YOUR SCIENCE

CREATING OPPORTUNITY

# Applied Analytics
# AI & ML for BI & Automation

## 6. Machine Learning for Supply Chain and Marketing Optimization

Your Science
Mathematical Consulting
Prof. Norbert Poncin
2025

**Your Science Center**: 31, Boulevard Prince Henri, L-1724 Luxembourg
**Phone**: +352 621 674 917, **Email**: info@yourscience.eu, **Website**: yourscience.eu
**Business Permit**: 10154927, **LBR**: A44409, **VAT**: LU35024328

YOUR SCIENCE

CREATING OPPORTUNITY

# Applied Analytics
# AI & ML for BI & Automation

## 6. Machine Learning for Supply Chain and Marketing Optimization

Your Science
Mathematical Consulting
Prof. Norbert Poncin
2025

**Your Science Center**: 31, Boulevard Prince Henri, L-1724 Luxembourg
**Phone**: +352 621 674 917, **Email**: info@yourscience.eu, **Website**: yourscience.eu
**Business Permit**: 10154927, **LBR**: A44409, **VAT**: LU35024328

# Contents

  - Speech recognition (e.g., voice-to-text)
  - Medical diagnosis (e.g., identifying diseases from medical images)
  - Natural language processing (e.g., language translation – increasingly DL-driven)

- **DL (Deep Learning):**
  - Autonomous drones
  - Advanced game playing (e.g., AlphaGo – a computer program that plays the board game Go – uses also Reinforcement Learning techniques)
  - Generative models (e.g., creating art or music)

## Machine Learning and Key Subfields

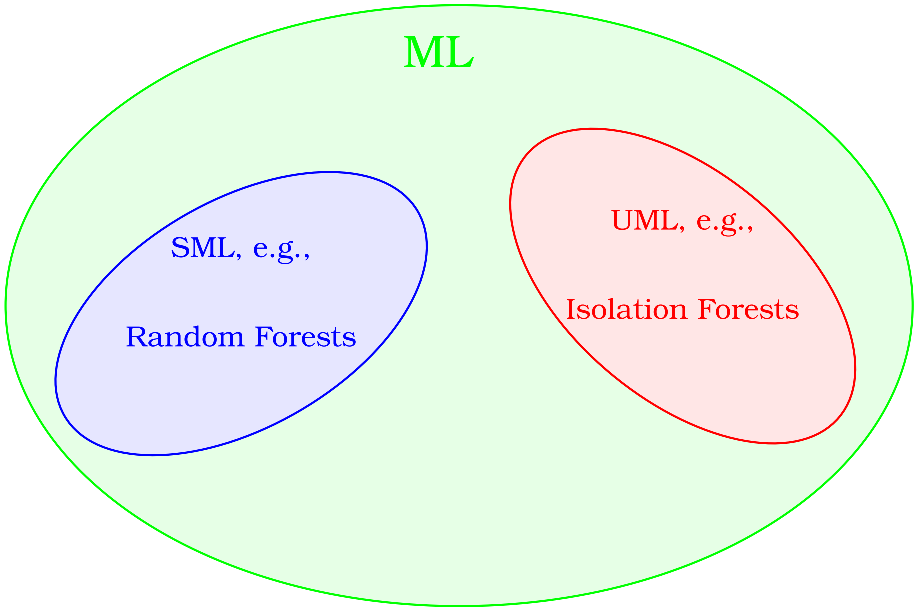


Figure 3: **S**upervised and **U**nsupervised ML

In **Supervised ML** (SML), the model learns patterns from a labeled training set (a dataset where both input features and their corresponding target values are known), is evaluated on a labeled test set (whose labels are used only

```
two_of_three_matching_clients)

    # Metric 3: Matches for product and region only
    product_region_matching_clients = cluster_clients[
        (cluster_clients[dominant_product] == 1) &
        (cluster_clients[dominant_region] == 1)
    ]
    product_region_matches[cluster_id] = len(
product_region_matching_clients)

    # Calculate percentages
    if total_counts[cluster_id] > 0:
        exact_match_percentage = (exact_matches[cluster_id] /
total_counts[cluster_id]) * 100
        two_of_three_match_percentage = (two_of_three_matches[
cluster_id] / total_counts[cluster_id]) * 100
        product_region_match_percentage = (product_region_matches[
cluster_id] / total_counts[cluster_id]) * 100
    else:
        exact_match_percentage = two_of_three_match_percentage =
product_region_match_percentage = 0

    # Print results for this cluster
    print(f"Cluster {cluster_id}:")
    print(f" - Total Customers: {total_counts.get(cluster_id, 0)}")
    print(f" - Exact Matches: {exact_matches.get(cluster_id, 0)} ({
exact_match_percentage:.2f}%)")
    print(f" - Matches for at least 2 out of 3: {two_of_three_matches
.get(cluster_id, 0)} ({two_of_three_match_percentage:.2f}%)")
    print(f" - Matches for Product and Region: {
product_region_matches.get(cluster_id, 0)} ({
product_region_match_percentage:.2f}%)")
```

## Code Output

```
Cluster 0 Characteristics:
 - Dominant Product: Product_Lightweight Papers
 - Dominant Region: Region_Europe
 - Dominant Season: Season_Spring
```

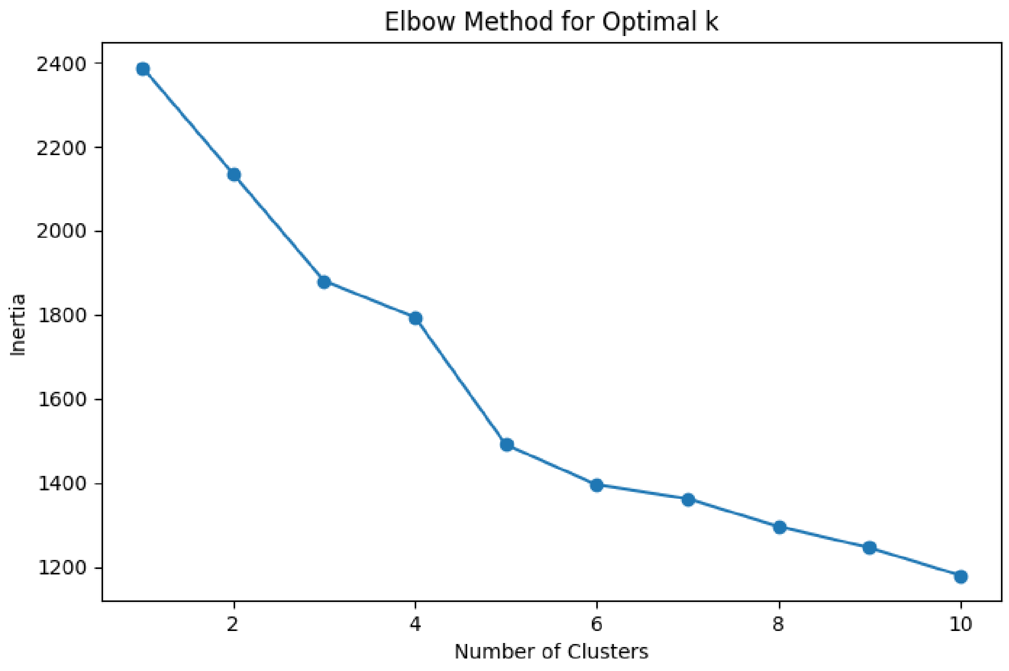


Figure 4

```
Cluster 0:
 - Total Customers: 264
 - Exact Matches: 64 (24.24%)
 - Matches for at least 2 out of 3: 94 (35.61%)
 - Matches for Product and Region: 64 (24.24%)

Cluster 1 Characteristics:
 - Dominant Product: Product_Lightweight Papers
 - Dominant Region: Region_Africa
 - Dominant Season: Season_Winter
Cluster 1:
 - Total Customers: 195
 - Exact Matches: 12 (6.15%)
 - Matches for at least 2 out of 3: 80 (41.03%)
 - Matches for Product and Region: 12 (6.15%)

Cluster 2 Characteristics:
 - Dominant Product: Product_Filter Papers
 - Dominant Region: Region_Asia
 - Dominant Season: Season_Summer
Cluster 2:
 - Total Customers: 157
 - Exact Matches: 13 (8.28%)
 - Matches for at least 2 out of 3: 68 (43.31%)
 - Matches for Product and Region: 13 (8.28%)

Cluster 3 Characteristics:
 - Dominant Product: Product_Botanical Infusion Papers
 - Dominant Region: Region_North America
 - Dominant Season: Season_Winter
Cluster 3:
 - Total Customers: 110
 - Exact Matches: 51 (46.36%)
 - Matches for at least 2 out of 3: 110 (100.00%)
 - Matches for Product and Region: 110 (100.00%)

Cluster 4 Characteristics:
 - Dominant Product: Product_Botanical Infusion Papers
 - Dominant Region: Region_Africa
 - Dominant Season: Season_Autumn
Cluster 4:
 - Total Customers: 191
 - Exact Matches: 7 (3.66%)
 - Matches for at least 2 out of 3: 81 (42.41%)
 - Matches for Product and Region: 7 (3.66%)

Cluster 5 Characteristics:
 - Dominant Product: Product_Lightweight Papers
 - Dominant Region: Region_Europe
 - Dominant Season: Season_Summer
Cluster 5:
 - Total Customers: 83
```

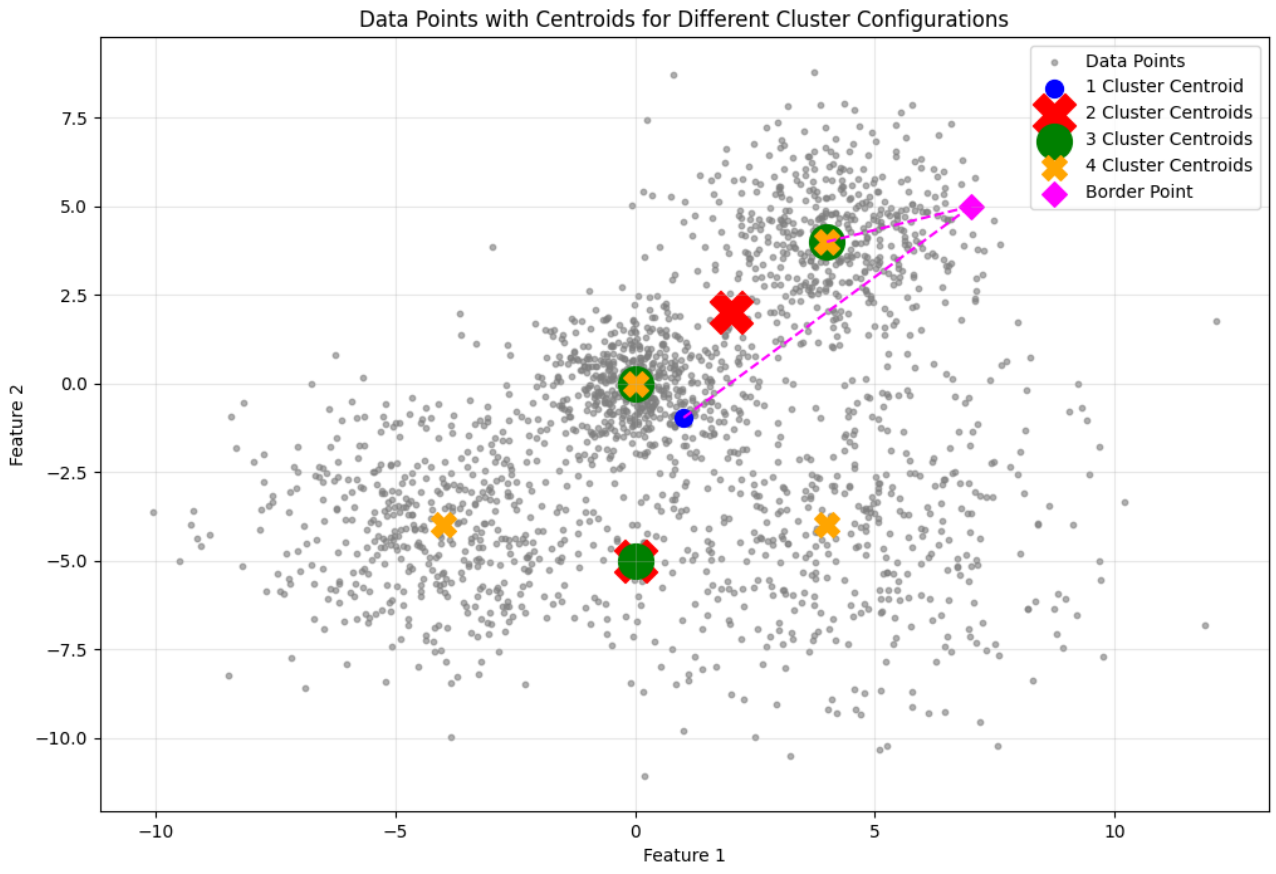


Figure 5: Illustration of Inertia

```
- Exact Matches: 56 (67.47%)
- Matches for at least 2 out of 3: 83 (100.00%)
- Matches for Product and Region: 56 (67.47%)
```

## Explanations

### Elbow Method

The graph in the output plots inertia against the number of clusters, say $k$. Inertia (see Figure 5) represents the sum of squared distances between each data point and the centroid of its assigned cluster, with lower values indicating tighter, more cohesive clusters. Initially, as $k$ increases, inertia decreases sharply, but the rate of reduction diminishes as additional clusters contribute little value. The optimal number of clusters is determined at the elbow point, where the steep decline transitions to a nearly stable decrease (see Figure 4). In our case, the optimal number of clusters is $k = 6$.

## Cluster Analysis

In our artificially generated data, we have encoded four product preferences by region and season:

```
# Region and Season-Specific Product Preference
    if region == 'North America' and season in ['Autumn', 'Winter']:
        product = 'Botanical Infusion Papers'
    elif region == 'Europe' and season in ['Spring', 'Summer']:
        product = 'Lightweight Papers'
```

Observe that the **K-Means++ Unsupervised Machine Learning** clustering algorithm successfully rediscovered three of the four patterns (see Cluster 0, 3, and 5 Characteristics), highlighting the **algorithm's effectiveness**.

At first glance, the relatively low match percentages of cluster members matching their Dominant Product, Region, and Season might seem disappointing. However, with $7$ products, $5$ regions, and $4$ seasons creating $140$ (Product, Region, Season) triplets grouped into just $6$ clusters, i.e., $6$ (Dominant Product, Dominant Region, Dominant Season) triplets, the **granularity** ensures that most clients will not fully match all three dominant features of their assigned cluster.

## K-Means & K-Means++ Clustering Algorithms

K-Means **initializes centroids** randomly, while K-Means++ selects them one at a time, *tending* to place each new centroid farthest from the previously chosen ones. Points are then assigned to the **nearest centroid**, and the **mean point** of each cluster – computed as the point whose coordinates are the means/averages of the respective coordinates of the cluster points – becomes the new centroid. This process repeats until the centroids stabilize.

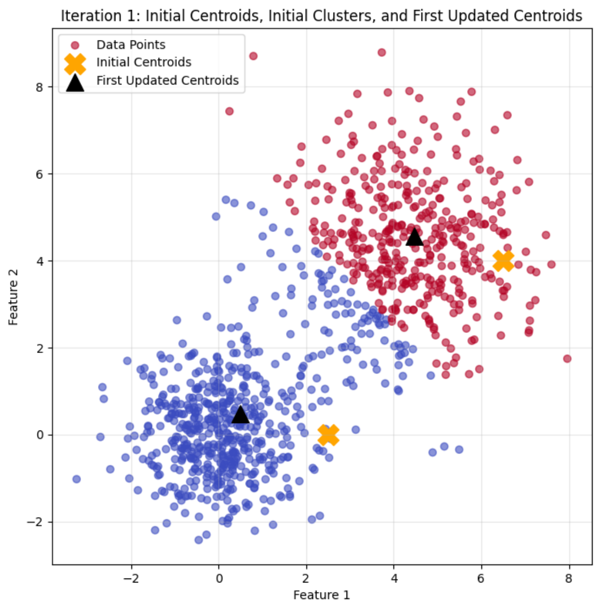


Figure 6: Initial cluster formation and first centroid update

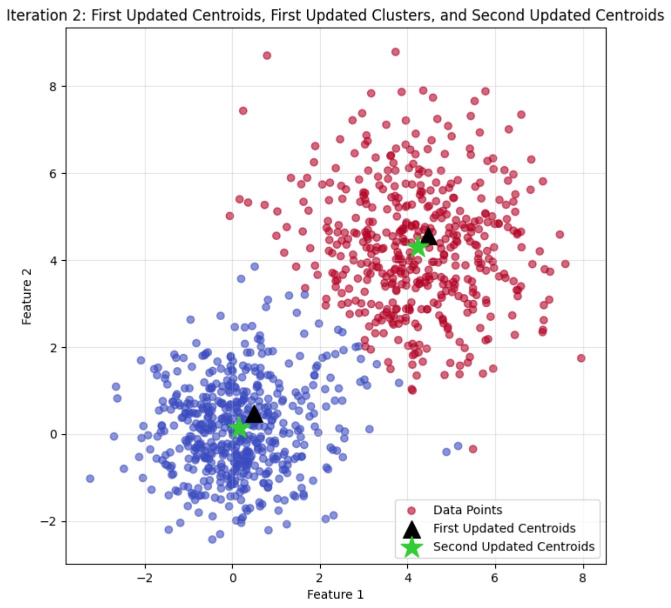


Figure 7: Updated clusters after one iteration and second centroid update

## Automated Email Campaigns

The following code provides a simplified example of automated targeted advertisements. It sends tailored discount ads to the **first customer** in Cluster 0 or 5, as well as the first customer in Cluster 3, who **meet the dominant product and region criteria**. Indeed, customers may be inclined to purchase during a promotion, even if they typically buy in a different season. Replace ZZZZZZZZZ with your password. Perhaps you will find it interesting to explore the code, which utilizes **SMTP, MIME, and HTML** – three key technologies for email communication and formatting?

## Python Code

```python
import smtplib #  SMTP: Simple Mail Transfer Protocol
from email.mime.text import MIMEText # MIME: Multipurpose Internet Mail Extensions - standard that extends email formatting to support text in multiple character sets, attachments, audio, video ...

# Campaign criteria
campaigns = [
    {
        "Product": "Product_Lightweight Papers",
        "Region": "Region_Europe",
        "Seasons": ["Spring", "Summer"],
        "Discount": "20%",
        "Subject": "Exclusive Spring and Summer Discount on Lightweight Papers!",
        "Body": """
        <p>We are excited to offer you an exclusive {discount} discount on <b>{product}</b>, available in <b>{region}</b> this <b>Spring and Summer</b>.</p>
        <p>To take advantage of this special promotion, click the link below for details:</p>
        <p><a href="https://yourscience.eu" target="_blank">Redeem Your Discount Here</a></p>
        """
    },
    {
```

```
To take advantage of this limited-time offer, click the link below
for more details:

Redeem Your Discount Here


Best regards,

Your Mathematical Consulting Team
```

- `Exclusive Spring and Summer Discount on Lightweight Papers!`

```
Dear C0007,

We're excited to offer you an exclusive 20% discount on Lightweight
Papers, available in Europe this Spring and Summer.

To take advantage of this special promotion, click the link below for
details:

Redeem Your Discount Here

Best regards,

Your Mathematical Consulting Team
```

## 2.2 Self-Paced Study: Unsupervised ML – Interactive Visualization

In the example of the previous subsection, clusters are based on three features, Product Preference, Region, and Season, which have 7, 5, and 4 possible values, respectively. When one-hot encoded, these features form therefore a 16-dimensional space (7 + 5 + 4). To visualize the clusters, this space must be reduced to 2 or 3 dimensions.

Commonly used **Unsupervised Machine Learning** algorithms for dimensionality reduction include **Principal Component Analysis (PCA)** and **t-Distributed Stochastic Neighbor Embedding (t-SNE)**.

### 2.2.1 Principal Component Analysis

**Principal Component Analysis** (PCA) is a technique used to compute new axes in a dataset, called Principal Components (PCs).

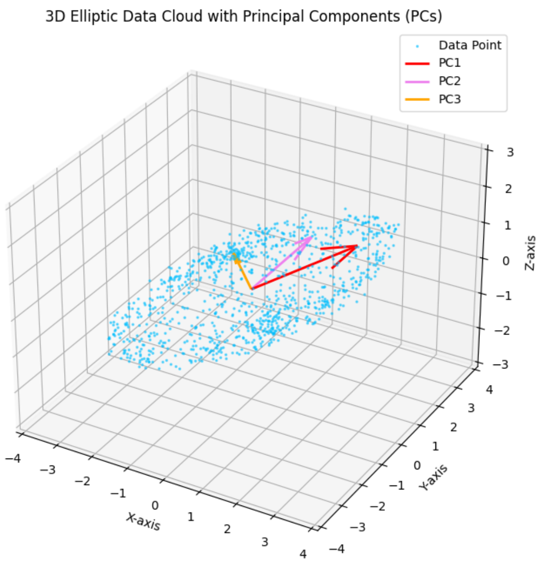
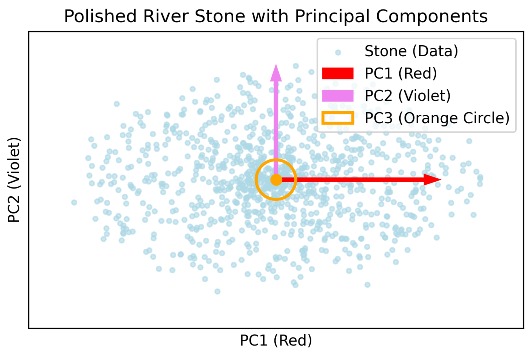


Figure 8: Principle Components and Variance

0.



The first principal component (PC1 – in red) captures the direction of **greatest variance** in the data, which, as shown in Figure 8, forms in the case considered an ellipsoid with clearly distinguishable long, medium, and short axes, resembling a polished river stone. Each subsequent PC (PC2 – in violet, and PC3 – in orange) captures **progressively less variance**. By projecting the data onto the first two PCs, we can reduce the number of dimensions from 3 to 2. This allows us to visualize the data on a 2D plane while capturing most of the variance in the original dataset.

### 2.2.2 t-Distributed Stochastic Neighbor Embedding

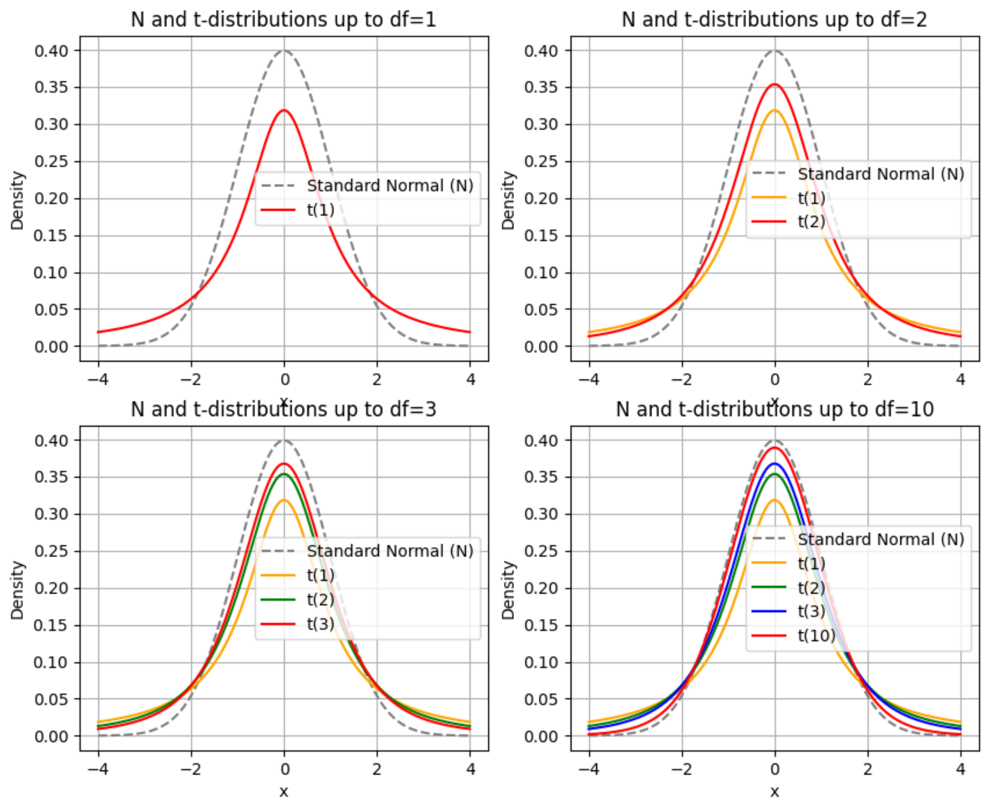


Figure 9: Student’s t-distributions and Standard Normal Distribution

**t-Distributed Stochastic Neighbor Embedding** (t-SNE) is a technique for reducing data dimensionality (e.g., from 16 dimensions to 2 or 3). It ‘projects’ nearby points in the high-dimensional space to nearby points in a low-dimensional space, while mapping distant points in the high-dimen-

```
    opacity=0.7
)

# Customize hover information
fig.update_traces(marker=dict(size=6), selector=dict(mode='markers'))

# Add rotation controls
fig.update_layout(scene=dict(
    xaxis_title="t-SNE Component 1",
    yaxis_title="t-SNE Component 2",
    zaxis_title="t-SNE Component 3"
))

# Save and Show the plot
fig.write_html("C:/Users/norbert.poncin/Downloads/3D_scatter_plot.
    html")
fig.show()
```

## Code Output

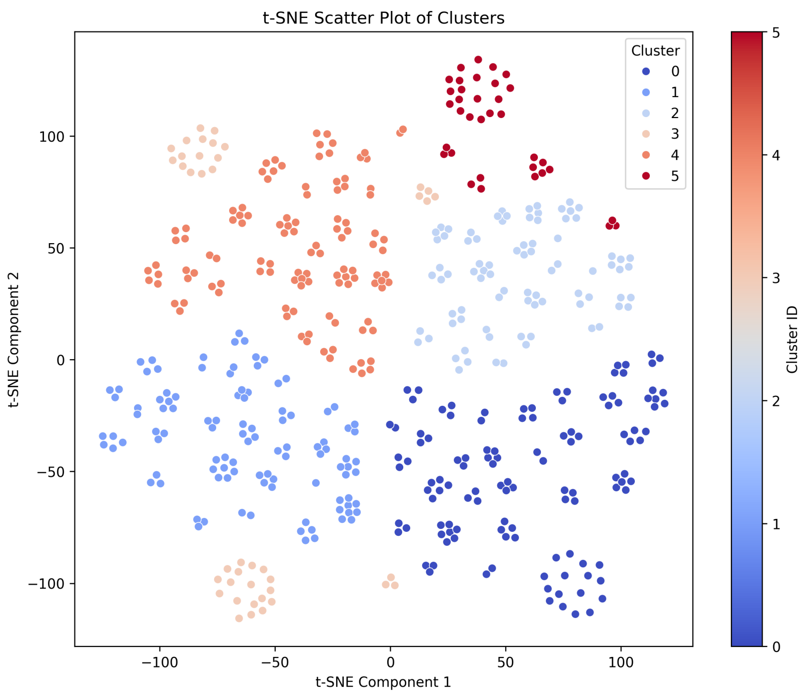


Figure 10: 2D t-SNE Cluster Scatter Plot

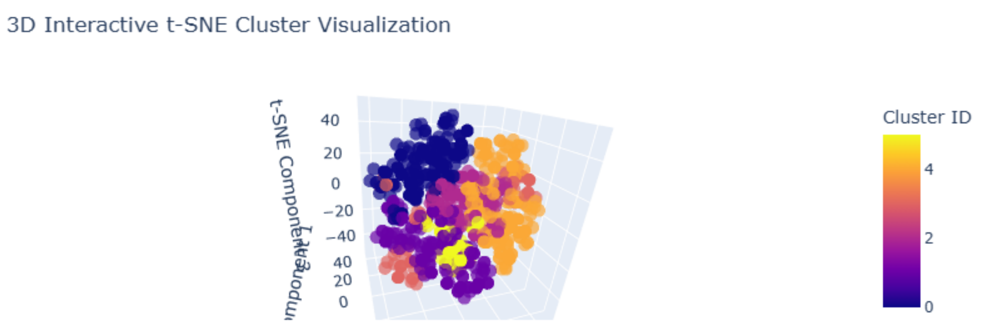


Figure 11: 3D t-SNE Cluster Scatter Plot

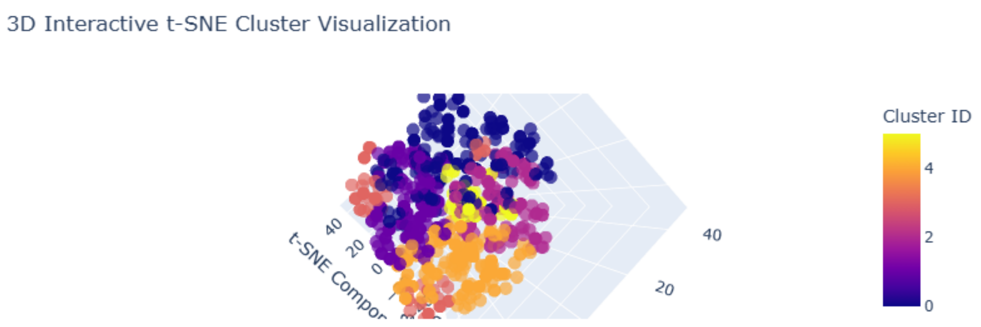


Figure 12: Rotated 3D t-SNE Scatter Plot

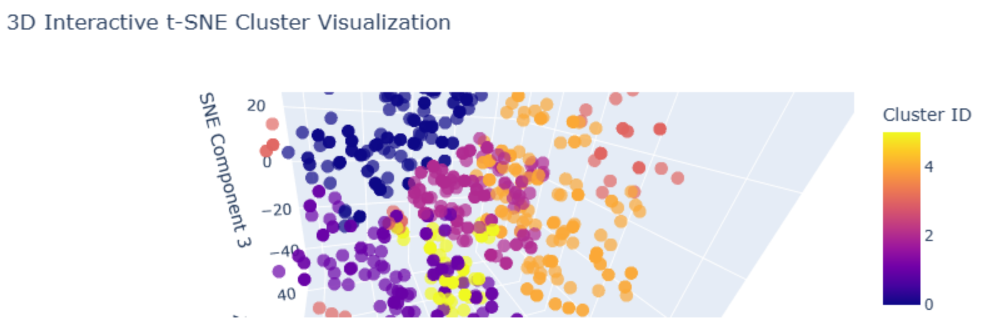


Figure 13: Zoomed-In 3D t-SNE Scatter Plot

## 2.3 Self-Paced Exercise: Unsupervised ML – Purchase Behavior

In the code above, we simulated real-world data by embedding industry-specific patterns:

```python
# Encoded Industry-Specific Patterns

    if industry == 'Food & Beverage':
        purchase_frequency = np.random.poisson(15)
        average_expenditure = np.random.normal(300, 50)
        sales_channels = np.random.choice(['Direct', 'Distributor'],
    p=[0.50, 0.50])
    elif industry == 'Pharmaceuticals':
        purchase_frequency = np.random.poisson(8)
        average_expenditure = np.random.normal(700, 100)
        sales_channels = np.random.choice(['Direct', 'Distributor'],
    p=[0.95, 0.05])
    elif industry == 'Industrial Manufacturing':
        purchase_frequency = np.random.poisson(10)
        average_expenditure = np.random.normal(500, 80)
        sales_channels = np.random.choice(['Direct', 'Distributor'],
    p=[0.75, 0.25])
    else:  # i.e., industry == 'Retail'
        purchase_frequency = np.random.poisson(17)
        average_expenditure = np.random.normal(200, 70)
        sales_channels = np.random.choice(['Direct', 'Distributor'],
    p=[0.25, 0.75])
```

The variable '`Average_Expenditure`' is modeled as following one of the **Gaussian (normal) distributions**

$$\mathcal{N}(200, 70), \quad \mathcal{N}(300, 50), \quad \mathcal{N}(500, 80), \text{ or } \quad \mathcal{N}(700, 100),$$

depending on whether the customer's '`industry`' is *Retail*, *Food & Beverage*, *Industrial Manufacturing*, or *Pharmaceuticals*, respectively. Here, the first parameter in the distribution represents the customers' mean '`Average_Expenditure`' (mean of the average expenditures across all customers from the industry considered), and the second represents the standard deviation of a customer's expenditure from the mean. Thus, our model is a mixture of Gaussian distributions, also known as a **Gaussian Mixture Model** (GMM).

Our next goal is to *cluster manufacturing data using numerical criteria* such as '`Average_Expenditure`' or '`Purchase_Frequency`'. While K-Means is commonly used as a baseline method, Gaussian Mixture Models often perform better, provided clustering features follow a Gaussian distribution conditioned on a hidden categorical feature, such as '`Industry`', in our case. However, this relationship between industries and specific Gaussian distributions is typically unknown and must be inferred from the data. Note finally that the term Gaussian Mixture Model refers not only to the modeling of '`Average_Expenditure`' but also to the algorithm used for clustering such data. In other words, a GMM is both a probabilistic model for our simulated expenditure and an **Unsupervised Machine Learning** algorithm commonly applied to probabilistic clustering.

**Exercise 3.** *Write Python code to group data in a DataFrame '*`df`*' by '*`Industry`*', perform a Shapiro-Wilk test to assess the normality of '*`Average_Expenditure`*' within each industry, and visualize the results using histograms, saving the plot to the Downloads folder.*

### A Possible Python Code

```python
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from scipy.stats import shapiro
import os

normality_path = os.path.join(os.path.expanduser('~'), 'Downloads', '
    normality_test.png')

# Group data by Industry and examine normality
industries = df['Industry'].unique()
variables = ['Average_Expenditure']

results = []

# Perform Shapiro-Wilk test for each variable in each industry
```

```python
for industry in industries:
    industry_data = df[df['Industry'] == industry]
    # If industry = Retail, for example, industry_data contains only
    the rows in df that correspond to retail
    for var in variables:
        stat, p_value = shapiro(industry_data[var])
        # Selects in industry_data the column `Average_Expenditure'
        results.append({
            'Industry': industry,
            'Variable': var,
            'Statistic': stat,
            'P-Value': p_value,
            'Normality': 'Yes' if p_value > 0.05 else 'No'
        })

# Convert results to a DataFrame for display
normality_results = pd.DataFrame(results)
print("Normality Test Results:")
print(normality_results)

# Visualizations: Histograms and KDE plots
for var in variables:
    plt.figure(figsize=(12, 8))
    sns.histplot(data=df, x=var, hue='Industry', kde=True, alpha=0.6,
     bins=30)
    plt.title(f'Distribution of {var} by Industry')
    plt.xlabel(var)
    plt.ylabel('Frequency')
    plt.tight_layout()
    plt.savefig(normality_path)
    plt.show()
```

### Code Output

```
Normality Test Results:
                  Industry             Variable  Statistic   P-Value  \
0          Food & Beverage  Average_Expenditure   0.997576  0.966780
1                   Retail  Average_Expenditure   0.990324  0.099753
2  Industrial Manufacturing  Average_Expenditure   0.995732  0.729821
3          Pharmaceuticals  Average_Expenditure   0.996900  0.916224
```

```
  Normality
0       Yes
1       Yes
2       Yes
3       Yes
```

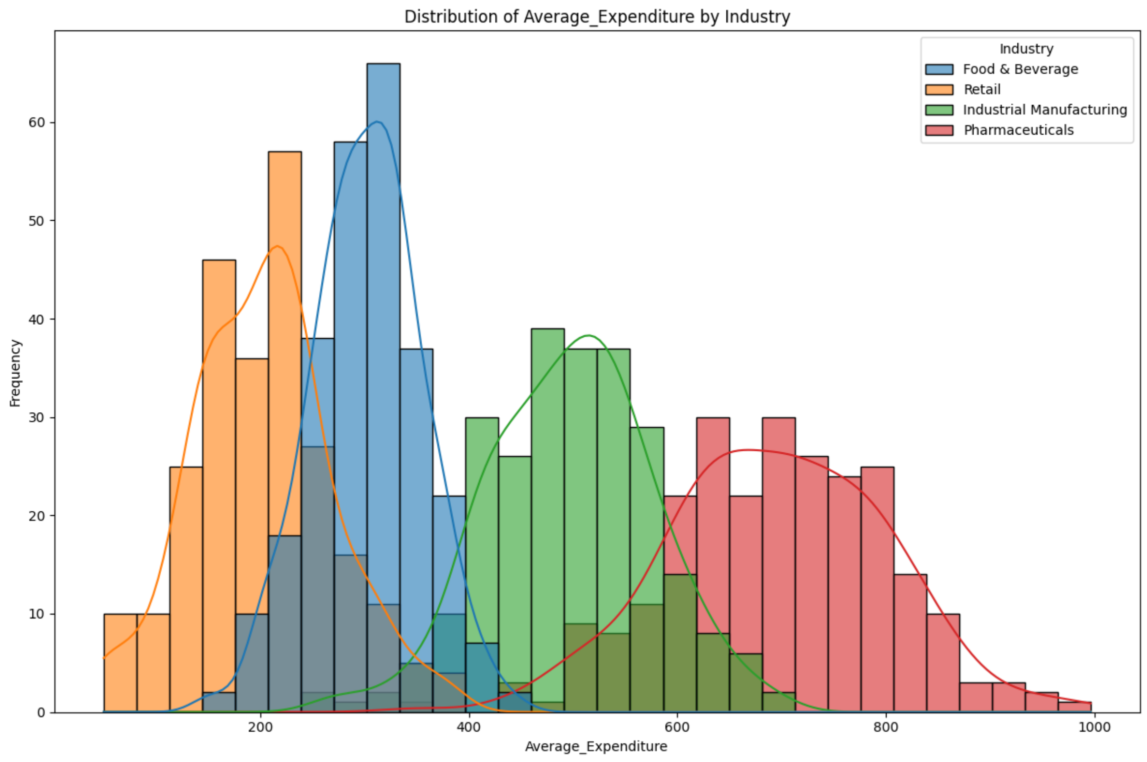


Figure 14: Shapiro-Wilk Normality Test

**Exercise 4.** *Since ‘`Average_Expenditure`‘ follows a Gaussian distribution with parameters specific to each industry, the assumptions of a GMM are satisfied. Use the ‘`GaussianMixture`’ class from the ‘`sklearn.mixture`’ module to cluster customers into four clusters based on ‘`Average_Expenditure`‘, and evaluate the clustering by comparing the predicted clusters to the encoded industry categories.*

## A Possible Python Code

```python
from sklearn.mixture import GaussianMixture
import matplotlib.pyplot as plt
import seaborn as sns

# GMM clustering based on average_expenditure
gmm_exp = GaussianMixture(n_components=4, random_state=42)
df['Cluster'] = gmm_exp.fit_predict(df[['Average_Expenditure']])

# Cluster characteristics
cluster_summary = df.groupby('Cluster').agg(
    Mean_Expenditure=('Average_Expenditure', 'mean'),
    Mean_Frequency=('Purchase_Frequency', 'mean'),
    Modal_Industry=('Industry', lambda x: x.mode()[0])
)
print("Cluster Characteristics:\n", cluster_summary)

# Probability densities of clusters
df['Cluster_Probability'] = gmm_exp.predict_proba(df[['
    Average_Expenditure']]).max(axis=1)

# Visualization
plt.figure(figsize=(10, 6))
sns.scatterplot(data=df, x='Average_Expenditure', y='
    Purchase_Frequency', hue='Cluster', palette='viridis', alpha=0.7)
plt.title('GMM Clustering Based on Average Expenditure')
plt.xlabel('Average Expenditure')
plt.ylabel('Purchase Frequency')
plt.legend(title='Cluster')
plt.savefig(r'C:/Users/norbert.poncin/Downloads/
    gmm_expenditure_clusters.png')
plt.show()
```

## Code Output

```
Cluster Characteristics:
          Mean_Expenditure  Mean_Frequency            Modal_Industry
Cluster
0              734.280667        7.748718          Pharmaceuticals
```

```
1             324.417464      14.778571            Food & Beverage
2             521.920491       9.716981   Industrial Manufacturing
3             191.672308      16.465385                     Retail
```

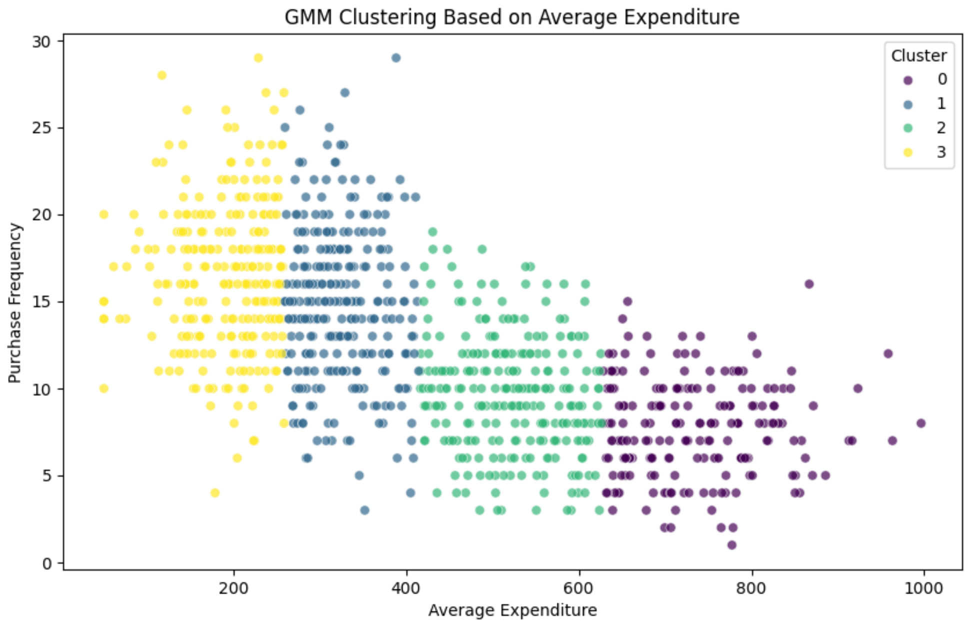


Figure 15: 2D GMM Scatter Plot

## Simplified Marketing Strategy

Our quick exploratory analysis identifies two main customer types:

- **Frequent, low-spending customers**: Increase their value through tiered discounts or bundled offers that encourage higher spending per transaction.
- **Infrequent, high-spending customers**: Foster engagement with exclusive offers and targeted relationship-building efforts, focusing on repeat purchase incentives and simplified bulk ordering.

While these insights provide a preliminary framework, a much deeper and comprehensive analysis is indispensable to develop a robust and actionable business strategy that addresses real-world complexities.

## 2.4 High-Profile Customers

High-profile customers, defined by high ‘`purchase_frequency`’ and high ‘`average_expenditure`’, are valuable assets that deserve special attention. Their unique spending patterns often classify them as outliers. Leveraging the ability of ‘Density-Based Spatial Clustering’ to detect low-density points allows us to identify these valuable customers.

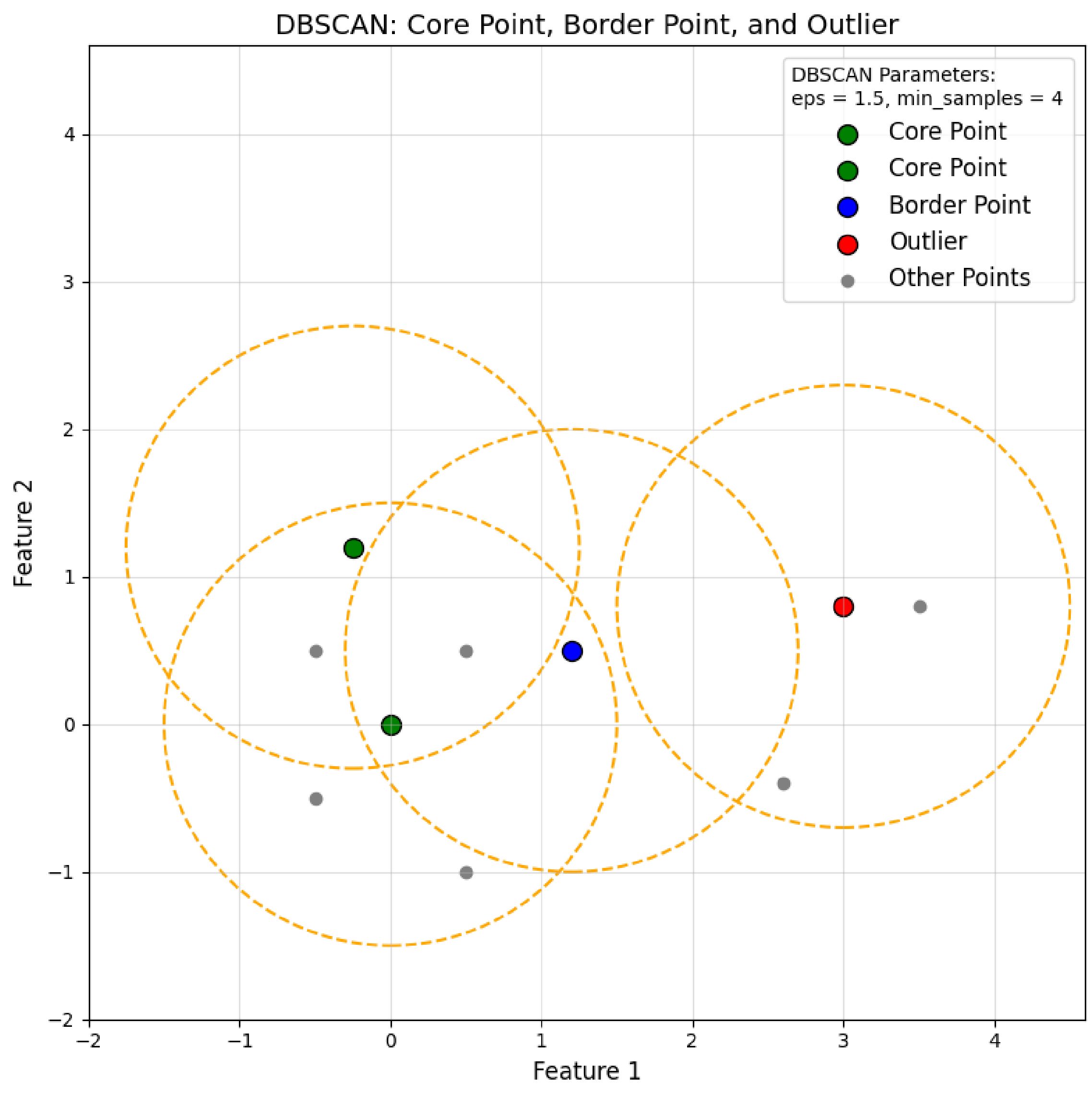


Figure 16

**Density-Based Spatial Clustering of Applications with Noise** (DB-

SCAN) is an **Unsupervised Machine Learning** algorithm used for density-based clustering and outlier detection, classifying points into three categories:

- Core Points: A point is a core point if at least `min_samples` points (including itself) exist within its `eps`-radius (epsilon-radius).
- Border Points: A point is a border point if it lies within the `eps`-radius of a core point but does not have enough neighbors to qualify as a core point itself.
- Noise Points: Points that are neither core nor border points are labeled as *noise* or *outliers*.

**Clusters** are formed as follows:

1. Core points within `eps`-distance of each other are connected to form the dense regions of a cluster.
2. Border points that fall within the `eps`-radius of a core point of the cluster are included in the cluster.

DBSCAN is effective at identifying arbitrarily shaped clusters and isolating outliers. However, it is sensitive to the choice of the parameters `eps` and `min_samples`, which must be carefully tuned to suit the dataset.

## Python Code

```python
from sklearn.cluster import DBSCAN
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Select features for DBSCAN
X = df[['Average_Expenditure', 'Purchase_Frequency']].values

# Standardize the data to ensure both features are on a comparable
    scale
from sklearn.preprocessing import StandardScaler
```

```python
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Apply DBSCAN
dbscan = DBSCAN(eps=0.5, min_samples=2)  # Adjust `eps` and `
    min_samples` as needed
dbscan_labels = dbscan.fit_predict(X_scaled)

# Add the cluster labels to the DataFrame
df['DBSCAN_Label'] = dbscan_labels

# Identify high-profile customers (outliers are labeled as -1 in
    DBSCAN)
high_profile_customers = df[df['DBSCAN_Label'] == -1]
print("High-Profile Customers (Outliers):")
print(high_profile_customers[['Customer_ID', 'Average_Expenditure', '
    Purchase_Frequency']])

# Visualize the clusters
plt.figure(figsize=(10, 6))
sns.scatterplot(
    x='Average_Expenditure',
    y='Purchase_Frequency',
    hue='DBSCAN_Label',
    data=df,
    palette='viridis',
    alpha=0.7
)
plt.title('DBSCAN Clustering: High-Profile Customer Detection')
plt.xlabel('Average Expenditure')
plt.ylabel('Purchase Frequency')
plt.legend(title='Cluster')
plt.tight_layout()

# Save the figure to the Downloads folder
import os
downloads_path = os.path.join(os.path.expanduser('~'), 'Downloads')
plt.savefig(os.path.join(downloads_path, '
    dbscan_high_profile_customers.png'))
plt.show()
```

```
High-Profile Customers (Outliers):
    Customer_ID  Average_Expenditure  Purchase_Frequency
485       C0486               866.91                  16
```

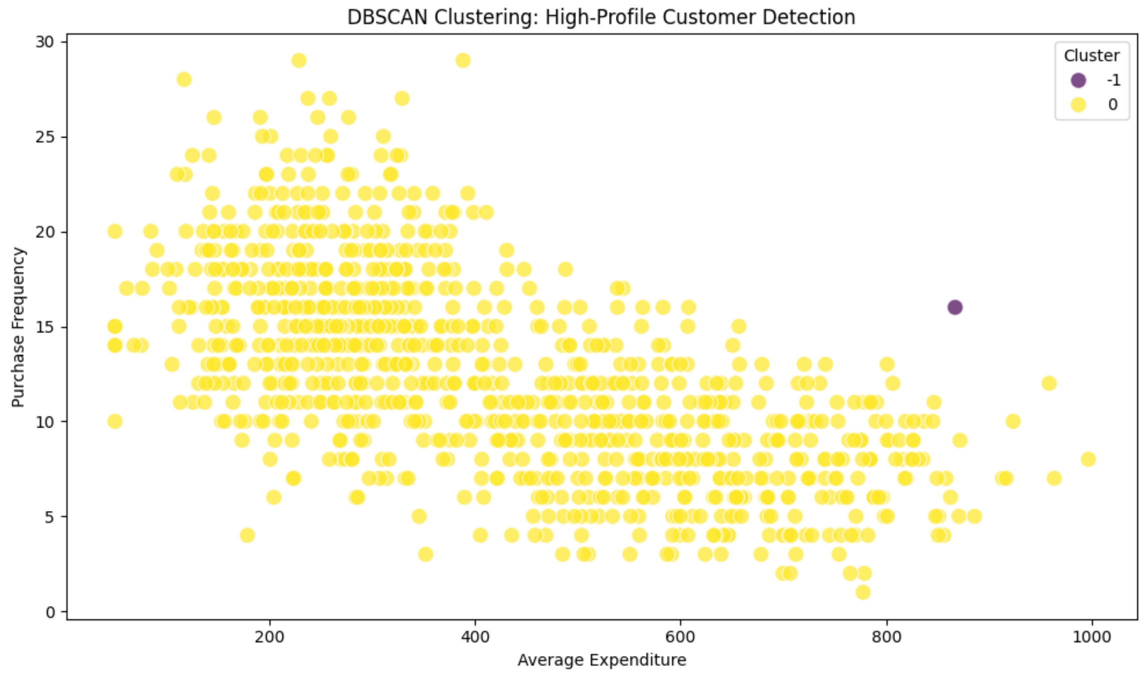


Figure 17: DBSCAN and Outliers

## 3 Supervised ML – Automated Supply Chain Management

We encourage readers to explore and implement selected aspects of the following **Research Project on Automated Supply Chain Management**. While full-scale deployment demands **EXPERT KNOWLEDGE AND ADVANCED RESEARCH SKILLS**, engaging with specific components offers a rewarding challenge that DEEPENS UNDERSTANDING AND ENHANCES PRACTICAL CAPABILITIES.

- **Demand Forecasting**: A combination of statistical methods (e.g., time series analysis, ARIMA) and machine learning models (e.g., regression models, GBMs, LSTM networks) analyzes historical data, market trends, and external factors (e.g., geopolitical issues, pandemics) to generate accurate predictions.

- **Inventory Automation**: A Python-based AI system streamlines inventory management by integrating real-time ERP data and automating reordering via APIs. It optimizes reorder points and quantities while handling uncertainties, anomalies, and stockout risks.
- **Route Optimization**: A tailored Python system enhances standard software capabilities (e.g., real-time traffic and weather data integration) by incorporating company-specific constraints (e.g., multi-stop routing, vehicle limitations). It uses advanced optimization techniques (e.g., Dijkstra's algorithm, A*, genetic algorithms, reinforcement learning) and tools (e.g., NetworkX, Google OR-Tools, TensorFlow), selecting the best approach based on the problem's requirements.
- **Risk Mitigation**: An AI-driven framework detects supply chain risks (e.g., geopolitical issues, pandemics, natural disasters) and suggests contingency plans using predictive analytics with Scikit-learn (forecasting vulnerabilities), anomaly detection with Isolation Forests/Autoencoders (identifying early signs of disruption), and scenario analysis with Monte Carlo simulations (modeling and preparing for potential disruptions).
- **Dynamic Pricing**: An AI system adjusts prices in real-time using historical sales data, competitor pricing, and market trends, or even more subtle criteria like inventory status, demand timing, and customer behavior. It employs regression models, decision trees, random forests, and reinforcement learning to optimize pricing strategies, selecting the most effective approach, leveraging tools such as Scikit-learn, XGBoost (for batch updates), and TensorFlow (for deep learning-based pricing models) when appropriate.

## 3.1 Demand Forecasting

While a first test implementation is partially available for the initial aspect of the research project, incorporating **advanced supervised ML mod-**

**els** such as GBMs and LSTMs, the subsequent points are only preliminarily addressed or left entirely to the reader. These aspects can be further developed by Your Science | Mathematical Consulting upon request.

### 3.1.1 Real-World Data Generation

Our simulated data from the automotive industry covers a 10-year period from 2014 to 2024 and is modeled as follows:

```
# Base demand with seasonal highs in April and October and lows in January and July
data['base_demand'] = 1000 + 200 * np.abs(np.sin(np.pi * data['index'] / 6))

# Market trend following steady growth, dynamically adjusted by temporary downturn factors (e.g., pandemics, geopolitical shifts, natural disasters)
data['actual_demand'] = (data['base_demand'] + data['market_trend']) * data['pandemic_factor'] * data['geo_factor'] * data['natural_disaster_factor']

# Incorporating real-world variability with random normally distributed noise
data['actual_demand'] += np.random.normal(0, 50, size=len(data))
```

The '`actual_demand`' in Month **86**, for instance, could be computed as:

$$86 \qquad [(1000+175)+90]*0.6*1*0.7-125\,.$$

### 3.1.2 Supervised Learning Models

We employ three **Supervised Machine Learning** models to predict `actual_demand` and compare their performance using Root Mean Square Error (RMSE) as the evaluation metric. The first two models are Linear Regression and Random Forest. The third is a hybrid model combining a Gradient Boosting Machine (GBM) – well-suited for handling *relatively static data* – and a Long Short-Term Memory (LSTM) network – effective for capturing *dynamic, rapidly changing patterns*.

### 3.1.3 Stabilizing Data for GBM – Moving Window Average

Here is the Python code for *smoothing fluctuations* using the Moving Window Average technique, followed by an illustration of the method:

```python
data['moving_avg_3'] = data['actual_demand'].rolling(window=3).mean()
```

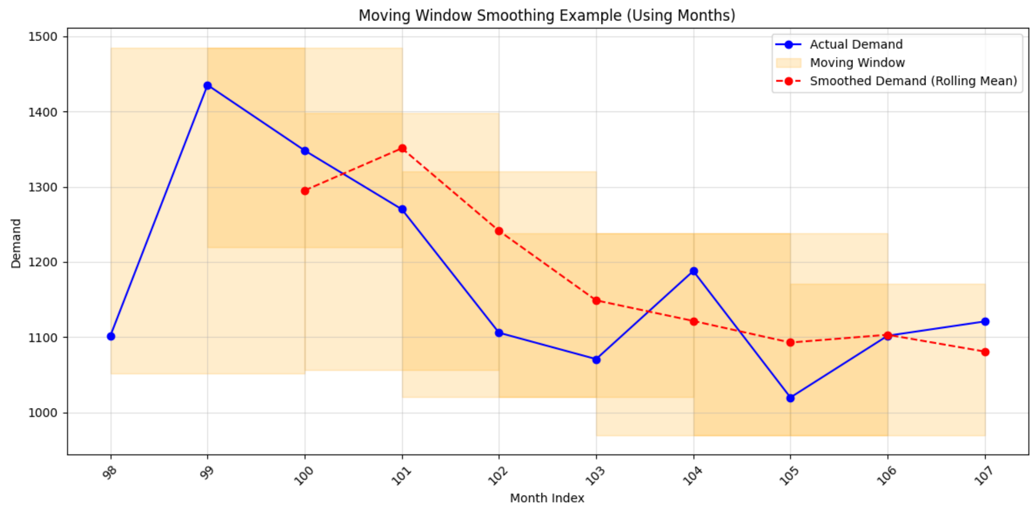


Figure 18: Illustration of Smoothing Data via Moving Window Average

The code creates a rolling window of three months, computes the mean of the actual demand within this window, and assigns the computed average to the last month of each window. For example, if the actual demand is $[100, 300, 200, 600, 200, 500]$, the resulting 3-month moving average is $[\mathrm{NaN}, \mathrm{NaN}, 200, 367, 333, 433]$, exhibiting smoother variations compared to the original demand.

Data such as '`moving_avg_3`' (red curve), '`base_demand`' (including seasonality), and '`market_trend`' are less dynamic compared to '`actual_demand`' (blue curve). They are referred to as **static data** and are particularly well-suited for our first prediction model, the **GBM**, which will predict '`actual_demand`' using these static data as predictors.

### 3.1.4 Sequential Data Preparation for LSTM – Lagging

Here is the Python code for creating Lags (French: `décalages temporels`, German: `Zeitverzögerungen`) of the '`actual_demand`'. Below, you will also find an illustration of the process:

```python
lags = 3
for lag in range(1, lags + 1):
    data[f'lag_{lag}'] = data['actual_demand'].shift(lag)
data.dropna(inplace=True)
```

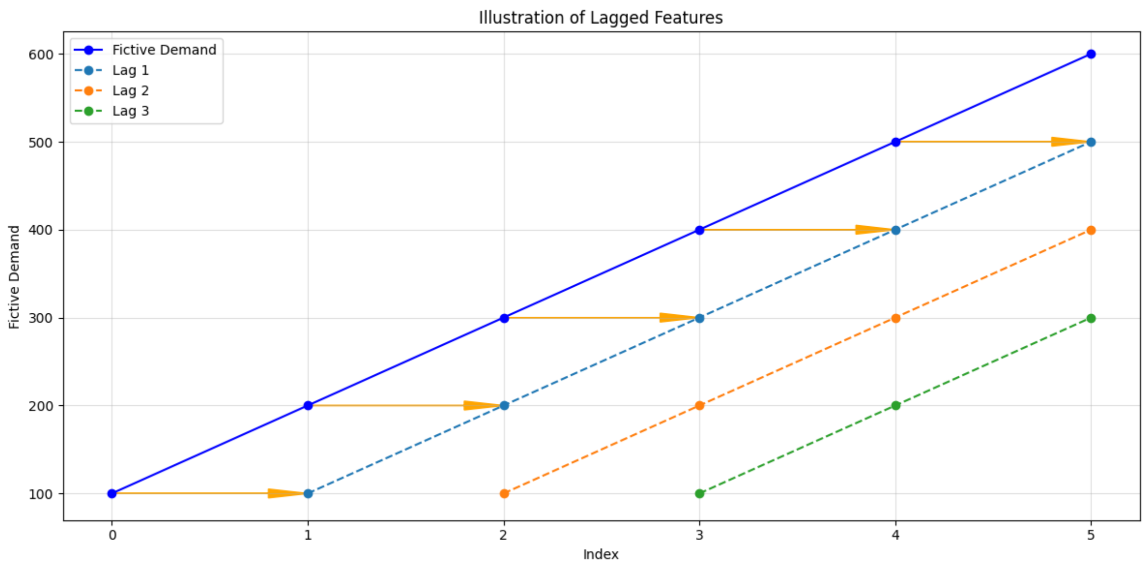


Figure 19: Illustration of Preparing Data via Lagging

Our second prediction model, **LSTM**, leverages sequential '`actual_demand`' data. Although '`base_demand`' and '`market_trend`' also evolve over time, they exhibit more structured and predictable variations and are therefore treated as static features in the present context rather than as **sequential inputs**. The LSTM model processes '`actual_demand`' data using lagged inputs, aligning past values at each current time step (e.g., Index 3). This preprocessing explicitly organizes past observations into a standardized format, ensuring compatibility with the LSTM's internal architecture. At each time step, the model receives the current '`actual_demand`' value along with a fixed-length sequence of past demand values (all those aligned along the vertical at that step), allowing it to effectively learn temporal dependencies and recurring demand patterns, including seasonal trends and holiday effects.

### 3.1.5 Hybrid Model

The strength of a hybrid model is that it combines the advantages of both of its components, which, for GBM and LSTM, are capturing static patterns and trends and learning temporal dependencies in sequential data, respectively. The predictions from both models are then merged using a **weighted average**, where the weights are determined based on the relative importance of each model's predictions.

#### Gradient Boosting Machine (GBM)

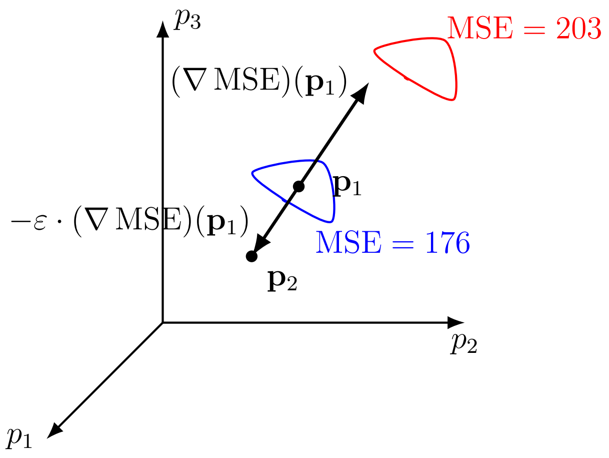

$p_3$
MSE $= 203$
$(\nabla\,\mathrm{MSE})(\mathbf{p}_1)$
$\mathbf{p}_1$
$-\varepsilon\cdot(\nabla\,\mathrm{MSE})(\mathbf{p}_1)$
MSE $= 176$
$\mathbf{p}_2$
$p_2$
$p_1$

Figure 20: Gradient Orthogonal to Level Sets in Increasing Direction

In a **Gradient Boosting Machine** (GBM), the objective is to *boost machine* performance by minimizing this model's Mean Squared Error (MSE). This is achieved through an ITERATIVE PROCESS that adjusts the model's parameters $\mathbf{p}$ in the direction of the negative *gradient* $-(\nabla\,\mathrm{MSE})(\mathbf{p}_1)$:

$$\mathbf{p}_2 = \mathbf{p}_1 - \varepsilon \cdot (\nabla\,\mathrm{MSE})(\mathbf{p}_1),$$

where $\varepsilon$ is the learning rate. The negative multiple $-\varepsilon \cdot (\nabla\mathrm{MSE})(\mathbf{p}_1)$ points in the direction of the steepest decrease of the MSE. Adding it to $\mathbf{p}_1$ yields

$p_2$, the new parameter value, which minimizes the MSE as efficiently as possible. This process is repeated iteratively, updating p step by step, until the parameter tuple p converges to a tuple $p_{min}$, which minimizes the MSE.

## Long Short-Term Memory (LSTM)

**Long Short-Term Memory** (LSTM) models are specialized neural networks designed to capture temporal dependencies in sequential data (e.g., seasonal trends or cause-effect relationships). They handle both short-term patterns (e.g., promotional discounts on cars) and longer-term effects (e.g., evolving emission regulations for cars). Unlike standard recurrent neural networks (RNNs), LSTMs use gating mechanisms to mitigate the vanishing gradient problem, enabling them to retain longer-term dependencies more effectively than traditional RNNs. While AI assistants and chatbots also manage both short-term memory (maintaining coherence within individual responses) and long-term memory (tracking the overall subject of the conversation) to maintain coherent interactions, they typically rely on transformer-based architectures rather than LSTMs, using mechanisms like self-attention and context windows, which efficiently capture both immediate context and conversational history.

## Python Code

The following code is quite interesting and deserves the readers attention.

```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, MinMaxScaler #
    StandardScaler: Z-Score
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import LinearRegression
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Input

```

```
# TensorFlow is a library designed to handle tensors, which are
    generalizations of scalars, vectors, and matrices to higher
    dimensions, and used in machine learning and deep learning; Keras
    is a module, a deep learning API, an interface between the
    developper and deep learning frameworks like TensorFlow; models is
     a submodule, and Sequential is a class for building neural
    networks; LSTM is a class for creating LSTM layers; Dense is a
    fully connected layer to compute activation functions, for
    instance; Input is a placeholder layer for specifying input shapes
     in a model

from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
import tensorflow as tf

# Set reproducibility for TensorFlow
# np.random.seed(42)
# tf.random.set_seed(42)

# I. Data Simulation over 10 Years (120 Months) with a Hybrid Model in mind

months = pd.date_range(start="2014-01-01", periods=120, freq='ME')  #
     'ME' means Month End
# date_range is a function of the library pd

data = pd.DataFrame({"month": months})

data['base_demand'] = 1000 + 200 * np.abs(np.sin(np.pi * data['index'
    ] / 6))
# Base demand with seasonal highs in April and October and lows in
    January and July

# Steady baseline growth - increases linearly from 0 to 360 over 120
    months
data['market_trend'] = np.linspace(0, 360, 120)

# External factors encoded as multiplicative adjustments, simulating
    pandemics, geopolitical issues, etc
# np.where(condition, x, y) returns x where the condition is True and
     y where it is False

```

```python
data['pandemic_factor'] = np.where((data['month'] >= '2020-03-01') & (data['month'] <= '2021-03-01'), 0.6, 1.0)
data['geo_factor'] = np.where((data['month'] >= '2022-02-24') & (data['month'] <= '2023-03-01'), 0.8, 1.0)
data['natural_disaster_factor'] = np.where((data['month'] >= '2019-09-01') & (data['month'] <= '2019-11-01'), 0.7, 1.0)

# Combine to simulate final demand

data['actual_demand'] = (
    data['base_demand'] + data['market_trend']
) * data['pandemic_factor'] * data['geo_factor'] * data['natural_disaster_factor']

# Add noise to simulate real-world data

data['actual_demand'] += np.random.normal(0, 50, size=len(data))

# II. Feature Engineering for Gradient Boosting

# The next code lines correspond to feature engineering for Gradient Boosting (static features); Gradient Boosting is a supervised ML model to predict demand based on static features like base_demand (including seasonality) and market_trend (features, which evolve slowly in comparison with dynamic/sequential features like actual_demand); the column 'moving_avg_3' contains the 'moving/rolling average over the last three months' of the actual_demand, computed using .rolling(window=3).mean(). This smooths the data, highlighting underlying trends by reducing short-term fluctuations. For example, if the demand values are [100, 200, 300, 400], the moving average is [NaN, NaN, 200, 300]. During fine-tuning of the model we changed 3 to 10.

data['moving_avg_3'] = data['actual_demand'].rolling(window=10).mean()
# rolling() is a method of pandas Series/DataFrame that creates a rolling window object, and mean() is a method applied to compute the mean within each window
data.dropna(inplace=True)

# III. Feature Engineering for Long Short-Term Memory Systems

```

```python
lags = 10
for lag in range(1, lags + 1):
    data[f'lag_{lag}'] = data['actual_demand'].shift(lag)
data.dropna(inplace=True)

# For each lag (1 to 10), the actual_demand values are shifted
    forward by that number of steps (1 to 10), creating new columns (
    lag_1, lag_2,...., lag_10): if you have a time series [100, 200,
    300, 400, 500, 600], where 600 is the current value, and apply a
    lag of 1, the result will be lag_1:[NaN,100,200,300,400,500]; the
    core idea is that shifting past values to the current position
    organizes the data in a way that aligns with the LSTM's internal
    structure, enabling it to effectively learn and predict temporal
    dependencies

# IV. Splitting Features into Static and Sequential Predictors

X_static = data[['base_demand', 'market_trend', 'moving_avg_3']]  #
    For GBM
X_seq = data[[f'lag_{i}' for i in range(1, lags + 1)]]  # For LSTM
y = data['actual_demand']
# X_static contains smoothed actual demand (moving_avg_3), while
    X_seq contains the lags of the actual demand; [[...]] selects
    multiple columns from a DataFrame; f-strings allow embedding
    variables directly into strings, e.g., f'lag_{i}'.

# V. Scaling Static and Sequential Features

scaler_static = StandardScaler() # Mean 0 and standard deviation 1
scaler_seq = MinMaxScaler() # Fixed range

# StandardScaler: Optimal for normally distributed static features;
    standardizes data to zero mean and unit variance; MinMaxScaler:
    Ideal for time series data; preserves relative magnitudes and
    trends by scaling to a fixed range.

X_static_scaled = scaler_static.fit_transform(X_static)
X_seq_scaled = scaler_seq.fit_transform(X_seq)
# The choice of scalers aligns with the models' working mechanisms

# VI. Splitting Data into Training and Test Sets

```

```
split_index = int(0.8 * len(data))  # Calculate 80% split index
X_train_static, X_test_static = X_static_scaled[:split_index],
    X_static_scaled[split_index:]
X_train_seq, X_test_seq = X_seq_scaled[:split_index], X_seq_scaled[
    split_index:]
y_train, y_test = y[:split_index], y[split_index:]

# VII. Random Forest Regressor

rf = RandomForestRegressor(n_estimators=100)
rf.fit(X_train_static, y_train)
y_pred_rf = rf.predict(X_test_static)
# models like RF, LR, and GBM don't inherently handle sequential
    dependencies, so they are typically trained on static features
    rather than lagged data.

# VIII. Linear Regressor

lr = LinearRegression()
lr.fit(X_train_static, y_train)
y_pred_lr = lr.predict(X_test_static)

# IX. Gradient Boosting Model

gbm = GradientBoostingRegressor(n_estimators=100)
gbm.fit(X_train_static, y_train)

# X. Long Short-Term Memory Model

X_train_seq = X_train_seq.reshape((X_train_seq.shape[0], X_train_seq.
    shape[1], 1))

# The new X_train_seq is a 3D matrix or tensor, structured in the
    format required by the LSTM model; its rows (index [0] in the
    array) represent the observations/months/time steps, its columns (
    index [1] in the array) correspond to the ten lagged values, and
    its depth contains the features used for prediction -- in this
    case, only actual_demand; since there is only one feature, the
    tensor contains only one slice and actually remains 2D; this setup
     ensures that the model receives past actual_demand values as
    input while not explicitly including the current actual_demand as
    an additional input
```

```

X_test_seq = X_test_seq.reshape((X_test_seq.shape[0], X_test_seq.
    shape[1], 1))

# Define LSTM model;

lstm = Sequential([
    Input(shape=(X_train_seq.shape[1], 1)),
    LSTM(100, activation='relu'),
    Dense(1)
])
lstm.compile(optimizer='adam', loss='mse')
lstm.fit(X_train_seq, y_train, epochs=50, batch_size=8, verbose=0)

# Input(shape=(X_train_seq.shape[1], 1)) selects the lagged demand
    from the sequential training data; LSTM(100): An LSTM layer with
    100 hidden neurons to learn sequential patterns (see below: NNs);
    activation='relu' sets the neuron activation function to be the `
    Rectified Linear Unit' activation function (see below: NNs); Dense
    (1): `Dense' refers to a fully connected output layer (i.e., a
    layer where each neuron receives input from every neuron in the
    previous layer), and `(1)' indicates that the output layer
    consists of a single neuron, predicting a single value -- the
    actual demand for unseen input data; compilation configures the
    model by specifying that optimizer='adam' (the Adam optimizer
    adapts the learning rate (the factor `epsilon' in Subsection
    Gradient Boosting Machine) for faster convergence), that the loss
    function (the function that the model aims to minimize during
    training to improve performance) is loss='mse' (Mean Squared Error
    ); in training, epochs=50 means the model iterates over the
    dataset 50 times, with each epoch processing the full training set
     in batches; each batch updates the model parameters using
    gradient descent, while convergence is typically achieved only
    after multiple epochs; batch_size=8 means that training uses mini-
    batches of 8 samples at a time; verbose=0 suppresses output logs
    during training for cleaner display

# XI. Predictions from the Hybrid Model

y_pred_gbm = gbm.predict(X_test_static)

@tf.function
```

```python
def predict_lstm(input_data):
    return lstm(input_data)
y_pred_lstm = predict_lstm(X_test_seq).numpy().flatten()

# @tf.function enables TensorFlow's main strength -- efficient
    computation graph execution -- which optimizes performance and
    allows parallel processing on Graphics Processing Units (GPUs) and
     Tensor Processing Units (TPUs) for deep learning; .numpy()
    converts the TensorFlow tensor output into a 2D NumPy array of
    shape (samples, 1), while .flatten() reshapes it into a 1D array
    of shape (samples), simplifying the output for comparison with the
     true target values during evaluation

# Combine predictions (weighted average)

alpha = 0.5  # Weight for GBM
beta = 0.5   # Weight for LSTM
y_pred_hybrid = alpha * y_pred_gbm + beta * y_pred_lstm

# XII. Evaluation of the Hybrid Model

rmse_hybrid = np.sqrt(mean_squared_error(y_test, y_pred_hybrid))
print(f"Hybrid Model RMSE: {rmse_hybrid}")
rmse_rf = np.sqrt(mean_squared_error(y_test, y_pred_rf))
print(f"Random Forest RMSE: {rmse_rf}")
rmse_lr = np.sqrt(mean_squared_error(y_test, y_pred_lr))
print(f"Linear Regression RMSE: {rmse_lr}")

# XIII. Visualization

plt.figure(figsize=(12, 6))

# Plot actual demand
plt.plot(data['month'], data['actual_demand'], label="Actual Demand")

# Hybrid model plot
plt.plot(data['month'].iloc[-len(y_test):], y_pred_hybrid, label="
    Hybrid Model Forecast", linestyle='--')

# Random Forest plot
plt.plot(data['month'].iloc[-len(y_test):], y_pred_rf, label="Random
    Forest Forecast", linestyle=':')
```

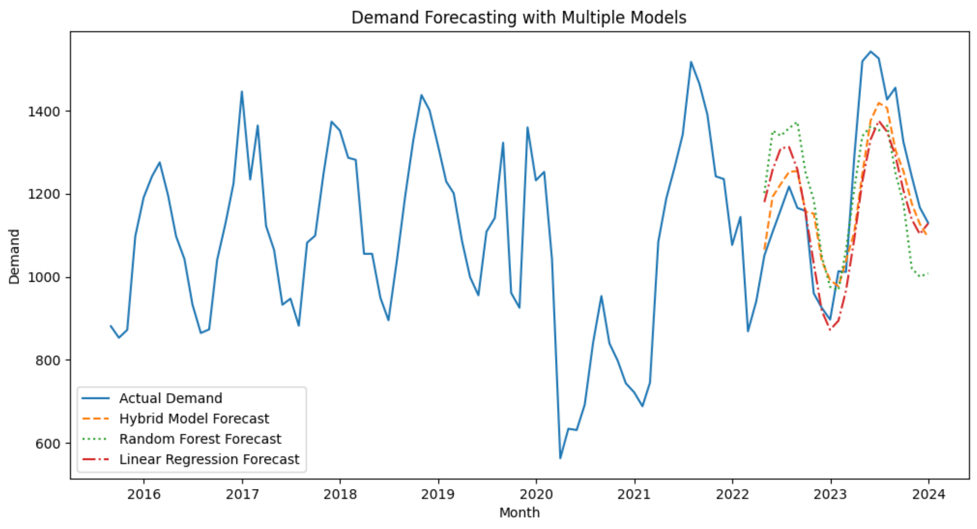
```python

# Linear Regressor plot
plt.plot(data['month'].iloc[-len(y_test):], y_pred_lr, label="Linear
    Regression Forecast", linestyle='-.')

# Add legend and labels
plt.legend()
plt.title("Demand Forecasting with Multiple Models")
plt.xlabel("Month")
plt.ylabel("Demand")
plt.savefig("C:/Users/norbert.poncin/Downloads/DemForMulMod.png")
plt.show()
```

### Code Output

```
Hybrid Model RMSE: 111.43207213590364
Random Forest RMSE: 157.5859239456133
Linear Regression RMSE: 129.3843798974777
```



### 3.1.6 Neural Networks

Neural networks are composed of **layers of neurons** – the **input layer** (in blue in Figure 21), **hidden layers** (in orange – only one in Figure 21), and the **output layer** (in green) – connected by **weighted edges**. These weights represent the strength of the connections and are adjusted during training to minimize error. The nodes $b_h$ (pink) and $b_o$ (red) and the weights $w_h$ and $w_o$ represent biases. Neural networks can learn complex patterns and relationships, making them powerful tools for tasks like classification, regression, and pattern recognition.

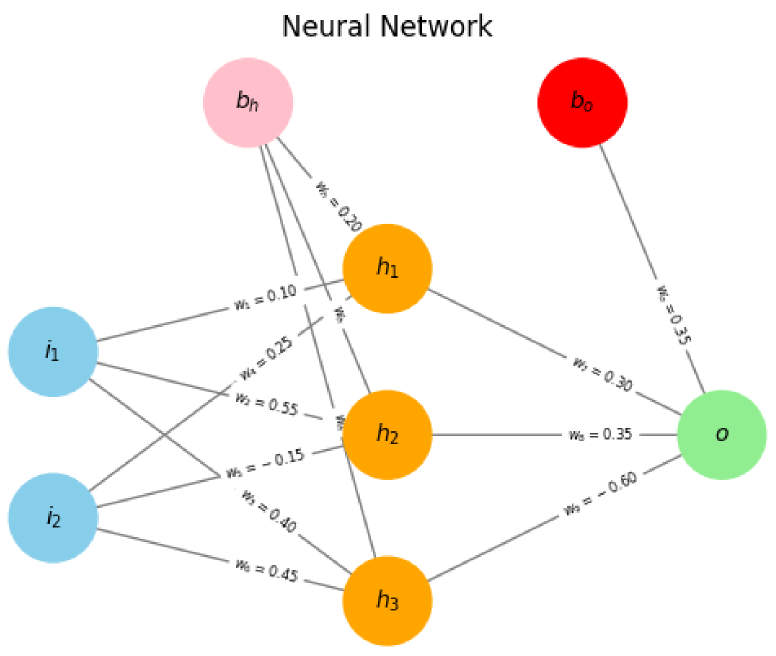


Figure 21: Visualization of a feedforward neural network (FNN)

More precisely, in a neural network, each node computes the weighted sum of inputs from the previous layer, then adds a **bias** term, which shifts the sum before applying an **activation function** (e.g., Logistic or Rectified Linear Unit). The activation function transforms this adjusted sum into an output signal, determining the information passed to the next layer, similar to biological neurons.

### 3.1.7 Examples

**Example 1**

For example, consider $h_1$ in the hidden layer with inputs $i_1 = 0.60$ and $i_2 = 0.80$, weights $w_1 = 0.10$ and $w_4 = 0.25$, and bias $b_h = 1 \cdot w_h = 0.20$. The shifted, weighted input is:

$$z_{h_1} = w_1 i_1 + w_4 i_2 + b_h = 0.10 \cdot 0.60 + 0.25 \cdot 0.80 + 0.20 = 0.46 \; .$$

Using the logistic function $f(z) = 1/(1 + e^{-z})$ as activation function, the output of $h_1$ is

$$f(z_{h_1}) = \frac{1}{1 + e^{-0.46}} \approx 0.613 \; .$$

This process works similarly for $h_2$ and $h_3$. The outputs from the hidden layer become inputs to the output neuron $o$, which applies the same steps with its own weights, inputs, and bias $b_o = 1 \cdot 0.35$.

**Example 2**

Consider a neuron in a neural network tasked with determining whether to send a signal forward, indicating the presence of a feature (e.g., detecting an edge of a simplified cat contour in an image). The neuron receives inputs from the previous layer, which are weighted sums of signals (e.g., features like brightness or color intensity from pixels).

The **Rectified Linear Unit** (ReLU) activation function transforms the weighted sum $z$ by outputting $\mathrm{ReLU}(z) = \max(0, z)$. If the weighted sum is positive, the neuron 'fires', passing the positive value forward. Otherwise, the neuron outputs 0, effectively not firing.

This mechanism allows the network to learn complex patterns. Different neurons can fire under different conditions, enabling the network to detect a variety of features. The nonlinear nature of activation functions like ReLU enables the network to model intricate relationships in data, beyond simple linear correlations.

## 3.2 Automated Inventory Management

*The subsequent phases of the Automated Supply Chain Management Project are presented as research activities (Subsections 3.2, 3.3, and 3.4).*

Write Python code to monitor inventory levels in real-time by processing ERP data or tracking simulated inventory updates via a file system. The program should analyze stock fluctuations and trigger a red alert when inventory drops below predefined thresholds. Upon detecting a critical stock-out risk, the system should automatically reorder the product by sending an email to the seller or using APIs to ensure timely restocking and prevent disruptions.

For this task, we refer the reader to the following sections:

- Real-time Monitoring and GDPR Compliance in Manufacturing
- Automated File System Watcher and Data Preprocessing Pipeline
- Automated Correlation Testing Every 100 Emails
- Automated Targeted Marketing Campaigns

Note that implementing a file system watcher is simpler compared to real-time monitoring.

## 3.3 Route Optimization

We outline the following steps as a **simplified research framework** for optimizing delivery routes. The approach provides a structured foundation but can be adjusted or expanded based on specific requirements and constraints.

- **Data Collection**: Gather real-time traffic, weather, and vehicle GPS data.
  - *Real-time Traffic*: Use, for instance, the Google Maps API or OpenStreetMap APIs to fetch traffic data (please check the official documentation of the Google Maps API and the specific OpenStreet Map-based services for pricing and free tiers).
  - *Weather Data*: Access APIs like OpenWeatherMap for weather updates (check the official documentation for pricing and free tiers).

# 4 Learning Outcomes

After carefully working through this chapter, the reader should be able to:

- Classify applications into categories of **Artificial Intelligence (AI)**, **Machine Learning (ML)**, **Neural Networks (NNs)**, and **Deep Learning (DL)**, and distinguish between **Supervised** and **Unsupervised ML**.
- Explain the functioning principles of **Neural Networks (NNs)**, including key concepts such as **bias terms, activation functions**, and the underlying **training mechanisms**.
- Model data in a **real-world-like manner** by using statistical distributions such as the **normal, truncated normal, log-normal, Student's t, Poisson**, and **chi-square** distributions, while encoding realistic and complex patterns.
- Explain and apply **dimensionality reduction** techniques such as **Principal Component Analysis (PCA)** and **t-Distributed Stochastic Neighbor Embedding (t-SNE)**.
- Solve clustering problems, including **Customer Segmentation, High-Profile Customer Detection**, and **Tailored Marketing Strategy Development**.
- Identify suitable **Machine Learning models for clustering**, such as **K-Means, K-Means++, Gaussian Mixture Models (GMMs)**, and **Density-Based Spatial Clustering of Applications with Noise (DBSCAN)**, and apply techniques like the **Elbow Method** for determining the optimal number of clusters.
- Explain and apply **Ensemble Methods** such as **Gradient Boosting Machines (GBMs)** and **Neural Networks (NNs)** like **Long Short-Term Memory (LSTM)**.
- Differentiate between **static and sequential features** and apply time-series techniques such as **Moving Window Averages** and **Lagging**.

- Write code to create **compelling static and interactive 2D and 3D visualizations** of data.
- Utilize technologies such as **SMTP, MIME**, and **HTML** for **email communication, formatting**, and **marketing strategy implementation**.
- Independently explore and implement a **professional approach** to solving **real-world problems in the manufacturing industry**.

YOUR SCIENCE

CREATING OPPORTUNITY

## About the Author

Norbert Poncin is a Luxembourgish mathematician, who was originally educated as a mathematical analyst and has worked extensively in partial differential equations (PDEs) at the University of Liège. His Master's thesis focused on the propagation of singularities in boundary value problems (BVPs) for dynamic hyperbolic systems. Applying the finite element method (FEM), his subsequent dissertation addressed BVPs for complex elliptic systems of PDEs. For his doctoral thesis, he explored mathematical quantization, while his post-doctoral education at the Polish Academy of Sciences strongly emphasized theoretical physics and its models.

Norbert has served as a Full Professor of Mathematics at the University of Luxembourg for more than 25 years and collaborated with more than 25 foreign professors and post-doctoral scholars. He has organized numerous academic events, notably approximately 10 international research meetings and over 20 research seminars focusing on theories, frameworks, concepts and models in Physics and Engineering. Beyond a substantial publication record in Differential Geometry, Algebraic Topology, and related disciplines, he has contributed roughly 25 papers to the fields of Mathematical Physics and Quantum Theory.

He was the leading instructor for over 20 university courses. Spanning a diverse spectrum of subjects, including mathematical analysis, probability theory, inferential statistics, point and solid dynamics, Lagrangian and Hamiltonian mechanics, mechanics of deformable solids, fluid dynamics, special relativity, quantum physics, geometric methods in mathematical physics, and supersymmetric models, his teaching portfolio underscores his extensive experience in applied aspects of mathematics.

In 2023, Norbert Poncin founded the mathematical consulting agency Your Science, where he currently serves as director. His primary interests include data science and artificial intelligence, along with mathematical modeling and computational science.