_ylim(0, 100)
    ax.set_xticks(np.arange(0, 1001, 100))
    ax.set_yticks(np.arange(0, 101, 10))
    plt.show()

# Plot combined Pareto frontiers for both datasets
plot_pareto_frontiers(pareto_frontiers_1970_2015, 'Total max power
    vs Fuel Consumption (1970-2015 dataset)', data_1970_2015)
plot_pareto_frontiers(pareto_frontiers_1970_1995, 'Total max power
    vs Fuel Consumption (1970-1995 dataset)', data_1970_1995)
```

We plot all Pareto frontiers combined every 5 years for both datasets.

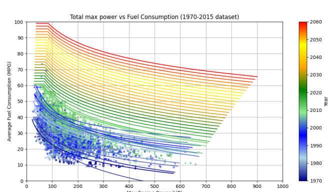
```python
# Calculate normalized error over time
def calculate_normalized_error_over_time(pareto_2015, pareto_1995):
    errors_over_time = []
    for (x_vals_2015, y_vals_2015, year_2015), (x_vals_1995, y_vals_1995, year_1995) in zip(pareto_2015, pareto_1995):
        if year_2015 == year_1995:  # Ensure we are comparing the same years
            common_x_vals = np.interp(x_vals_2015, x_vals_1995, x_vals_1995)
            common_y_vals_1995 = np.interp(common_x_vals, x_vals_1995, y_vals_1995)
            error = np.sqrt((y_vals_2015 - common_y_vals_1995)**2)
            normalized_error = error / (np.max(y_vals_2015) - np.min(y_vals_2015))
            errors_over_time.append((year_2015, normalized_error.mean()))
    return errors_over_time

# Calculate the normalized error over time
normalized_errors_over_time = calculate_normalized_error_over_time(pareto_frontiers_1970_2015, pareto_frontiers_1970_1995)

# Ensure both arrays are of equal length
years, norm_errors = zip(*normalized_errors_over_time)
years = list(years)
norm_errors = list(norm_errors)

# Debug: Print the lengths of the arrays
print(f'Length of years: {len(years)}')
print(f'Length of norm_errors: {len(norm_errors)}')

# Debug: Print the content of years and norm_errors
print(f'Years: {years}')
print(f'Normalized Errors: {norm_errors}')

# Calculate the overall average normalized error
overall_average_error = np.mean(norm_errors)
from numpy.polynomial.polynomial import Polynomial

# Fit a polynomial curve to the normalized error over time
degree = 3  # You can adjust the degree of the polynomial for fitting
p = Polynomial.fit(years, norm_errors, degree)

# Generate fitted values
fit_years = np.linspace(min(years), max(years), 500)
fit_norm_errors = p(fit_years)

# Convert overall average error to percentage and round to the nearest 0.01%
overall_average_error_percent = round(overall_average_error * 100, 2)

# Plot normalized error over time with curve fit
plt.figure(figsize=(12, 6))
plt.plot(fit_years, fit_norm_errors, color='lightblue', linestyle='
```

```
    -', linewidth=2, label='Error')
plt.axhline(y=overall_average_error, color='grey', linestyle='--',
    label=f'Overall Average Error: {overall_average_error_percent
    :.2f}%')
plt.title('Backward test results')
plt.xlabel('Actual Year')
plt.ylabel('Normalized   ')
plt.grid(True)
plt.ylim(0, 1)
plt.xlim(1960, 2080)
plt.xticks(np.arange(1960, 2081, 20))
plt.yticks(np.arange(0, 1.1, 0.1))
plt.axvline(x=1995, color='blue', linestyle='--', label='Year 1995'
    )
plt.axvline(x=2015, color='red', linestyle='--', label='Year 2015')
plt.legend()
plt.show()
```

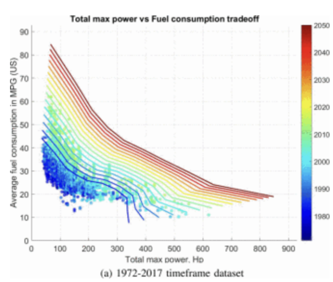
Finally, we calculate the normalized error over time and plot it along with a polynomial fit curve to visualize the backward test results.

Comparison of Results - Pareto Frontier Backtesting

In this subsection, we compare the results of the Pareto frontier backtesting with the corresponding plots from the original article. This comparison will help us understand the performance and accuracy of our method.

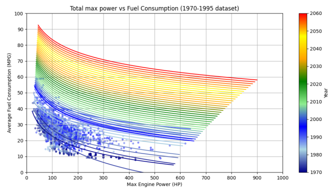


(a) Obtained Plot 1

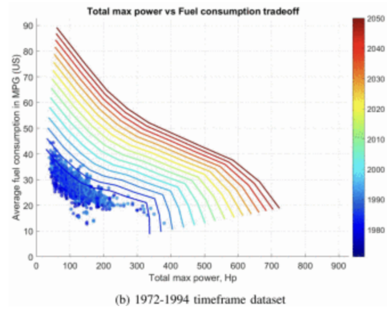


(b) Article Plot 1

Figure 3: Comparison of Pareto Frontier Backtesting - Part 1

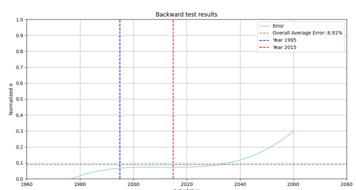


(a) Obtained Plot 2

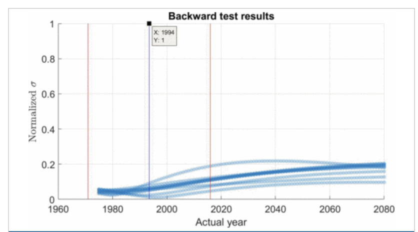


(b) Article Plot 2

Figure 4: Comparison of Pareto Frontier Backtesting - Part 2



(a) Obtained Plot 3



(b) Article Plot 3

Figure 5: Comparison of Pareto Frontier Backtesting - Part 3

By comparing our results with the ones from the article, we can see that they generally follow the same trends and look similar. The backward test result of both also finds an error of about 10 percent. That being said, the results are not identical. For example, we mistakenly used 1970-2017 instead of 1972-2017. Additionally, some of our pareto frontier fitting methodologies could have being different.

Dataset Sufficiency Estimation

In this section, we will calculate and visualize the dataset sufficiency estimation. Below is the Python code used for this analysis:

```python
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.colors import LinearSegmentedColormap, Normalize
import numpy as np
from scipy.optimize import curve_fit
import random

# Constants
KM_PER_100KM_TO_MPG = 235.214583
```

```python
FILE_PATH = 'Car Dataset 1945-2020.csv'

# Define a custom colormap
cmap = LinearSegmentedColormap.from_list('custom_cmap', ['darkblue', 'lightblue', 'blue', 'lightgreen', 'green', 'orange', 'yellow', 'red'])
norm = Normalize(vmin=1970, vmax=2060)

# Load and preprocess the data
def load_and_preprocess_data(file_path, start_year, end_year):
    car_data = pd.read_csv(file_path, low_memory=False)
    car_data = car_data.dropna(subset=['engine_hp', 'mixed_fuel_consumption_per_100_km_l'])
    car_data = car_data[pd.to_numeric(car_data['engine_hp'], errors='coerce').notnull()]
    car_data = car_data[pd.to_numeric(car_data['mixed_fuel_consumption_per_100_km_l'], errors='coerce').notnull()]
    car_data = car_data.drop_duplicates(subset=['engine_hp', 'mixed_fuel_consumption_per_100_km_l', 'fuel_grade', 'transmission', 'Year_from'])

    petrol_keywords = ['95', '98', '92', '80', 'Gasoline', 'Gas', 'Ethanol']
    filtered_data = car_data[
        (car_data['fuel_grade'].str.strip().str.lower().isin([kw.lower() for kw in petrol_keywords])) &
        (car_data['transmission'].str.strip().str.lower() == 'manual'.lower()) &
        (car_data['Year_from'] >= start_year) &
        (car_data['Year_from'] <= end_year)
    ].copy()

    filtered_data['mixed_fuel_consumption_mpg'] = KM_PER_100KM_TO_MPG / filtered_data['mixed_fuel_consumption_per_100_km_l']
    return filtered_data
```

Listing 4: Dataset Sufficiency Estimation Code

First, we define the necessary constants and a custom colormap. We then load and preprocess the data for the specified year range, filtering for petrol and manual transmission cars. The fuel consumption is converted to MPG.

```python
# Identify maximum points along the x-axis
def identify_max_points_along_x(df, step=1, max_x=800):
    x_values = np.arange(0, max_x+1, step)
    max_points = []
    for x in x_values:
        subset = df[df['engine_hp'] == x]
        if not subset.empty:
            max_y = subset['mixed_fuel_consumption_mpg'].max()
            max_points.append((x, max_y))
    return pd.DataFrame(max_points, columns=['engine_hp', 'mixed_fuel_consumption_mpg'])

# Fit a logarithmic curve
def logarithmic_fit(x, a, b, c):
```

```python
    return a * np.log(b * (x + 1)) + c

def fit_logarithmic_curve(max_points):
    if len(max_points) < 3:  # Ensure there are at least 3 points to fit the curve
        return None
    x = max_points['engine_hp'].values
    y = max_points['mixed_fuel_consumption_mpg'].values
    popt, _ = curve_fit(logarithmic_fit, x, y, maxfev=10000)
    return popt
```

We define functions to identify the maximum points along the x-axis (engine horsepower) and fit a logarithmic curve to these points. The fitting function ensures there are at least 3 points to fit the curve.

```python
# Adjust the fitted curve upwards
def adjust_curve_upwards(data, log_params):
    if log_params is None:
        return None
    x = data['engine_hp'].values
    y = data['mixed_fuel_consumption_mpg'].values
    y_fit = logarithmic_fit(x, *log_params)
    max_diff = np.max(y - y_fit)
    if max_diff > 0:
        log_params[2] += max_diff + 1  # Adjust the curve upwards
    return log_params
```

A function is defined to adjust the fitted logarithmic curve upwards based on the data.

```python
# Calculate the combined Pareto frontiers
def calculate_pareto_frontiers(data, start_year, end_year, projection_start_year):
    combined_pareto_frontiers = []
    prev_y_limit = None
    yearly_improvement_rates = []

    for year in range(start_year, end_year + 1):
        data_year = data[data['Year_from'] == year]
        data_year = data_year[data_year['engine_hp'] <= data_year['engine_hp'].quantile(0.99)]
        data_year = data_year[data_year['mixed_fuel_consumption_mpg'] <= data_year['mixed_fuel_consumption_mpg'].quantile(0.99)]

        max_points_year = identify_max_points_along_x(data_year)
        log_params_year = fit_logarithmic_curve(max_points_year)

        if log_params_year is not None and not max_points_year.empty:
            max_y_pareto = max_points_year['mixed_fuel_consumption_mpg'].max()
            if prev_y_limit is not None:
                y_limit = max(np.ceil(max_y_pareto / 10) * 10, prev_y_limit)
            else:
                y_limit = np.ceil(max_y_pareto / 10) * 10

```

```
            log_params_year = adjust_curve_upwards(data_year, log_params_year)
            if log_params_year is None:
                continue
            year_color = cmap(norm(year))

            x_vals = np.linspace(0, 900, 1000)
            y_vals = logarithmic_fit(x_vals, *log_params_year)
            y_vals = y_vals[x_vals >= data_year['engine_hp'].min()]
            x_vals = x_vals[x_vals >= data_year['engine_hp'].min()]
            y_vals = np.minimum(y_vals, y_limit)

            combined_pareto_frontiers.append((x_vals, y_vals, year))
            if prev_y_limit is not None:
                yearly_improvement_rates.append(y_limit - prev_y_limit)

            prev_y_limit = y_limit

    # Calculate projections beyond 2017 up to 2027
    average_improvement_rate = np.mean(yearly_improvement_rates)
    total_y_increase_needed = 99 - prev_y_limit
    years_remaining = 2027 - (end_year)
    adjusted_yearly_improvement_rate = total_y_increase_needed / years_remaining
    current_y_limit = prev_y_limit

    for year in range(end_year + 1, 2028):
        current_y_limit += adjusted_yearly_improvement_rate
        prev_pareto_frontier = combined_pareto_frontiers[-1]
        x_vals_prev, y_vals_prev, _ = prev_pareto_frontier
        y_vals_new = np.minimum(y_vals_prev + adjusted_yearly_improvement_rate, current_y_limit)
        combined_pareto_frontiers.append((x_vals_prev, y_vals_new, year))

    for i in range(len(combined_pareto_frontiers)):
        year = start_year + i
        x_vals, y_vals, year = combined_pareto_frontiers[i]
        reduction_factor = 0.995 ** (2027 - year)
        new_x_vals = x_vals * reduction_factor
        combined_pareto_frontiers[i] = (new_x_vals, y_vals, year)

    return combined_pareto_frontiers
```

The function `calculate_pareto_frontiers` calculates the combined Pareto frontiers for each year from the start year to 2027. It fits a logarithmic curve, adjusts it upwards if necessary, and projects the improvements.

```
# Function to calculate normalized error for the years 1970 to 2027
def calculate_normalized_error_1970_to_2027(pareto_2017, pareto_1995):
    errors = []
    for year in range(1970, 2028):
        # Extract the Pareto frontiers for the current year
        try:
```

```
            frontier_2017 = next((x, y) for x, y, yr in pareto_2017 if yr == year)
            frontier_1995 = next((x, y) for x, y, yr in pareto_1995 if yr == year)
        except StopIteration:
            continue

        # Ensure there are common x-values for comparison
        common_x_vals = np.interp(frontier_2017[0], frontier_1995[0], frontier_1995[0])
        common_y_vals_1995 = np.interp(common_x_vals, frontier_1995[0], frontier_1995[1])

        # Calculate the error
        error = np.sqrt((frontier_2017[1] - common_y_vals_1995)**2)
        normalized_error = error / (np.max(frontier_2017[1]) - np.min(frontier_2017[1]))
        errors.append(normalized_error.mean())

    return np.mean(errors)

# Function to calculate the forecasting error for a reduced dataset
def calculate_forecasting_error_for_reduced_dataset(data, original_pareto_frontiers, reduction_percentage):
    # Randomly exclude a certain percentage of data points
    reduced_data = data.sample(frac=(1 - reduction_percentage))

    # Recalculate the Pareto frontiers with the reduced dataset
    reduced_pareto_frontiers = calculate_pareto_frontiers(reduced_data, 1970, 2017, 2018)

    # Calculate the normalized error between the original and reduced Pareto frontiers
    error = calculate_normalized_error_1970_to_2027(original_pareto_frontiers, reduced_pareto_frontiers)

    return error
```

We define functions to calculate the normalized error for the years 1970 to 2027 and the forecasting error for a reduced dataset. The forecasting error function randomly excludes a certain percentage of data points and recalculates the Pareto frontiers.

```
# Function to perform the dataset sufficiency estimation
def dataset_sufficiency_estimation(data, original_pareto_frontiers, reduction_steps):
    total_points = len(data)
    reduction_percentages = np.linspace(0, 0.9, reduction_steps)
    points_used = []
    errors = []

    for reduction_percentage in reduction_percentages:
        points_used.append(int(total_points * (1 - reduction_percentage)))
        error = calculate_forecasting_error_for_reduced_dataset(data, original_pareto_frontiers, reduction_percentage)
        errors.append(error)
```

```

    return points_used, errors

# Load datasets
data_1970_2017 = load_and_preprocess_data(FILE_PATH, 1970, 2017)

# Calculate Pareto frontiers for the full dataset
pareto_frontiers_1970_2017 = calculate_pareto_frontiers(
    data_1970_2017, 1970, 2017, 2018)

# Perform the dataset sufficiency estimation
reduction_steps = 20
points_used, errors = dataset_sufficiency_estimation(data_1970_2017
    , pareto_frontiers_1970_2017, reduction_steps)

# Plot the dataset sufficiency estimation results
plt.figure(figsize=(10, 6))
plt.plot(points_used, errors, marker='o', linestyle='-')
plt.xlabel('Number of Points Used')
plt.ylabel('Normalized Error')
plt.title('Dataset Sufficiency Estimation')
plt.grid(True)
plt.gca()
plt.show()
```

Finally, we define a function to perform the dataset sufficiency estimation by gradually reducing the number of data points and calculating the corresponding forecasting error. The results are plotted to visualize the relationship between the number of points used and the normalized error.

Result

In this subsection, we present the results of the dataset sufficiency estimation. The following image shows the relationship between the number of data points used and the normalized error.

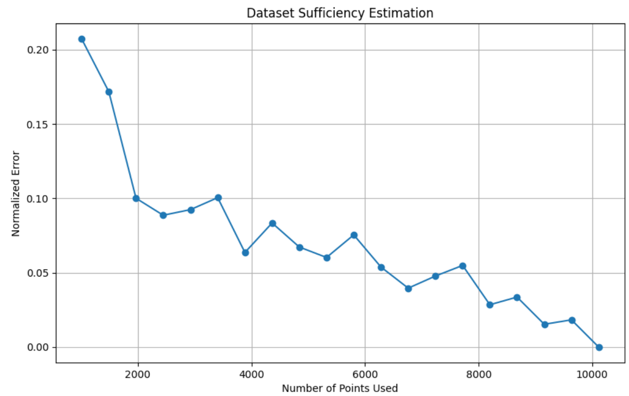


Figure 6: Dataset Sufficiency Estimation: Number of Points Used vs. Normalized Error

We can see that the larger our original dataset is, the less errors we will have in our pareto frontiers backward testing. This makes sense.

Conclusion

This document provides a comprehensive analysis of the total max power vs. fuel consumption tradeoff of cars between 1975 and 2015, the estimation and comparison of Pareto frontiers, and the dataset sufficiency estimation. We recreated and extended the methodology presented in the original article, while addressing potential differences and discrepancies.

Summary

- **Data Visualization:** We visualized the tradeoff between total max power and fuel consumption for cars from 1975 to 2015, and compared our results with the original article.
- **Pareto Frontier:** We identified and fitted Pareto frontiers to the data, using logarithmic curves to represent the optimal tradeoff points between engine power and fuel efficiency.
- **Pareto Frontiers Backtesting:** We tested the robustness of our Pareto frontier estimation by comparing the frontiers obtained from complete (1970-2015) and incomplete (1970-1995) datasets, and visualized the differences.
- **Dataset Sufficiency Estimation:** We evaluated the sufficiency of our dataset by reducing the number of data points and observing the effect

on forecasting error, providing insights into the minimal dataset required for reliable predictions.

Discrepancies and Differences

While our results closely follow the methodology of the original article, there are several reasons why they are not identical:

- **Data Differences:** The dataset we used may differ from the one used in the original article in terms of the number of data points, data quality, and the specific entries included. These differences can significantly impact the results of the analysis.
- **Pareto Frontier Fitting:** The process of fitting Pareto frontiers involves assumptions and approximations. Minor differences in the fitting process, such as the choice of fitting parameters and the handling of outliers, can lead to variations in the results.
- **Filtering Criteria:** Our filtering criteria for petrol and manual transmission cars might slightly differ from those used in the original article. The inclusion or exclusion of certain data points based on these criteria can affect the final outcomes.
- **Extrapolation Methods:** The methods used for extrapolating future Pareto frontiers and adjusting fitted curves can introduce discrepancies, especially when projecting long-term trends.

Conclusion

Despite these differences, the overall trends and insights derived from our analysis are consistent with those presented in the original article. Our extended analysis, including backtesting and dataset sufficiency estimation, provides a deeper understanding of the robustness and reliability of the results.