Big Data Project

Group 8: Car Price Comparison

Introduction

This Jupyter Notebook documents the process of comparing the prices of various BMW models from manufacturer and market data sourced from willhaben.at. The project aims to identify discrepancies and provide insights into pricing strategies in the automotive market.

Team Members

- Boni Vircheva: Data Extraction and Manipulation
- Gabriel Cabraja: Data Extraction and Manipulation
- Jelena Kocic: Data Analysis
- Sara Azzam: Data Analysis

Data Description

The data for our analysis originates from two sources, each offering unique insights into vehicle specifications, pricing, and market trends for BMW models

1. Contains detailed specifications and official prices directly from the manufacturer. Key attributes from this dataset include model, price, year of release, power output (PS), and fuel type. This dataset is typically clean, well-structured, and requires minimal preprocessing before analysis.
2. Consists of prices collected from willhaben.at, an online marketplace. To use this data effectively, we need to undertake several preparatory steps:

Data Acquisition

- Web Scraping: We use Python scripts to scrape vehicle listing data from willhaben.at. This involves programmatically accessing the website and extracting relevant data such as model, listed price, year of release, power output, and fuel type from HTML content.

Data Preprocessing

- Data Extraction: The raw HTML collected during the scraping process is parsed using the CarDataExtractor class, which isolates the necessary information from each listing based on predefined patterns.
- Data Cleaning: Extracted data often contains inconsistencies and errors such as typos, irrelevant characters, or missing values. We use a custom Cleaner class to standardize and clean the data, ensuring attributes like price are numeric and removing any extraneous information.
- Normalization: To facilitate comparisons, we standardize the naming conventions and formats across both datasets. For example, model names and fuel types are normalized to have consistent capitalization and spelling.

Data Storage

- CSV Conversion: After cleaning and normalizing, the data is compiled into a structured format and saved as a CSV file. This file then serves as a standardized source for further analysis and comparison with the manufacturer data.

Integration and Analysis

- Data Merging: We combine the cleaned market data with the manufacturer data to analyze discrepancies in pricing and specifications. This step involves merging datasets based on common attributes like model and year.
- Analytical Processing: Utilizing pandas in Python, we perform various analyses such as price comparisons, trend identification, and statistical summaries to derive insights into market behavior versus official specifications.

By handling data from willhaben.at through these comprehensive steps—scraping, cleaning, normalizing, and storing—we ensure that the market data is robust and reliable for subsequent analytical tasks. This meticulous process aids in our understanding of the automotive market dynamics and supports strategic decision-making based on empirical data.

In [ ]:
```python
import pandas as pd

# Load data
manufacturer_data = pd.read_csv('/path/to/manufacturer_data.csv')
web_data = pd.read_csv('/path/to/web_data.csv')
```

Data Processing with the BMWDataProcessor Class

To effectively manage and analyze data concerning BMW vehicles from both our company's internal databases and external transaction records, we have developed the BMWDataProcessor class. This class streamlines the steps required to clean, merge, and prepare the datasets for in-depth analysis.

Class Structure and Functions

The BMWDataProcessor class encapsulates various methods to process and refine vehicle data. It takes two CSV file paths as inputs, corresponding to the vehicle master list (fahrzeugstamm_path) and transaction records (transaktion_path), and implements the following functionalities:

Initialization

Upon initialization, the class stores the paths to the datasets and initializes placeholders for the merged and cleaned data.

In [ ]:
```
class BMWDataProcessor:
    def __init__(self, fahrzeugstamm_path, transaktion_path):
        self.fahrzeugstamm_path = fahrzeugstamm_path
        self.transaktion_path = transaktion_path
        self.merged_df = None
        self.cleaned_df = None
```

Data Merging

Reads data from the specified CSV files, selects relevant columns, and merges them on the fahrzeug_id key.

In [ ]:
```
def read_and_merge(self):
    fahrzeugstamm_df = pd.read_csv(self.fahrzeugstamm_path)
    transaktion_df = pd.read_csv(self.transaktion_path)
    self.merged_df = pd.merge(fahrzeugstamm_df, transaktion_df, on='fahrz
```

Data Cleaning

Implements multiple data cleaning steps including removing rows with null or zero values, converting data types, and normalizing text fields.

In [ ]:
```
def drop_null_values(self):
    self.merged_df = self.merged_df.dropna()

def additional_cleaning(self):
    self.merged_df['year'] = pd.to_numeric(self.merged_df['year'], errors
    self.merged_df['treibstoff'] = self.merged_df['treibstoff'].apply(lam
```

Filtering Specific Models

Filters the dataset to include only selected BMW models, ensuring that subsequent analyses are focused on relevant data.

In [ ]:
```
def filter_bmw_models(self):
    bmw_models = ['X1', 'X3', '3er-Reihe', '5er-Reihe']
    self.cleaned_df = self.merged_df[self.merged_df['modell'].isin(bmw_mo
```

Data Export

Exports the cleaned and filtered data to a CSV file, making it available for further analysis or reporting.

In [ ]:
```
def export_to_csv(self):
    output_path = 'cleaned_bmw_data.csv'
    self.cleaned_df.to_csv(output_path, index=False)
    print(f"Data exported to CSV at: {output_path}")
```

The BMWDataProcessor class provides a structured approach to handling vehicle data, facilitating efficient data integration, cleaning, and preparation processes. By automating these tasks within a single class, we enhance the reliability and reproducibility of our data analysis workflows, leading to more accurate insights and decision-making based on the processed BMW vehicle data.

Management of Webpage Data for BMW Model Listings

In our project, one crucial component is managing and simulating webpage data for various BMW models. To facilitate this, we developed the Pages class, which stores HTML content for specific models and years. This class acts as a central repository of webpage data, crucial for our data extraction tasks.

The Pages class is designed to hold predefined HTML content that reflects the web pages from which we extract vehicle data. This approach allows us to consistently test and develop our data extraction algorithms without repeatedly querying the website.

Class Definition and Attributes:

- The class is initialized with multiple lists, each designated to store the HTML content of different BMW model pages (e.g., X1, X3, 3er-Reihe, 5er-Reihe). These lists are populated with HTML strings that simulate actual webpage content.
- This structure is particularly useful for ensuring that our data extraction

functions can operate reliably in a controlled environment.

In [ ]:
```
class Pages:
    def __init__(self) -> None:
        self.pages_bmw_dreier_reihe: list = ['
...

<div id="skip-to-resultlist" class="Box-sc-wfmb7k-0 sc-be7b792a-0 eFEoBM
```

Integration with Data Extraction

The Pages class is directly used by our data extraction classes or functions. By accessing these predefined lists, the data extraction logic can simulate reading from a live webpage, allowing us to fine-tune and debug our processes effectively.

In [ ]:
```
pages_instance = Pages()
html_content = pages_instance.pages_bmw_dreier_reihe[0]
extracted_data = extract_function(html_content)
```

The Pages class is a foundational part of our data management strategy, enabling efficient and consistent data extraction for our analysis of BMW vehicle listings. It simplifies the development and testing of our data processing algorithms, ensuring that we can deliver accurate and reliable results in our project.

Purpose and Benefits

- Testing and Development: The HTML stored within these attributes provides a stable, repeatable dataset for testing our data extraction algorithms. This ensures that our methods are robust and perform as expected without the need to access the live website repeatedly.
- Data Consistency: By using fixed data sets, we can easily identify changes in the extraction outcomes triggered by modifications in our parsing logic, rather than variability in the source data.

Data Cleaning

Data cleaning involves removing irrelevant features, handling missing values, and ensuring data type consistency across datasets. Our goal is to gather insights about pricing, performance metrics (PS), model years, and fuel types for BMW vehicles listed on the platform. This information is crucial for market analysis and competitive positioning.

The process involves several key steps: data extraction using regular expressions, data cleaning, data aggregation, and finally data storage. We employ Python with libraries such as pandas for data manipulation and re for pattern matching in HTML content.

Step 1: Data Extraction

We created a CarDataExtractor class to encapsulate the data extraction logic. This class utilizes regular expressions to locate specific data (prices, horsepower, years, and fuel types) within the HTML content of the webpage.

In [ ]:
```
class CarDataExtractor:
    def __init__(self):
        self.valid_years = ['2020', '2021', '2022', '2023']

    def extract_data(self, html_string, model):
        # Initialize lists for data collection
        gather_prices, gather_ps, gather_years, gather_fuel = [], [], [],
        # Regular expression patterns to locate data within the HTML
        pattern_prices = "search-result-entry-price-.{100}"
        ...
        # Find all matches in the string
        matches_prices = re.findall(pattern_prices, html_string)
        ...
        # Add the found data to the lists
        for match in matches_prices:
            gather_prices.append(match)
        ...
        return pd.DataFrame(data)
```

Step 2: Data Cleaning

After extraction, data cleaning is crucial to eliminate any inconsistencies and prepare the data for analysis. For this, we use a helper class Cleaner which provides methods to clean and transform the collected data fields.

In [ ]:
```
# Clean data
cleaner = Cleaner(combined_df)
cleaner.clean_fuel()
cleaner.clean_prices()
cleaner.convert_years_pd_object()
cleaner.check_for_null_values()
```

Step 3: Aggregating Data Across Pages

We aggregate data from multiple pages into a comprehensive DataFrame. This involves extending lists with data from each page and ensuring that all lists are of the same length to form a well-structured DataFrame.

In [ ]:
```
def process_pages(self, pages_dict):
    all_prices, all_years, all_ps, all_fuels = [], [], [], []
    for model_key, page_list in pages_dict.items():
        for page_html in page_list:
            df = self.extract_data(page_html, model_key)
            all_prices.extend(df['Price'].tolist())
            ...
    combined_df = pd.DataFrame({
        'Brand': all_brands,
        'Model': all_models,
        'Price': all_prices,
        ...
    })
    return combined_df
```

Step 4: Exporting Data to CSV

Finally, we export the cleaned and aggregated data to a CSV file for further analysis or reporting.

In [ ]:
```
csv_file_path = '/path/to/save/cleaned_bmw_data_web.csv'
combined_df.to_csv(csv_file_path, index=False)
print(f"Data exported to CSV file at: {csv_file_path}")
```

Analysis of BMW Car Data from Company and Web Datasets

In this section of our project, we aim to analyze and compare BMW vehicle data extracted from both a company's internal database and the external website willhaben.at. Our objectives are to identify differences in pricing, analyze average prices based on various criteria, and assess the availability of different models across both datasets.

Step 1: Data Preparation

Upon initializing the class, we load and preprocess two datasets: one from the company's internal records and one from the web. This includes parsing year data into a datetime format, renaming columns for uniformity, and ensuring numerical fields are properly formatted for calculations.

In [ ]:
```
class AnalyseCarData:
    def __init__(self):
        self.company_df = pd.read_csv('/path/to/company_data.csv')
        self.company_df['year'] = pd.to_datetime(self.company_df['year'],
        self.web_df = pd.read_csv('/path/to/web_data.csv')
        self.web_df['Year'] = pd.to_datetime(self.web_df['Year']).dt.year
        ...
```

Step 2: Filtering for Common Characteristics

We filter both datasets to only include rows where the model, year, horsepower (PS), and fuel type are common across both datasets. This helps in making accurate comparisons and analyses.

In [ ]:
```
self.common_models = set(self.company_df['Model']).intersection(set(self.
self.company_df_filtered = self.company_df[self.company_df['Model'].isin(
self.web_df_filtered = self.web_df[self.web_df['Model'].isin(self.common_
```

Step 3: Data Analysis

3.1 Price Comparison Across Datasets

We compare the average prices of the same models, based on year, PS, and fuel, across the datasets. The differences are calculated and organized in a pivot table for easy comparison.

In [ ]:
```
    def compare_prices_across_datasets(self):
        grouped_prices = self.combined_df.groupby(['Model', 'year', 'PS',
        pivot_prices = grouped_prices.pivot_table(...)
        return pivot_prices
```

3.2 Average Price Analysis

We calculate the average prices of vehicles within each dataset, filtering out those below a threshold to ensure our analysis is focused on relevant data.

In [ ]:
```
    def analyse_company_models(self):
        filtered_company = self.company_df[self.company_df['Price'] > 150
        return filtered_company.groupby([...])['Price'].mean().reset_inde
```

3.3 Model Count Analysis

We count how many different models are available in each dataset, providing insights into the diversity of the offerings.

In [ ]:
```
    def count_bmw_company_models(self):
        return self.company_df.groupby('Model').size().reset_index(name='
```

By comparing the datasets in these ways, we're able to identify key differences and trends in the pricing and availability of BMW models across different sales channels. This analysis provides valuable insights into market dynamics and helps guide strategic decisions regarding inventory and pricing.

Data Visualization

To effectively communicate our findings from the analysis of BMW vehicle data, we have implemented the DataVisualizer class. This class uses visualizations to illustrate various aspects of the data, such as price distributions, average price comparisons, and model counts across different datasets.

The DataVisualizer class is initialized with a DataFrame, which contains merged and analyzed data, and an instance of the AnalyseCarData class to utilize its data analysis functionalities.

In [ ]:
```
class DataVisualizer:
    def __init__(self, dataframe):
        self.df = dataframe
        self.analyzer = AnalyseCarData()
```

Visualization Methods

1. Price Distribution by Model and Year:

This method creates a boxplot to show the distribution of prices for different BMW models over the years. It helps in understanding price variations and trends within the data.

In [ ]:
```
    def plot_price_distribution(self):
    plt.figure(figsize=(10, 6))
    sns.boxplot(x='Model', y='Price', hue='year', data=self.df)
    plt.title('Price in EURO Distribution by Model and Year')
    plt.savefig('price_distribution.png')
    plt.close()
```

2. Average Price Comparison Across Datasets:

This bar plot compares the average prices of BMW models across the datasets (company vs. web). It visualizes the disparities in pricing between internal data and market data.

In [ ]:
```
    def plot_average_price_comparison(self):
    plt.figure(figsize=(14, 8))
    ax = sns.barplot(x='Model', y='Price', hue='Source', data=self.df)
    plt.title('Average Price Comparison Across Datasets')
    plt.ylabel('Average Price')
    plt.xlabel('BMW Model')
    ax.set_xticklabels(ax.get_xticklabels(), rotation=90)
    plt.legend(title='Source')
    plt.tight_layout()
    plt.savefig('average_price_comparison.png')
    plt.close()
```

3. Model Counts in Company and Web Data:

This plot shows the counts of different BMW models available in the company and web datasets. It uses dual bar plots for a direct visual comparison.

In [ ]:
```
    def plot_model_counts(self):
    company_model_counts = self.analyzer.count_bmw_company_models()
    web_model_counts = self.analyzer.count_bmw_web_models()

    fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(14, 6), sharey=Tr
    sns.barplot(x='Model', y='Count', data=company_model_counts, ax=axes[
    axes[0].set_title('Model Counts in Company Data')
    sns.barplot(x='Model', y='Count', data=web_model_counts, ax=axes[1])
    axes[1].set_title('Model Counts in Web Data')

    plt.tight_layout()
    plt.savefig('model_counts_comparison.png')
    plt.close()
```

The DataVisualizer class enhances our understanding of the BMW vehicle data through detailed and intuitive visualizations. By representing the data graphically, we can more easily communicate trends, anomalies, and insights to stakeholders, aiding in data-driven decision-making processes.

Conclusion

This section summarizes the findings of the analysis, discusses any challenges faced, and suggests future work.