PROJECT: DEMAND FORECASTING

Abstract

This project endeavors to address the complex challenge of demand forecasting in the retail industry, leveraging a vast dataset from one of the world's largest retail chains. Over a three-year period, historical sales data, pricing information, and promotional details for a diverse product portfolio across 76 stores were analyzed on a week-to-week basis. The objective was to construct an efficient forecasting model capable of predicting sales for each unique product-store combination over the next 12 weeks.

The dataset, encompassing various features such as total price, base price, and promotional indicators, was subjected to thorough preprocessing and feature engineering. A detailed exploration of the data was facilitated through visualizations, revealing temporal patterns and highlighting the impact of promotional activities on sales. Time series methodologies, including the conversion of temporal data and the incorporation of lag features, were employed to capture inherent patterns. Some distinct machine learning models – Random Forest Regression, Support Vector Machine, and LightGBM – were implemented and fine-tuned to achieve optimal predictive accuracy.

The results of the models were evaluated using Mean Squared Error and Mean Squared Logarithmic Error metrics, providing insight into their performance. Additionally, a grid search approach was employed for hyperparameter tuning, optimizing models. The best-performing model was selected based on the achieved accuracy, and the implications of the findings were discussed in the context of the retail forecasting domain. Visualization played a crucial role in model interpretation, showcasing feature importance, temporal trends, and the comparative performance of different algorithms.

This project not only contributes to the advancement of predictive modeling for demand forecasting but also underscores the importance of thoughtful feature engineering, model selection, and hyperparameter tuning in optimizing model performance for time series data.

Problem Statement

Demand forecasting for a large retail chain, spanning 76 stores and a diverse product portfolio, using three years of historical sales data. The objective is to predict weekly sales for each unique product-store combination over the next 12 weeks.

Methodology

The project involves thorough preprocessing and feature engineering of the dataset, including the exploration of temporal patterns and the impact of promotional activities through visualizations. Time series methodologies, such as lag features, are employed. Three machine learning models – Random Forest Regression, Support Vector Machine, Decision Tree, KNeigbhours and LightGBM, – are implemented and fine-tuned for optimal predictive accuracy.

Key Findings

1. Visualizations reveal temporal patterns and highlight the influence of promotions on sales.
2. The Random Forest, SVM, and LightGBM models provide varying levels of predictive accuracy.
3. Hyperparameter tuning using grid search optimizes the RandomForestRegressor and LightGBM models.

Conclusions

The project contributes to advancing predictive modeling for demand forecasting in retail. It emphasizes the importance of thoughtful feature engineering, model selection, and hyperparameter tuning. The best-performing model is selected based on accuracy, and the findings are discussed in the context of retail forecasting.

Introduction

Consider running a huge retail chain with 76 locations and a wide range of products. One of the most difficult tasks is determining how much of each product to keep in stock each week. This is the point at which precise demand forecasting becomes useful. Simply said, it's estimating how much of each item buyers will purchase.

For our project, we're diving into three years' worth of sales data to build a smart system that predicts sales for each product in each store. Why is this important? Well, it helps the retail chain avoid running out of popular items and wasting money on excess stock. It's like having a crystal ball to make sure they always have just the right amount of everything.

Our goal is not simply to make predictions, but to do it very effectively. We're employing cutting-edge machine learning methods to build a system that learns from past sales patterns and can adapt to shifting trends. In addition, we're investigating how promotions affect sales because, let's face it, who doesn't appreciate a good deal?

In a nutshell, we want to help the retail chain make better judgements about what to stock in each location. This means less guesswork and more data-driven strategic planning. Our quest include investigating time patterns, comprehending the impact of promotions, and selecting the finest instruments for the work. Stay tuned to learn how this effort not only benefits this retail behemoth but also lays the groundwork for improved retail operations everywhere.

Data Description

The backbone of this project lies in a comprehensive dataset spanning three years of sales history for a vast retail chain with 76 stores. Let's delve into the key characteristics of this dataset:

1. Source
   The dataset originates from the internal records of one of the world's largest retail chains. It encapsulates weekly sales data for a diverse product portfolio across its 76 stores.

2. Size
   Encompassing millions of records, the dataset provides an extensive temporal snapshot of sales activities. This large-scale dataset is crucial for capturing nuanced patterns across various product-store combinations.

3. Features
   The dataset is enriched with a range of features crucial for demand forecasting:

VARIABLE	DEFINITION
record_ID	A special code that uniquely identifies each combination of a week, a store, and a product.
week	The starting date of the week for which the data is recorded.
store_id	A unique code for each store; each store has its own identifier.
sku_id	A unique code for each product; each product has its own identifier.
total_price	The price at which the product was sold during that week.
base_price	The initial or standard price of the product.
is_featured_sku	Indicates whether the product was part of a featured promotion for that week.
is_display_sku	Indicates whether the product was prominently displayed within the store.
units_sold	The number of units (products) sold during that week for the specific combination of store, product, and week.

4. Preprocessing Steps

- Datetime Conversion
  The `week` column, initially in object format, is converted into datetime format to facilitate temporal analysis.
- Feature Engineering
  Additional features such as day of the week, month, and year are derived from the `week` column to capture temporal trends effectively.
- Exploratory Data Analysis (EDA)
  Exploratory visualizations are employed to uncover temporal patterns, relationships between features, and the impact of promotions on sales.
- Data Cleaning
  Address any missing or inconsistent values to ensure the dataset's integrity.

SCOPE 01: SALES PREDICTION MODEL

The primary goal of this scope is to develop a predictive model that can forecast the sales for each SKU (Stock Keeping Unit) in each store for the upcoming weeks. The prediction is based on historical sales data spanning the past three years, recorded on a week-on-week basis. The model should consider not only the historical sales figures but also utilize promotional information associated with each product and store.

SCOPE 02: PRODUCT CLUSTERING AND VISUALIZATION

This scope focuses on grouping products based on their sales patterns and creating visualizations to represent these clusters effectively.

DATA EXPLORATION

1.LOADING AND EXPLORING DATASET

```
import pandas as pd
df = pd.read_csv('train_0irEZ2H.csv') # load the data
```

The provided code prints the shape of the dataset using the `shape` attribute of the pandas Data Frame (`df`). This gives a quick overview of the number of rows and columns in the dataset. Understanding the dataset's shape is an essential initial step in the data exploration process.

2.DISPLAYING DATASET SHAPE

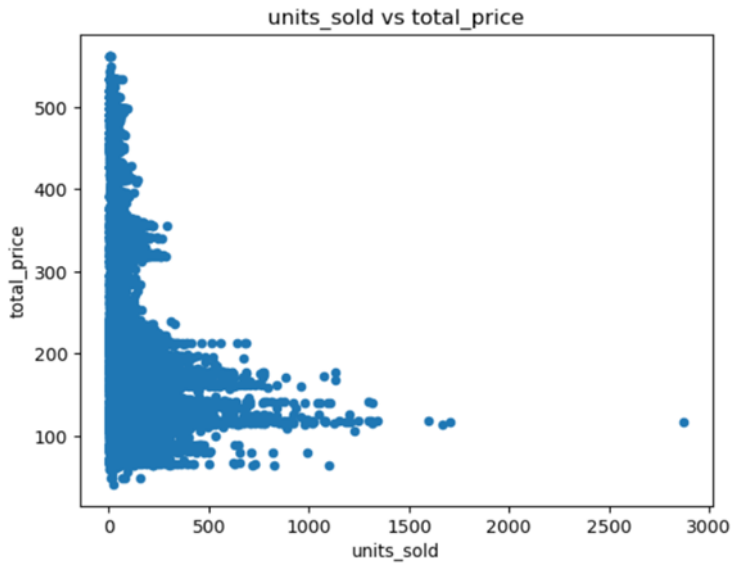
```
print("Shape of the Dataset : \n") # shape
df.shape
```

```
Shape of the Dataset :

(150150, 9)
```

In this section, the code prints the shape of the dataset, providing a concise overview of the number of rows and columns. This step is crucial for understanding the basic structure of the dataset before delving into further analysis.

PRODUCT VISUALIZATION

1.SCATTER PLOT: UNITS SOLD VS. TOTAL PRICE



This code utilizes Plotly Express to generate a scatter plot, visualizing the relationship between "units_sold" and "total_price" from the DataFrame (df). The size of each data point in the plot is determined by the "units_sold" variable.

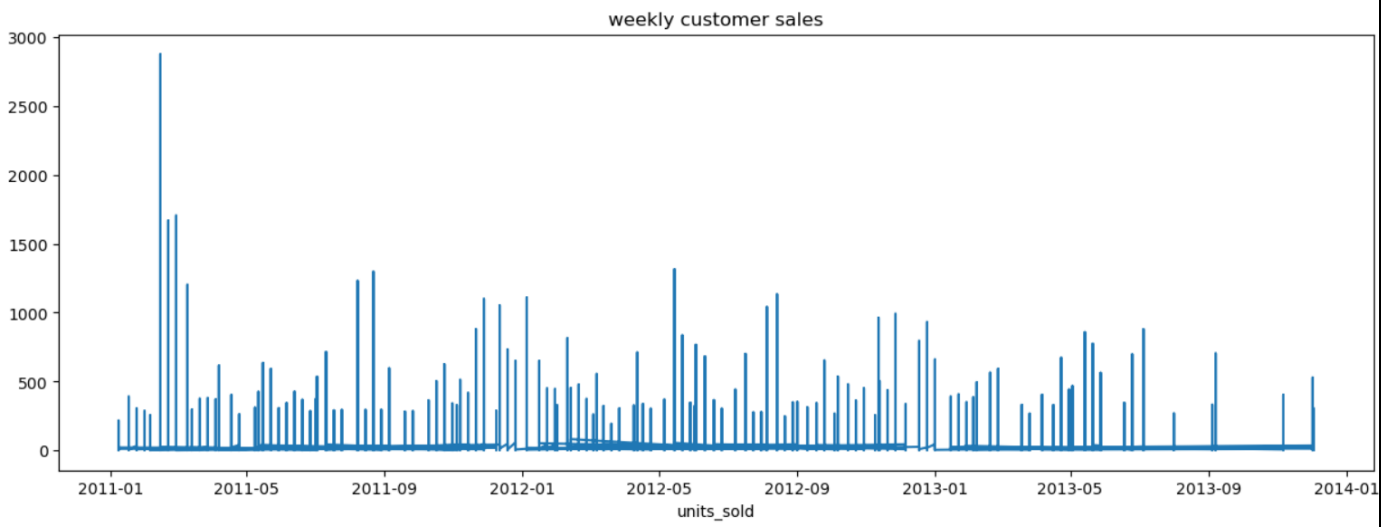
2.SCATTER PLOT: UNITS SOLD VS. BASE PRICE

This code generates a scatter plot using the `plot` function from pandas, showcasing the relationship between "units_sold" and "base_price" from the DataFrame (`df`). The plot provides a visual representation of how the base price impacts the number of units sold. The title of the plot is set as "units_sold vs base_price."

OBSERVATION

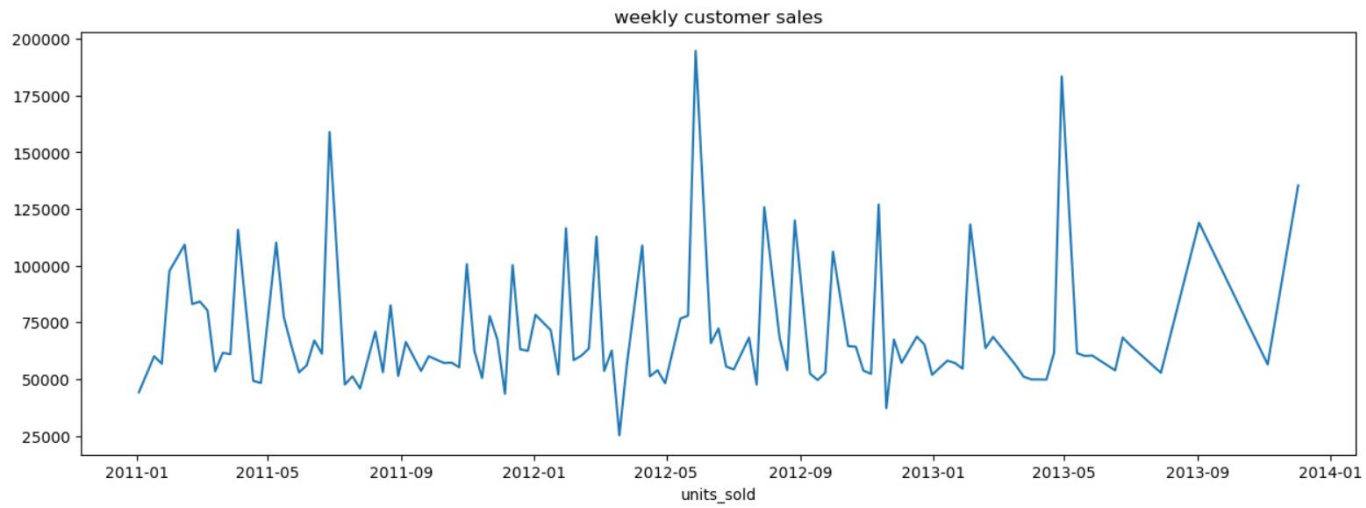
Both of these scatter plots illustrate a notable trend where products with considerably lower prices tend to have higher sales. The majority of sales fall within the price range of 50 to 250. This observation suggests that there may be a price sensitivity among customers, with lower-priced products being more attractive or accessible, potentially driving higher sales volumes.

3.WEEKLY CUSTOMER SALES TREND



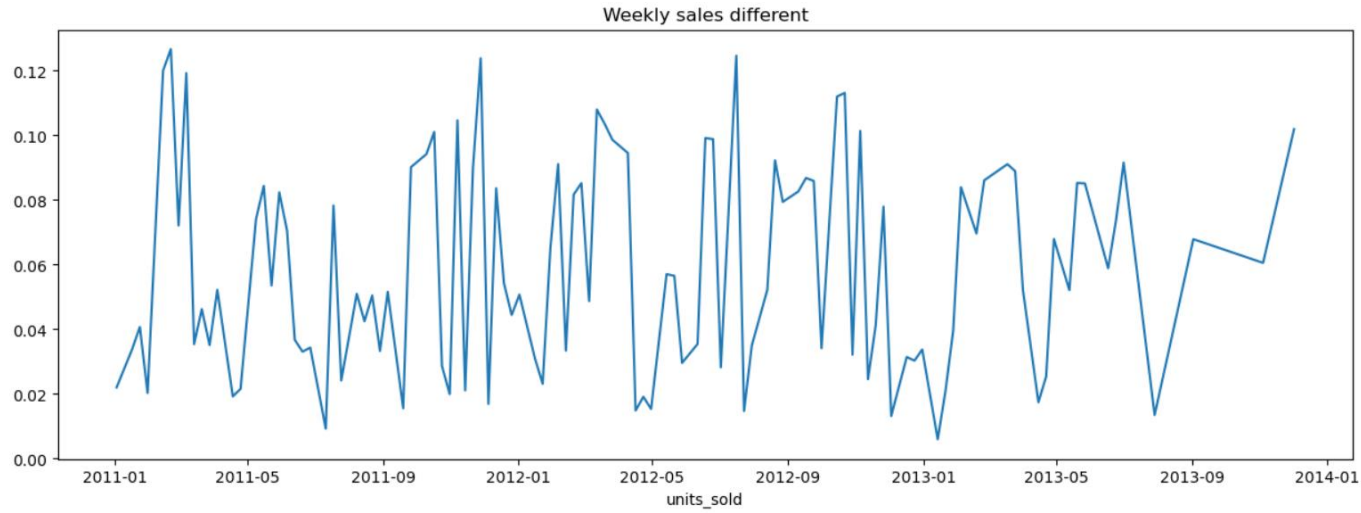
This code utilizes Matplotlib to plot the trend of weekly customer sales. The x-axis represents the weeks, while the y-axis represents the corresponding units sold. The plotted data reveals a discernible trend in customer sales over time. Initially, in the earlier years around 2011, sales were notably high. However, a gradual decline is observed in subsequent years, particularly in 2012 and 2013. The trend continues to show a significant decrease, reaching its lowest point around the beginning of 2014. This observation suggests a declining sales pattern over the analyzed time period. Further investigation into potential factors contributing to this decline may be necessary for a comprehensive understanding of the sales dynamics.

4.WEEKLY CUSTOMER SALES TREND (AGGREGATED)



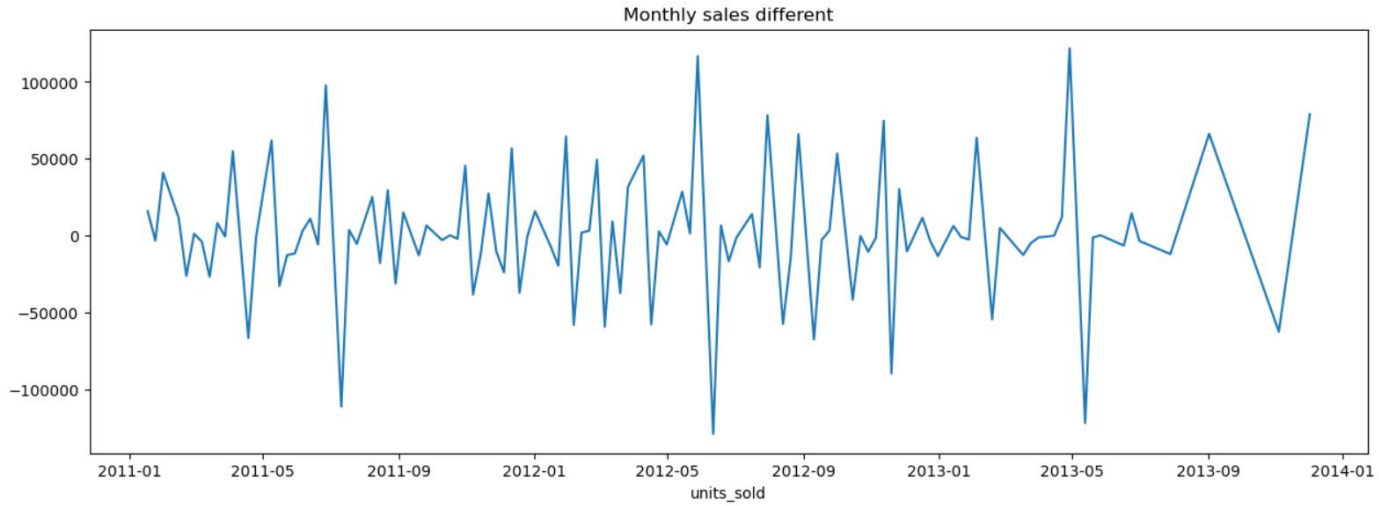
This code uses Matplotlib to plot the trend of weekly customer sales after aggregating the data. The x-axis represents the weeks in timestamp format, and the y-axis represents the corresponding aggregated units sold. The plotted data reveals a recurring pattern, with noticeable spikes in sales occurring consistently around the months of June to July each year. This observation suggests a seasonal trend where sales experience a significant increase during this period. Understanding and analyzing the factors contributing to these spikes can be valuable for strategic planning, inventory management, and promotional activities during these specific months.

5.WEEKLY SALES DIFFERENCES ANALYSIS



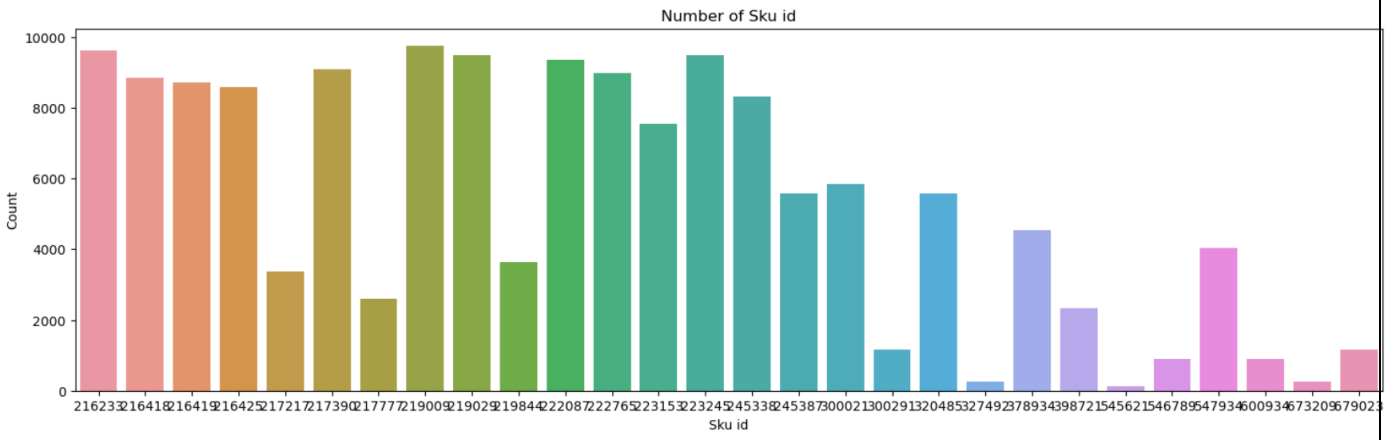
This code utilizes Matplotlib to plot the differences in weekly sales ('diff') over time. The x-axis represents the weeks in timestamp format, and the y-axis represents the corresponding differences in units sold. The plotted graph displays the differences in weekly sales over time, aiming to capture variations or changes in sales patterns. However, it appears that discernible patterns are not readily apparent in this representation. Analyzing fluctuations in this way may require further investigation or alternative methods to identify underlying trends or anomalies in the data. Additional exploratory data analysis techniques or statistical measures could be considered for a more in-depth understanding of the sales dynamics.

6.Monthly Sales Differences Analysis



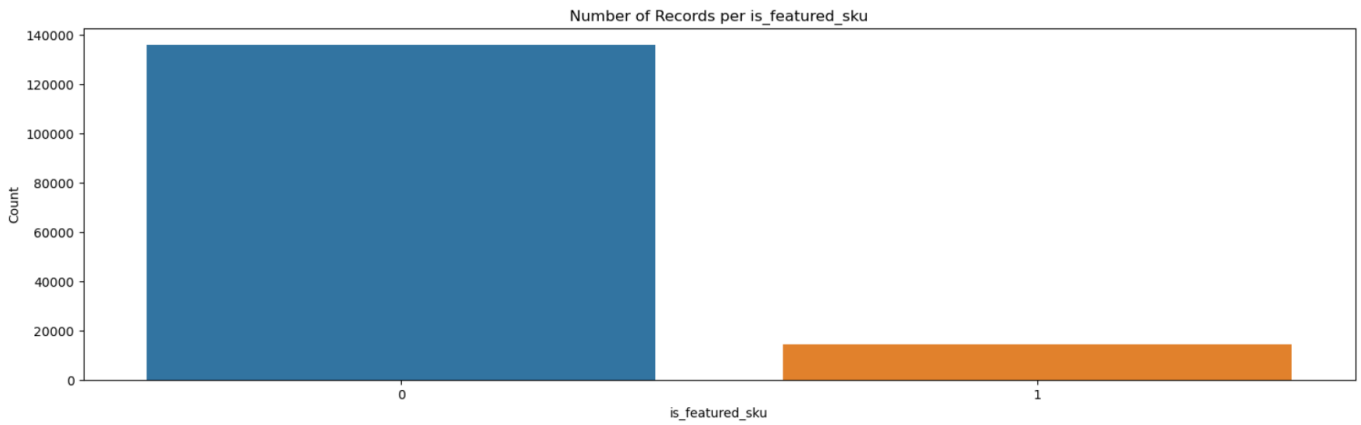
This code utilizes Matplotlib to plot the differences in monthly sales ('sales_diff') over time. The x-axis represents the weeks in timestamp format, and the y-axis represents the corresponding differences in units sold. Notably, there are spikes in sales differences observed around the months of June to July each year. This pattern indicates fluctuations from high to low sales during this period. Understanding the reasons behind these monthly variations can be crucial for effective planning, marketing strategies, and optimizing inventory management during these specific months.

7.SKU Distribution Analysis



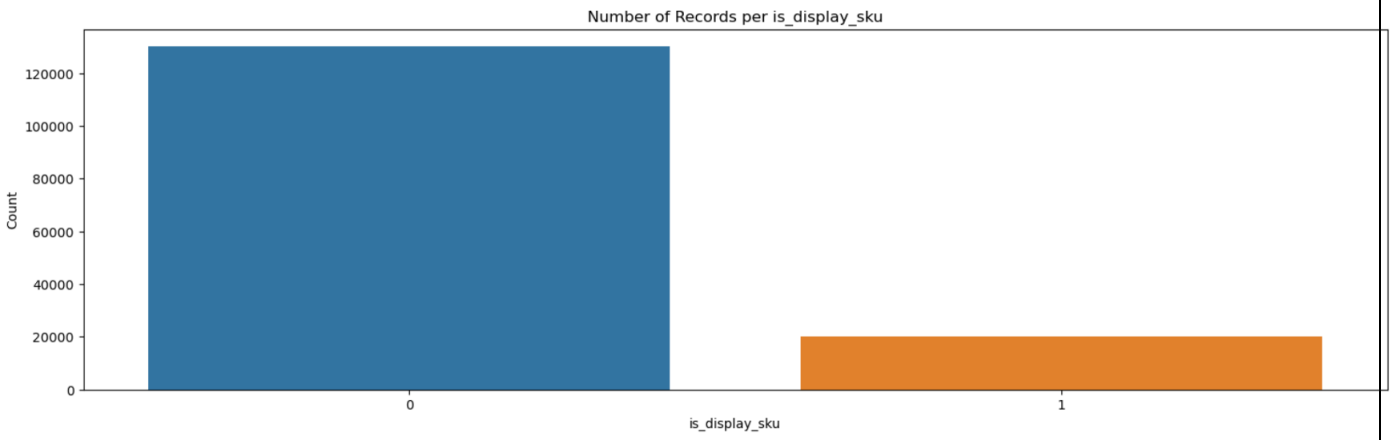
This code uses Seaborn to create a barplot, providing insights into the distribution of SKU IDs in the dataset. The x-axis represents the unique SKU IDs, and the y-axis represents the count of occurrences. It's offering insights into the popularity or sales frequency of each product. It becomes evident that some products are selling at a high frequency, while others have lower sales. This analysis aids in identifying top-performing products and those with lower sales volumes, providing valuable information for inventory management, marketing strategies, and decision-making related to product offerings.

8.Featured SKU Distribution Analysis



This code uses Seaborn to create a barplot, providing insights into the distribution of featured and non-featured SKUs in the dataset. The x-axis represents the 'is_featured_sku' variable, and the y-axis represents the count of occurrences. Resulting plot aims to visualize the frequency of featured and non-featured SKUs, offering insights into the distribution of promotional products within the dataset.

9.Displayed SKU Distribution Analysis

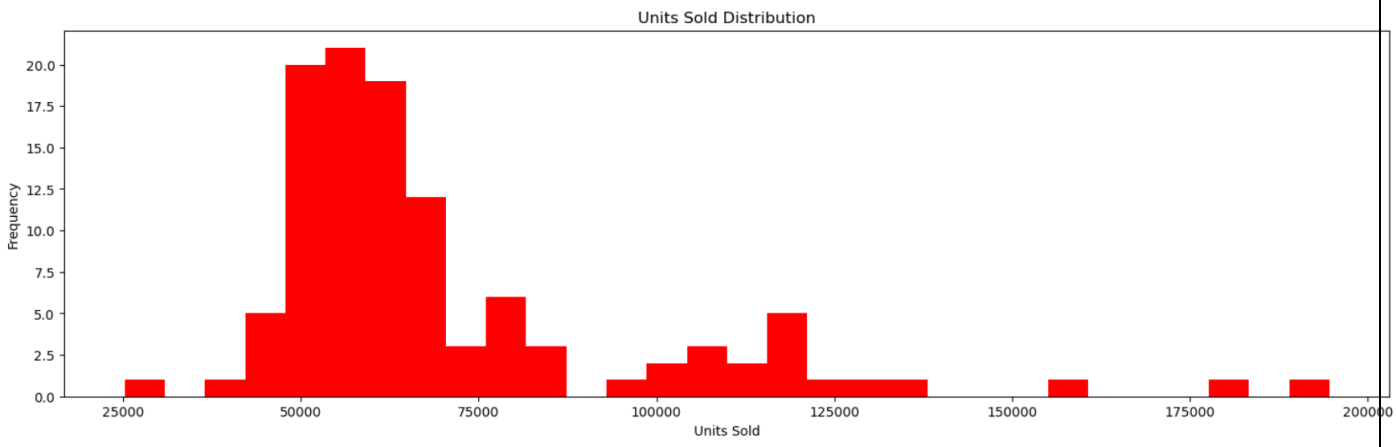


This code utilizes Seaborn to generate a barplot, providing insights into the distribution of displayed and non-displayed SKUs in the dataset. The x-axis represents the 'is_display_sku' variable, and the y-axis represents the count of occurrences. The resulting plot aims to visualize the frequency of products that are prominently displayed and those that are not, offering insights into the distribution of displayed SKUs within the dataset.

Observation

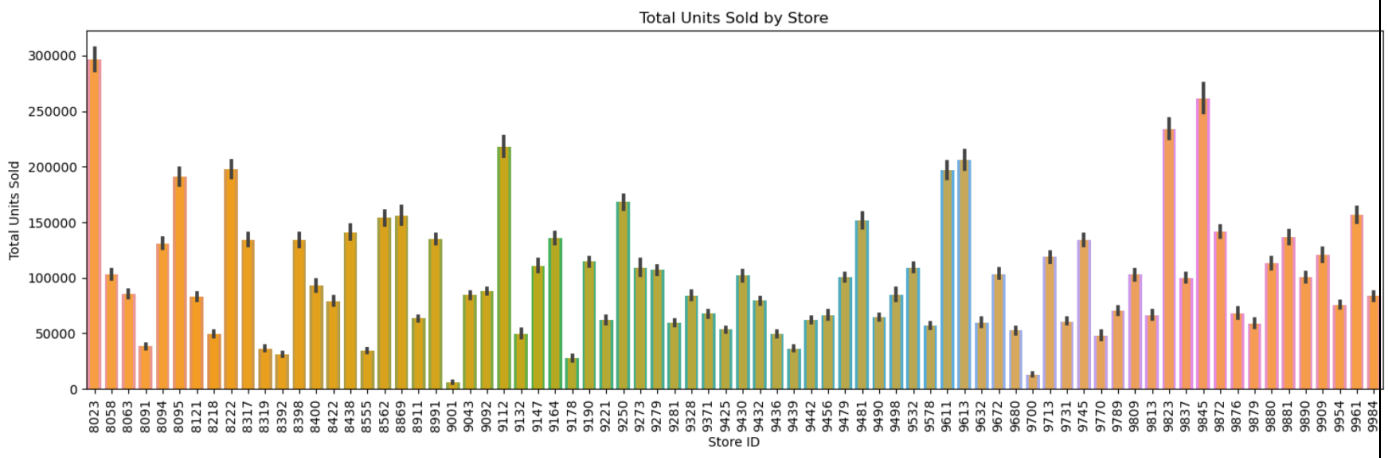
The generated barplots for both `is_featured_sku` and `is_display_sku` reveal insights into the distribution of binary features within the dataset. Both features exhibit a binary structure with values of 0 and 1. The analysis suggests that the dataset contains a balanced distribution of featured and non-featured SKUs, as well as displayed and non-displayed SKUs. Understanding the balance of these features is crucial for interpreting the impact of promotions and product displays on sales and can inform strategies for promoting certain products within the retail chain.

10.Distribution of Weekly Units Sold



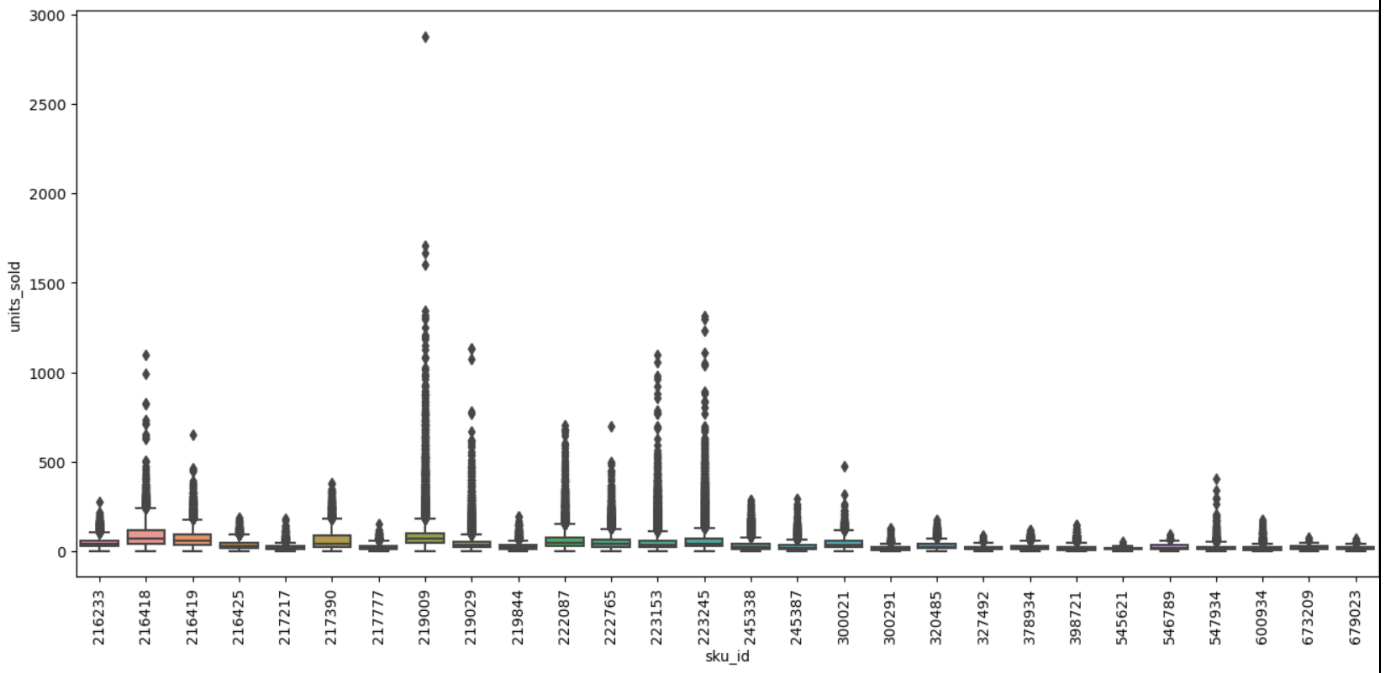
This code utilizes Matplotlib to create a histogram, providing insights into the distribution of weekly units sold. The x-axis represents the range of units sold, and the y-axis represents the frequency of occurrences within each bin. The color is set to red, and the number of bins is specified as 30 for better visualization. The resulting histogram aims to visualize the spread of weekly units sold, offering an overview of the distribution pattern in the dataset.

11.Total Units Sold by Store



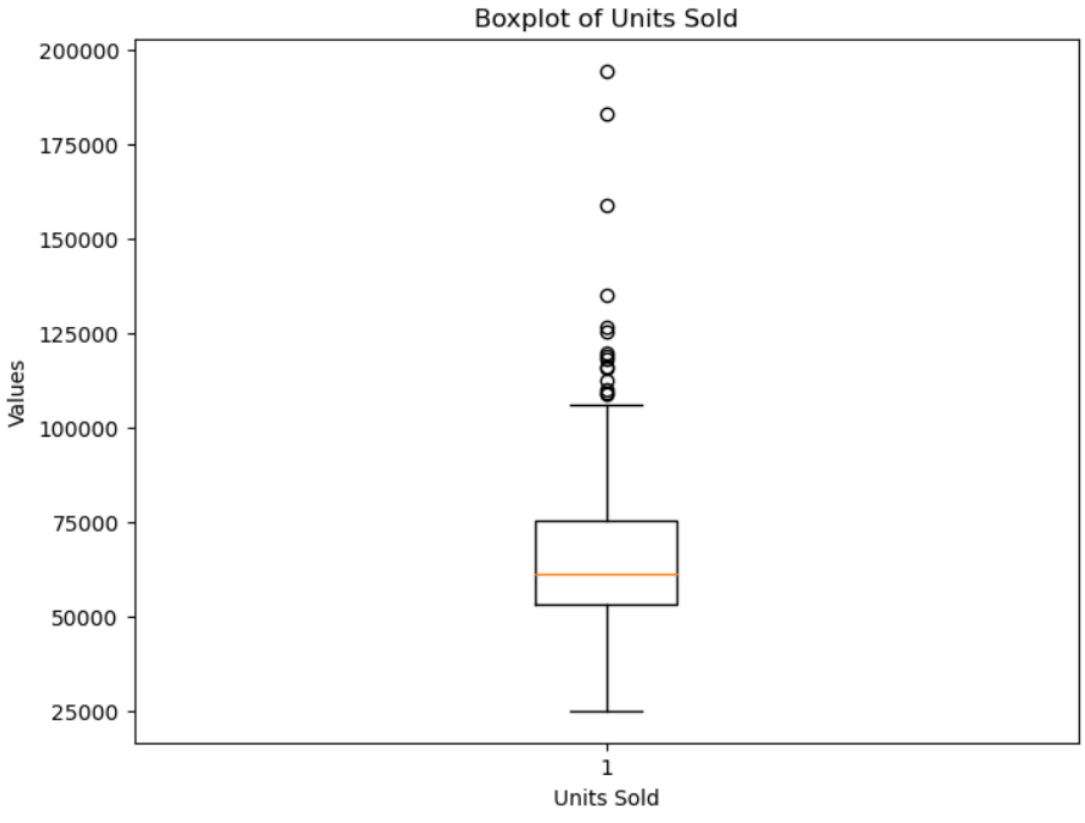
This code utilizes Seaborn and Matplotlib to create a barplot visualizing the total units sold by each store. Two visualizations are presented: a Seaborn barplot showing the sum of units sold for each store and a grouped bar plot generated using Matplotlib. The x-axis represents the store IDs, and the y-axis represents the total units sold. The presented visualizations highlight the store-wise distribution of total units sold. Stores with IDs 8023, 9112, 9823, and 9845 exhibit significantly higher sales, while stores with IDs 9001 and 9700 show comparatively lower total units sold. These insights provide a valuable understanding of the variation in sales performance among different stores, facilitating strategic decision-making for inventory management, marketing, and resource allocation.

12.Boxplot of Units Sold by SKU



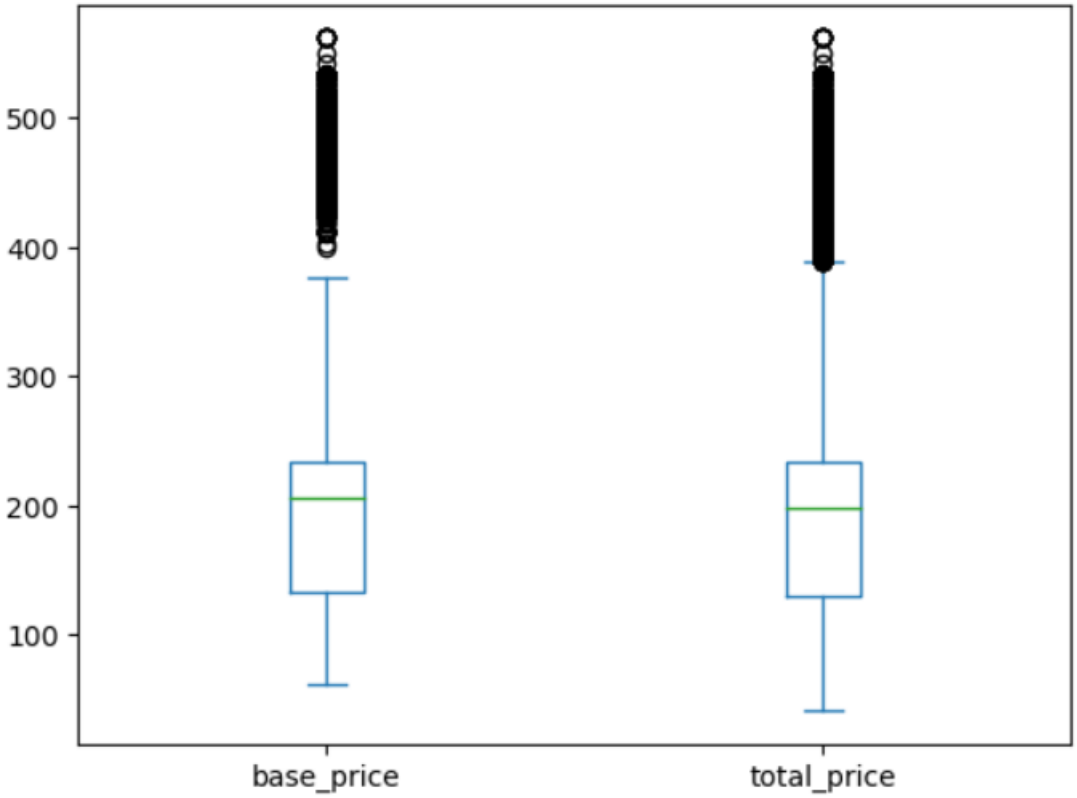
This code utilizes Seaborn to generate a boxplot, visualizing the distribution of units sold for each SKU in the 'train' dataset. The x-axis represents the unique SKU IDs, and the y-axis represents the corresponding units sold. This boxplot provides insights into the variability and distribution of units sold across different products (SKUs), helping to identify sales patterns and potential outliers for individual products.

13.Boxplot of Weekly Units Sold



This code utilizes Matplotlib to create a boxplot, offering insights into the distribution of weekly units sold. The boxplot visualizes the median, quartiles, and potential outliers in the dataset. The x-axis represents the units sold, while the y-axis represents the corresponding values. The generated boxplot indicates that the central tendency of weekly units sold is concentrated around 6000, with a significant number of data points falling within this range. Additionally, values above 10,000 are identified as outliers, suggesting instances where weekly sales significantly deviate from the typical pattern. This visualization provides a clear overview of the distribution, central tendency, and presence of outliers in the weekly units sold data.

14.Boxplot of Base Price and Total Price



This code utilizes Pandas to create a boxplot for the 'base_price' and 'total_price' columns in the 'train' DataFrame. The resulting boxplot visually represents the distribution, central tendency, and potential outliers for both the base and total prices. The boxplot analysis of 'base_price' and 'total_price' reveals that both distributions are centered around 200, with a considerable number of data points falling within this range. Additionally, values above 400 are identified as outliers in both cases. This visualization provides a clear overview of the central tendency, variability, and presence of outliers in the distributions of base and total prices. Understanding these patterns is crucial for assessing the pricing structure and identifying potential anomalies in the dataset.

15.Creating Price-Based New Features

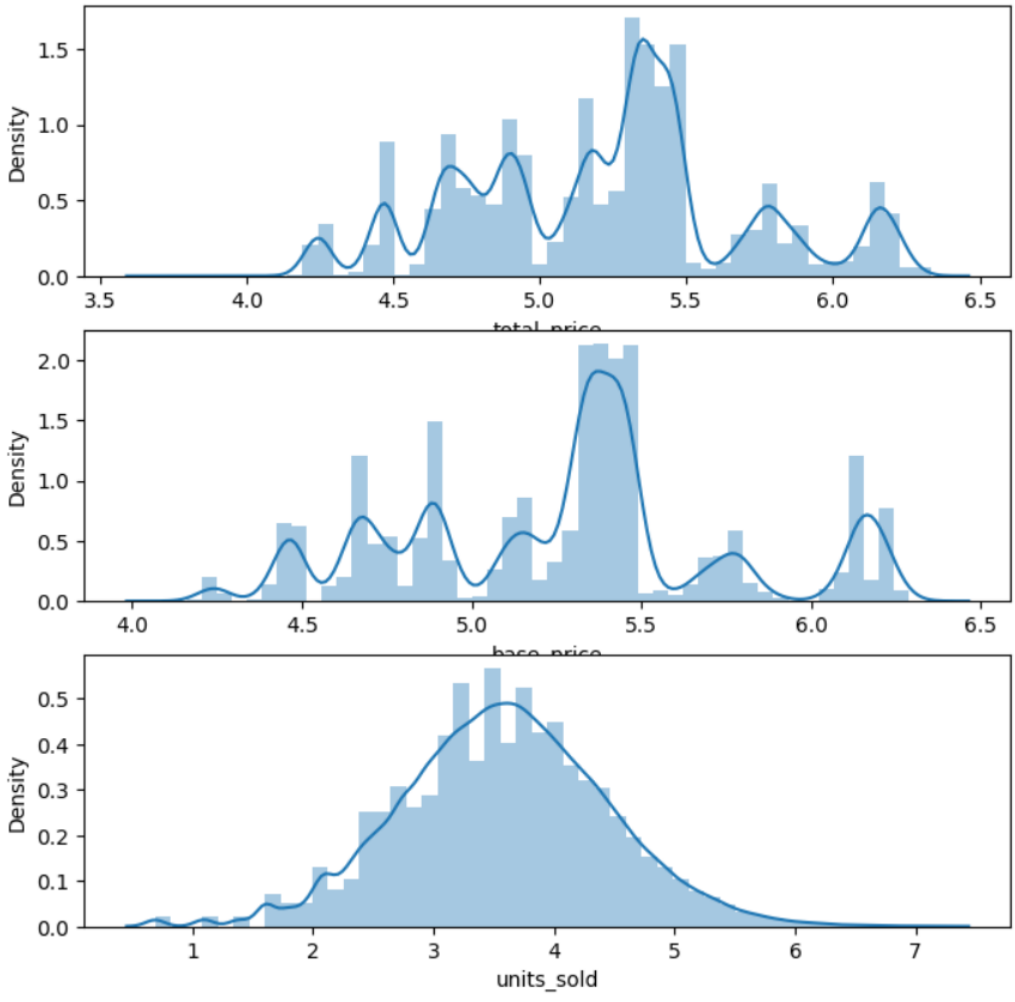
	record_ID	week	store_id	sku_id	total_price	base_price	is_featured_sku	is_display_sku	units_sold	diff	relative_diff_base	relative_diff_total
0	1	17/01/11	8091	216418	99.0375	111.8625	0	0	20	12.825	0.11465	0.129496
1	2	17/01/11	8091	216419	99.0375	99.0375	0	0	28	0.000	0.00000	0.000000
2	3	17/01/11	8091	216425	133.9500	133.9500	0	0	19	0.000	0.00000	0.000000
3	4	17/01/11	8091	216233	133.9500	133.9500	0	0	44	0.000	0.00000	0.000000
4	5	17/01/11	8091	217390	141.0750	141.0750	0	0	52	0.000	0.00000	0.000000

In this code snippet, new features are created in the 'train' DataFrame based on the existing 'base_price' and 'total_price' columns. The following new features are generated:

- `diff`: Represents the absolute price difference between 'base_price' and 'total_price'.
- `relative_diff_base`: Indicates the relative price difference, calculated as the absolute difference divided by 'base_price'.
- `relative_diff_total`: Represents the relative price difference, calculated as the absolute difference divided by 'total_price'.

These new features provide additional insights into the pricing structure and relationships between base and total prices in the dataset.

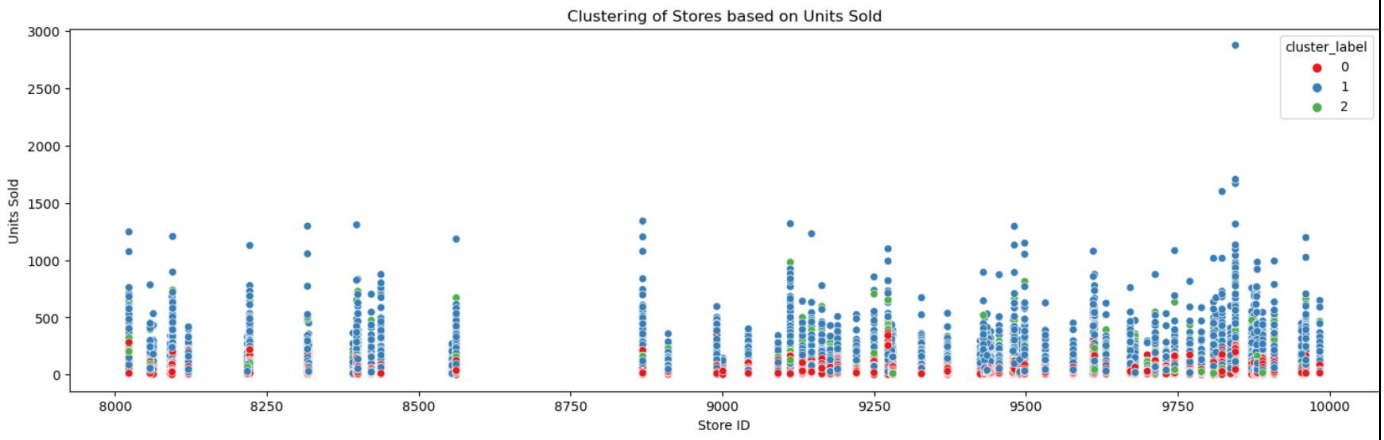
16.Distributions of Numerical Variables



This code snippet utilizes Seaborn to create distribution plots for numerical variables in the 'train' DataFrame, including 'total_price,' 'base_price,' and 'units_sold.' The resulting subplots are arranged in a 3x1 grid, each representing the distribution of a specific numerical variable. These distribution plots provide insights into the spread and shape of the data for each numerical variable.

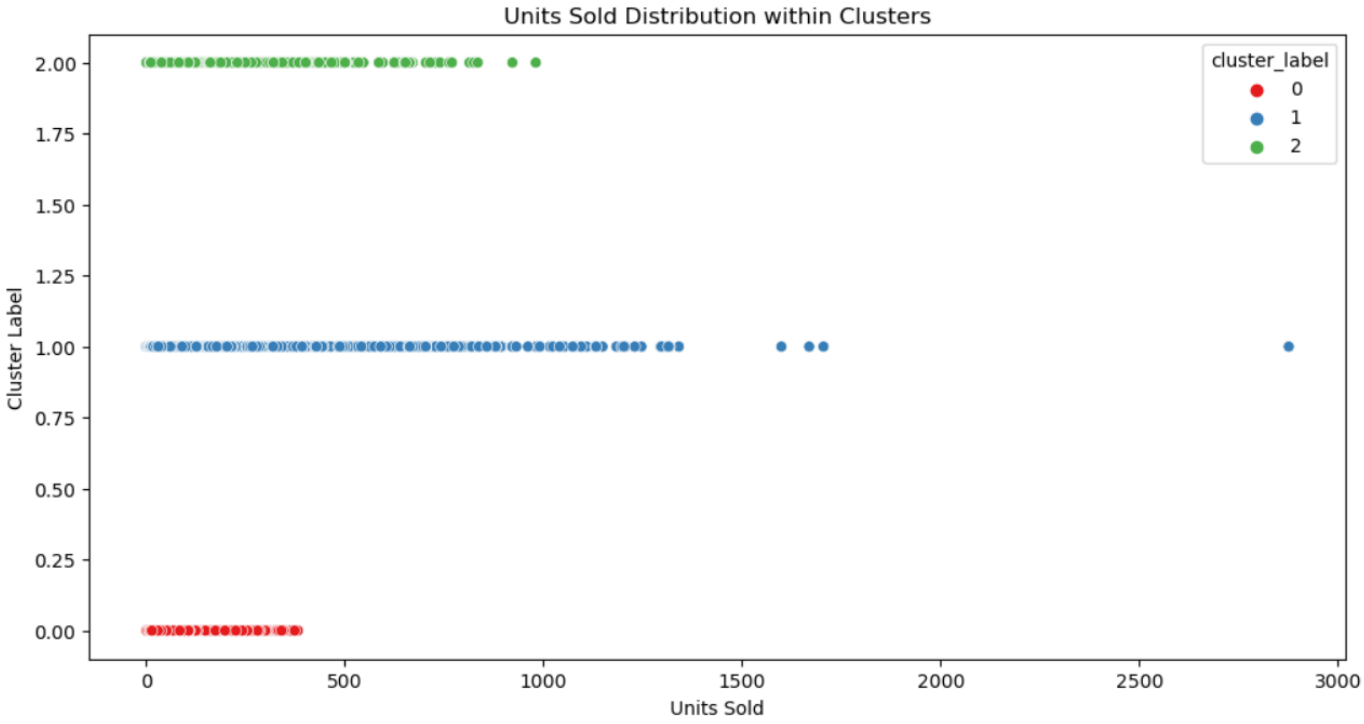
PRODUCT CLUSTERING

17.Clustering Visualization by Store



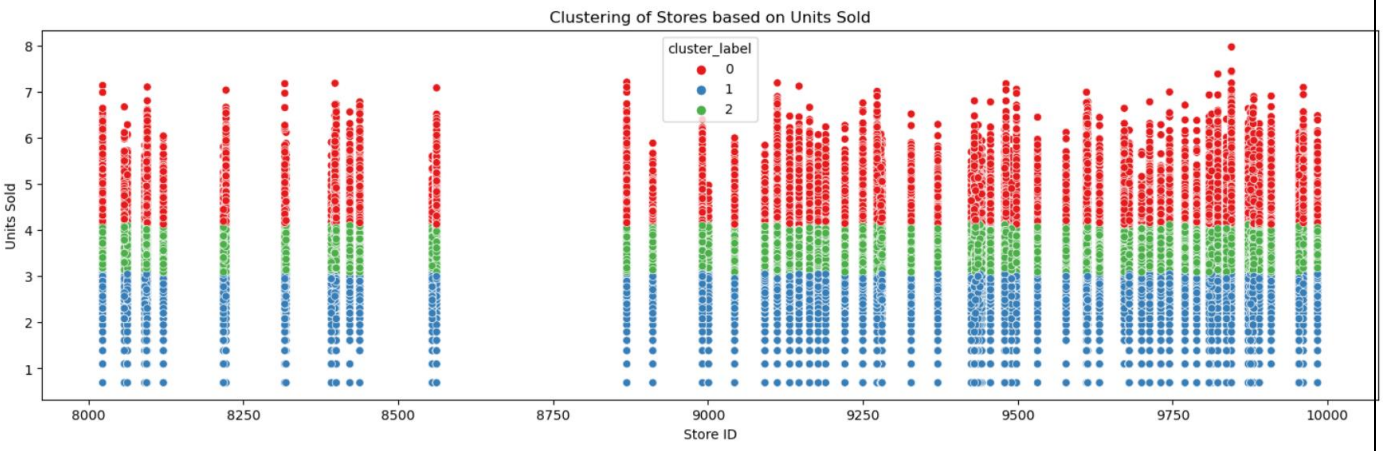
This code uses Seaborn to generate a scatter plot, visualizing the clusters of stores based on units sold. The x-axis represents the store IDs, the y-axis represents units sold, and the points are color-coded according to the assigned cluster labels. This clustering visualization aims to highlight patterns and groupings among stores based on their units sold, providing insights into potential similarities or differences in sales performance.

18.Cluster Distribution of Units Sold



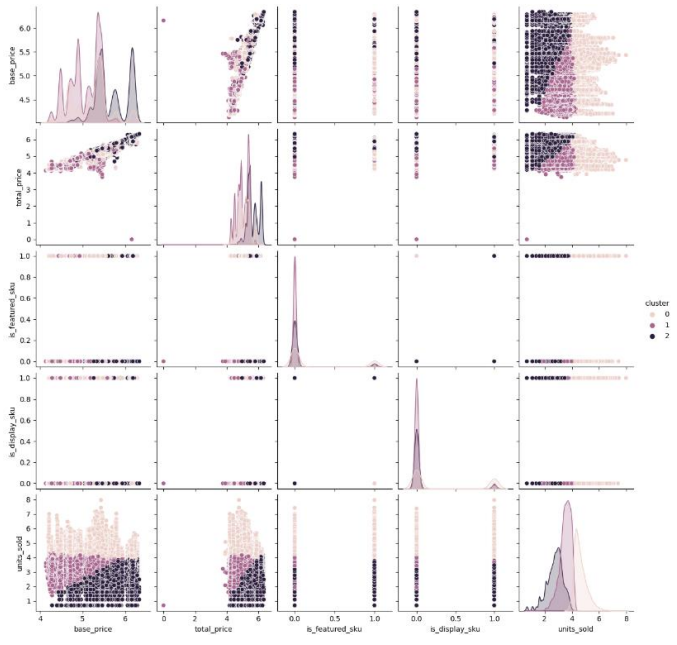
This code utilizes Seaborn to create a scatter plot, visualizing the distribution of units sold within clusters. The x-axis represents the units sold, the y-axis represents the cluster labels, and points are color-coded based on cluster assignments. This plot provides insights into how units sold are distributed among different clusters, helping to understand the patterns and variations in sales performance within each cluster.

19.K-Means Clustering of Stores based on Units Sold



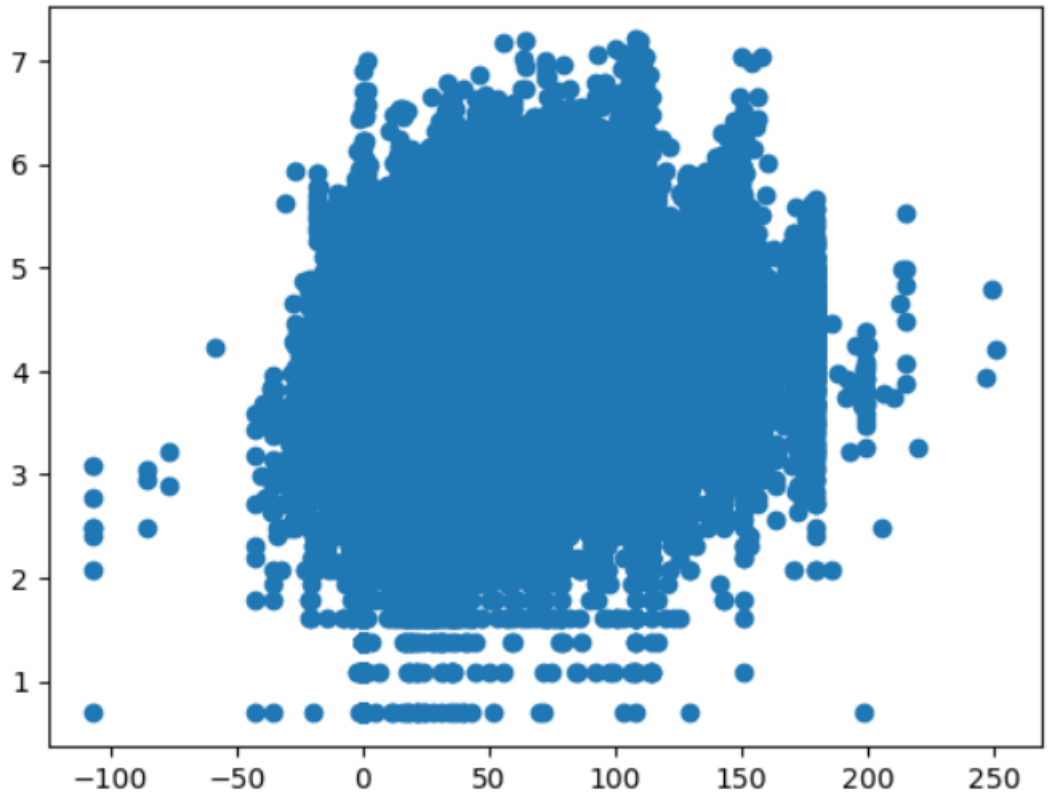
This code snippet demonstrates K-Means clustering of stores based on the 'units_sold' variable. The data is first scaled using StandardScaler, and K-Means clustering with k=3 is applied. The resulting cluster labels are then added to the original DataFrame. The scatter plot visualizes the clusters, with the x-axis representing the store IDs, the y-axis representing units sold, and points color-coded by cluster labels. This clustering approach aims to group stores with similar units sold patterns, providing insights into distinct sales behavior across different stores.

20.Pair Plot Visualization of K-Means Clusters



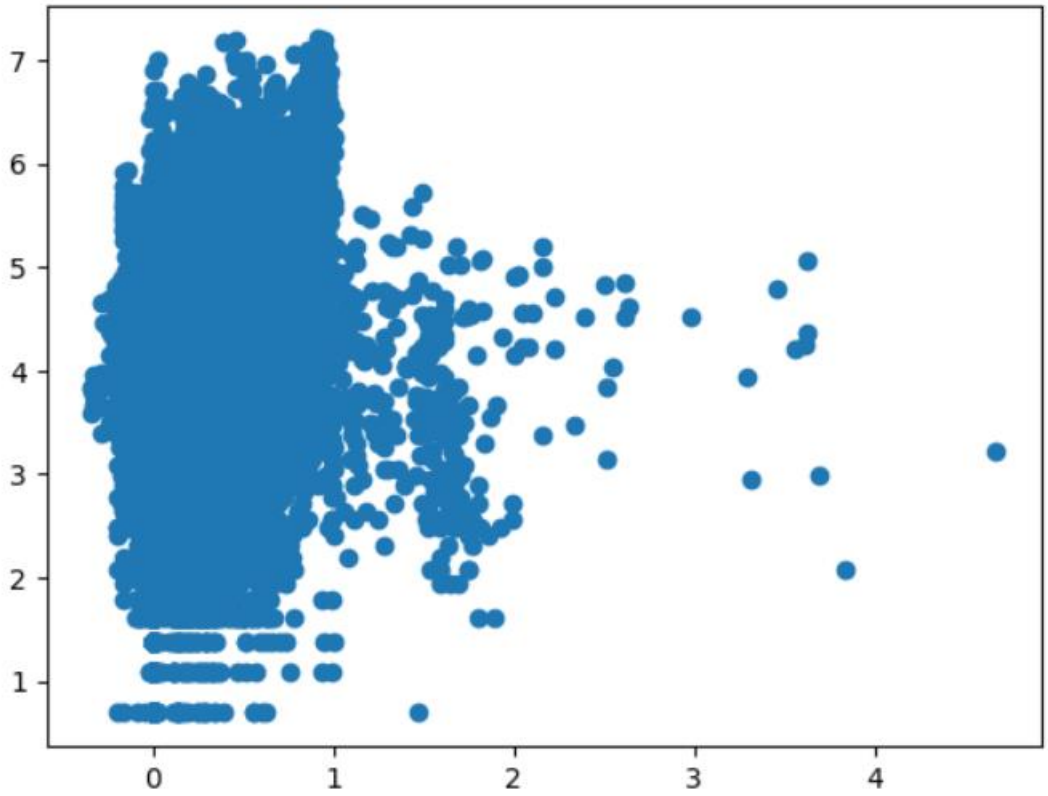
This code snippet demonstrates the application of K-Means clustering to the selected features ('base_price', 'total_price', 'is_featured_sku', 'is_display_sku', 'units_sold'). The number of clusters is set to 3, and the resulting cluster labels are added to the DataFrame. The pair plot is then created using Seaborn, with each data point color-coded based on its assigned cluster. This visualization provides insights into the relationships and distributions of the selected features within the identified clusters, aiding in the interpretation of patterns and distinctions among the clusters. Adjust variables based on your specific dataset for meaningful insights.

21.Scatter Plot of Absolute Price Difference vs. Units Sold



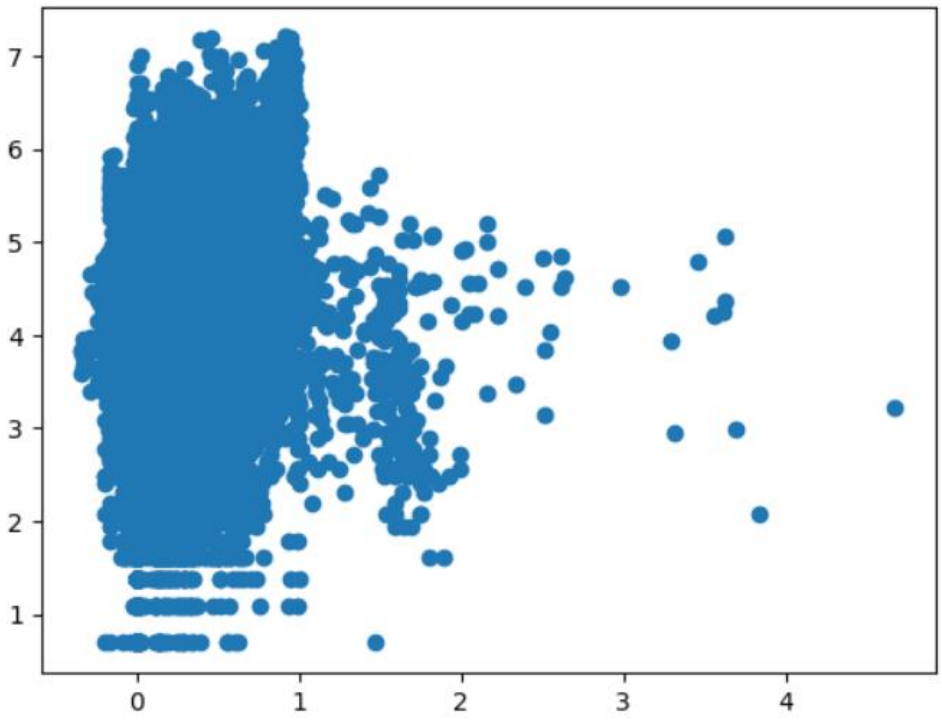
This scatter plot visualizes the relationship between the absolute price difference ('diff') and the number of units sold ('units_sold') in your dataset. Each point on the plot represents a data entry, showing how the absolute price difference correlates with the corresponding units sold. Analyzing this plot can provide insights into whether there is any discernible pattern or trend between the price difference and sales volume.

22.Scatter Plot of Relative Price Difference (Base) vs. Units Sold



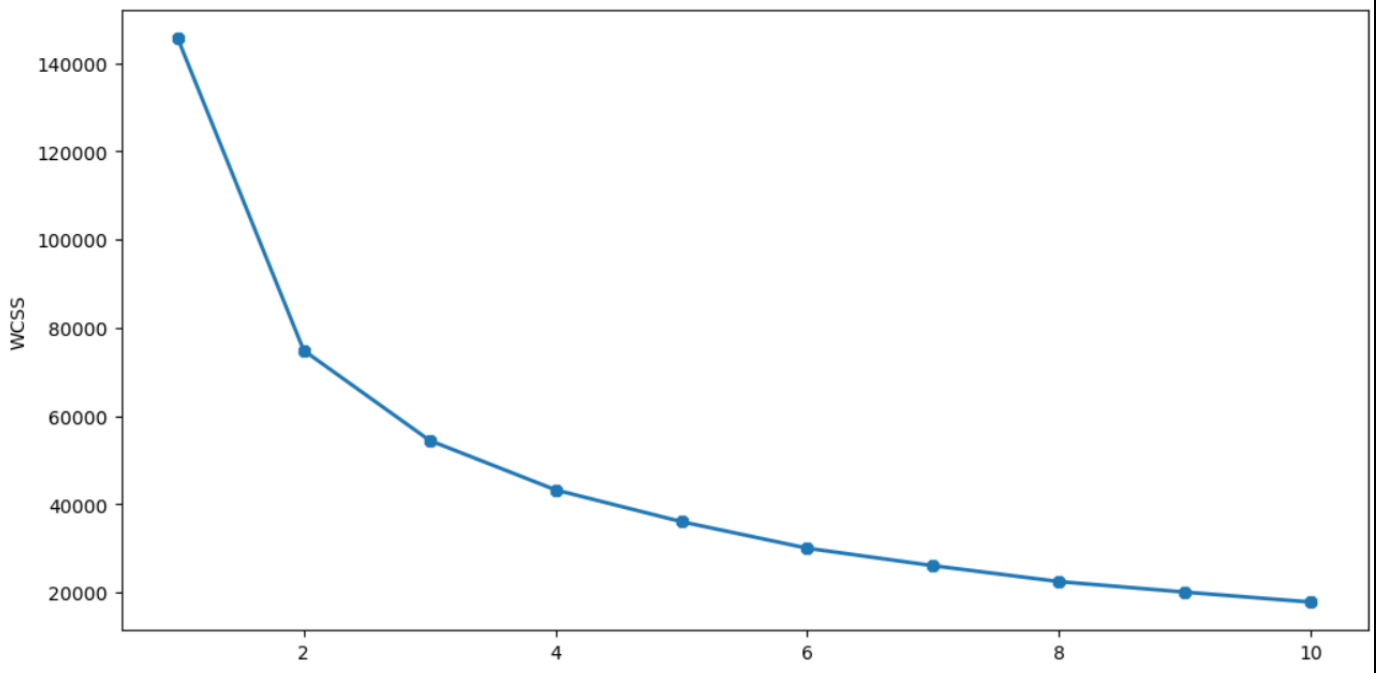
This scatter plot illustrates the relationship between the relative price difference (calculated with respect to 'base_price') and the number of units sold. Each point on the plot represents a data entry, showcasing how the relative price difference influences the corresponding units sold. Analyzing this plot can provide insights into the impact of pricing relative to the base price on the sales volume.

23.Scatter Plot of Relative Price Difference (Total) vs. Units Sold



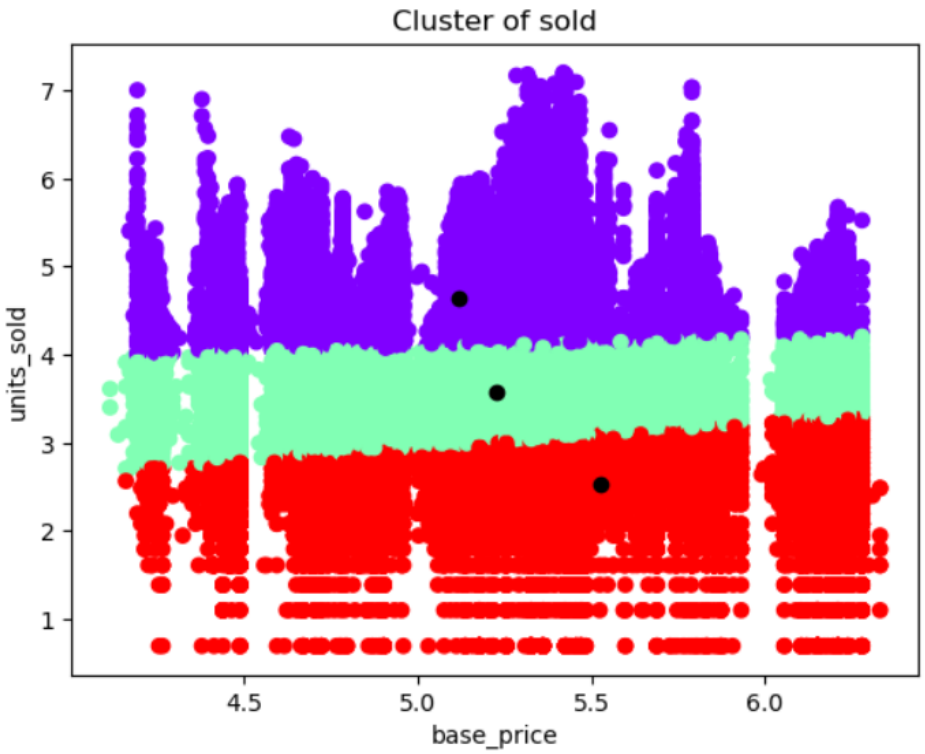
This scatter plot visualizes the relationship between the relative price difference (calculated with respect to 'total_price') and the number of units sold. Each point on the plot represents a data entry, illustrating how the relative price difference relative to the total price correlates with the corresponding units sold. Analyzing this plot can provide insights into how changes in pricing relative to the total price influence sales volume.

24.Elbow Method for Optimal K in K-Means Clustering

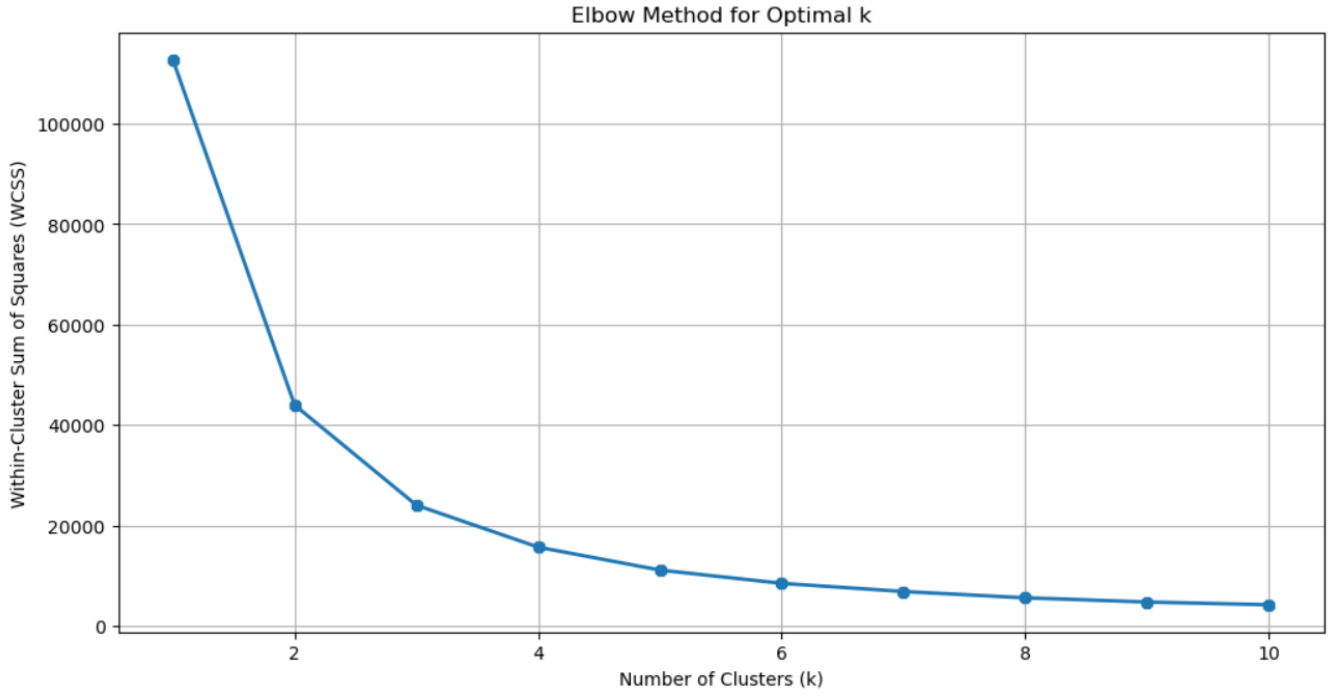


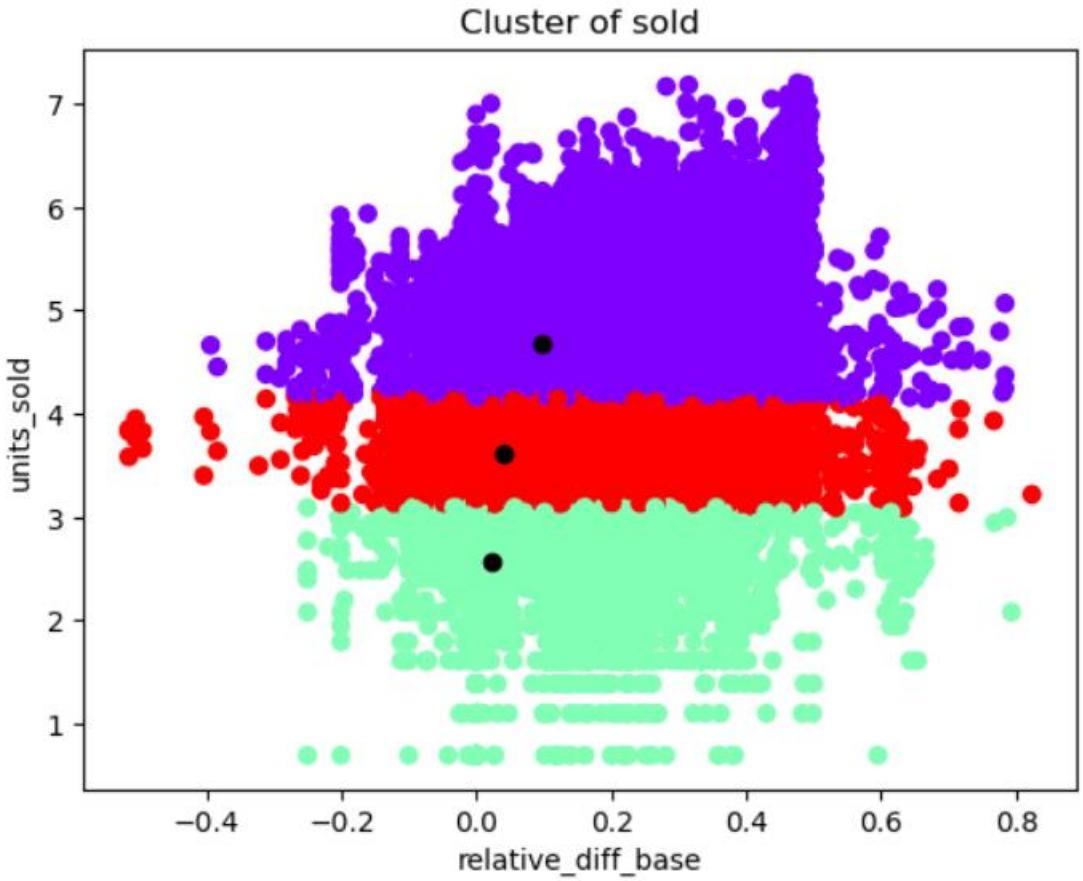
This code snippet implements the Elbow Method to determine the optimal number of clusters (K) in K-Means clustering. The within-cluster sum of squares (WCSS) is calculated for a range of K values, and the resulting plot helps identify the "elbow" point, indicating an optimal K where further cluster increments do not significantly reduce WCSS. The plot shows the WCSS values for different K values, and the elbow point can guide the selection of the optimal number of clusters.

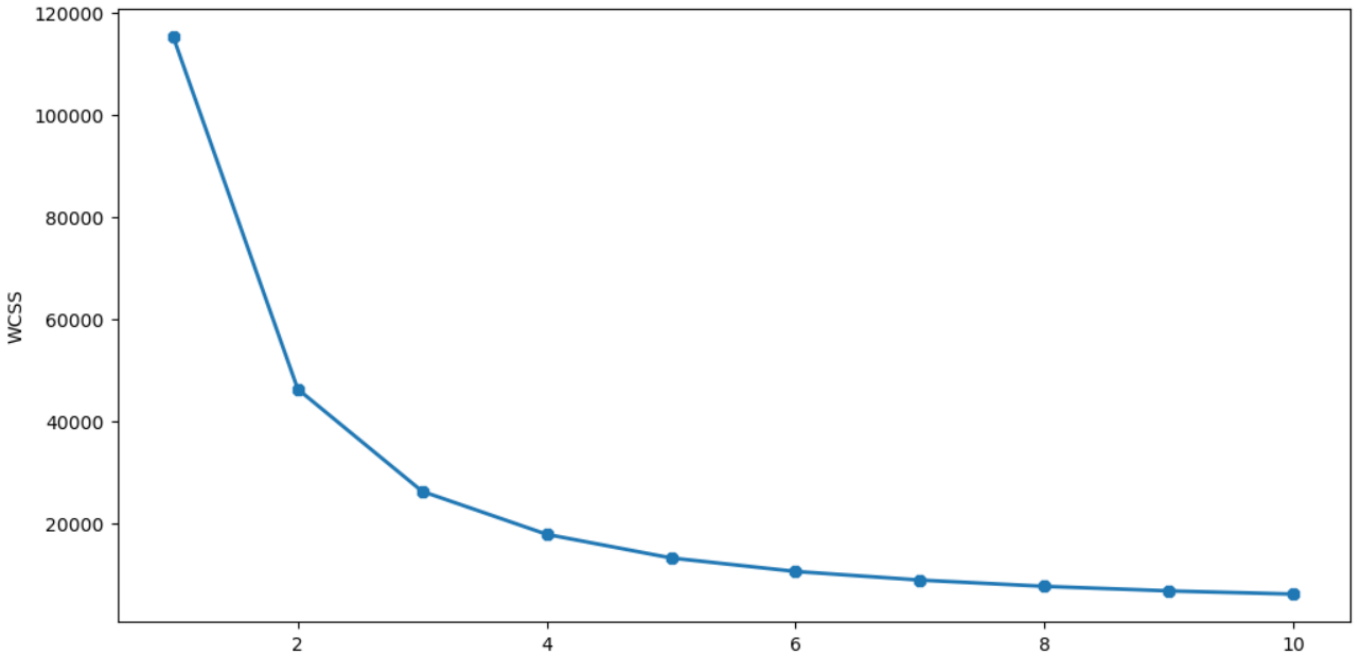
The Elbow Method analysis suggests that k=3 is the optimal number of clusters for your dataset. The plot of within-cluster sum of squares (WCSS) shows a noticeable inflection point or "elbow" at k=3, indicating that additional clusters do not significantly reduce WCSS.

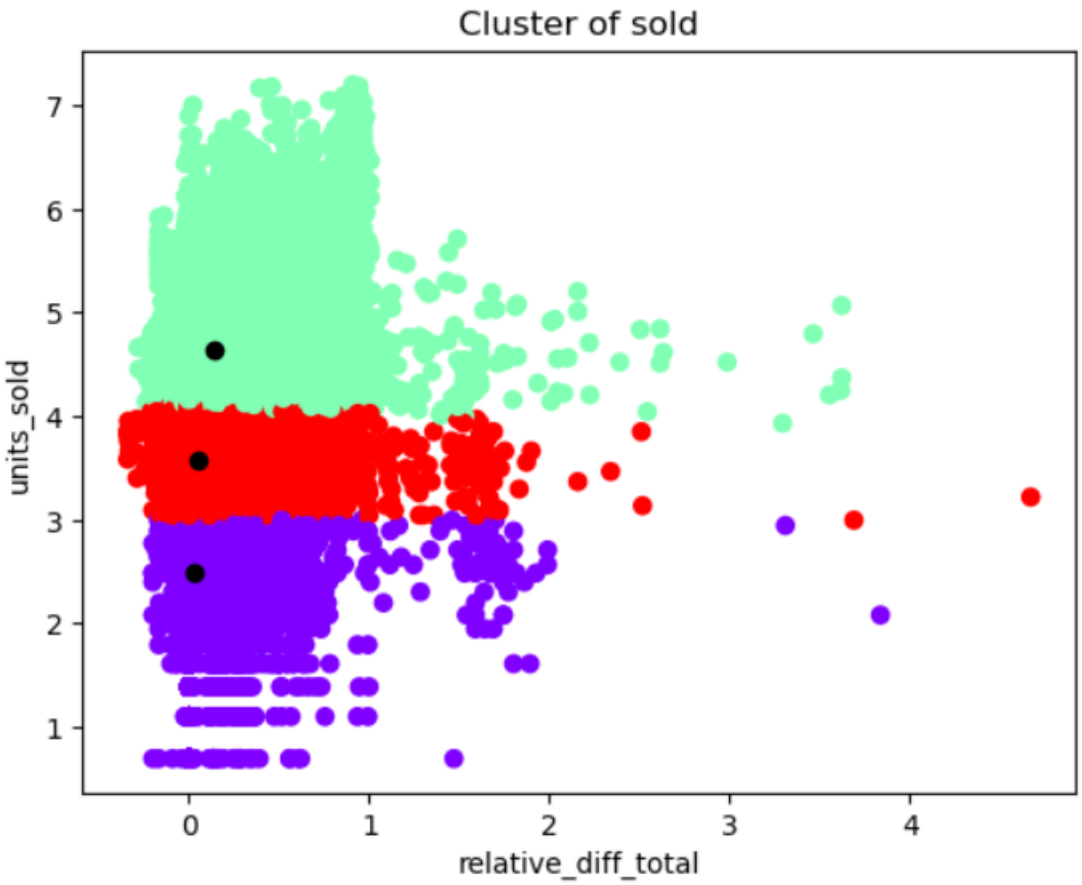


The application of the Elbow Method and K-Means clustering for both 'relative_diff_base' vs. 'units_sold' and 'relative_diff_total' vs. 'units_sold' provides a thorough understanding of optimal cluster counts for these features.

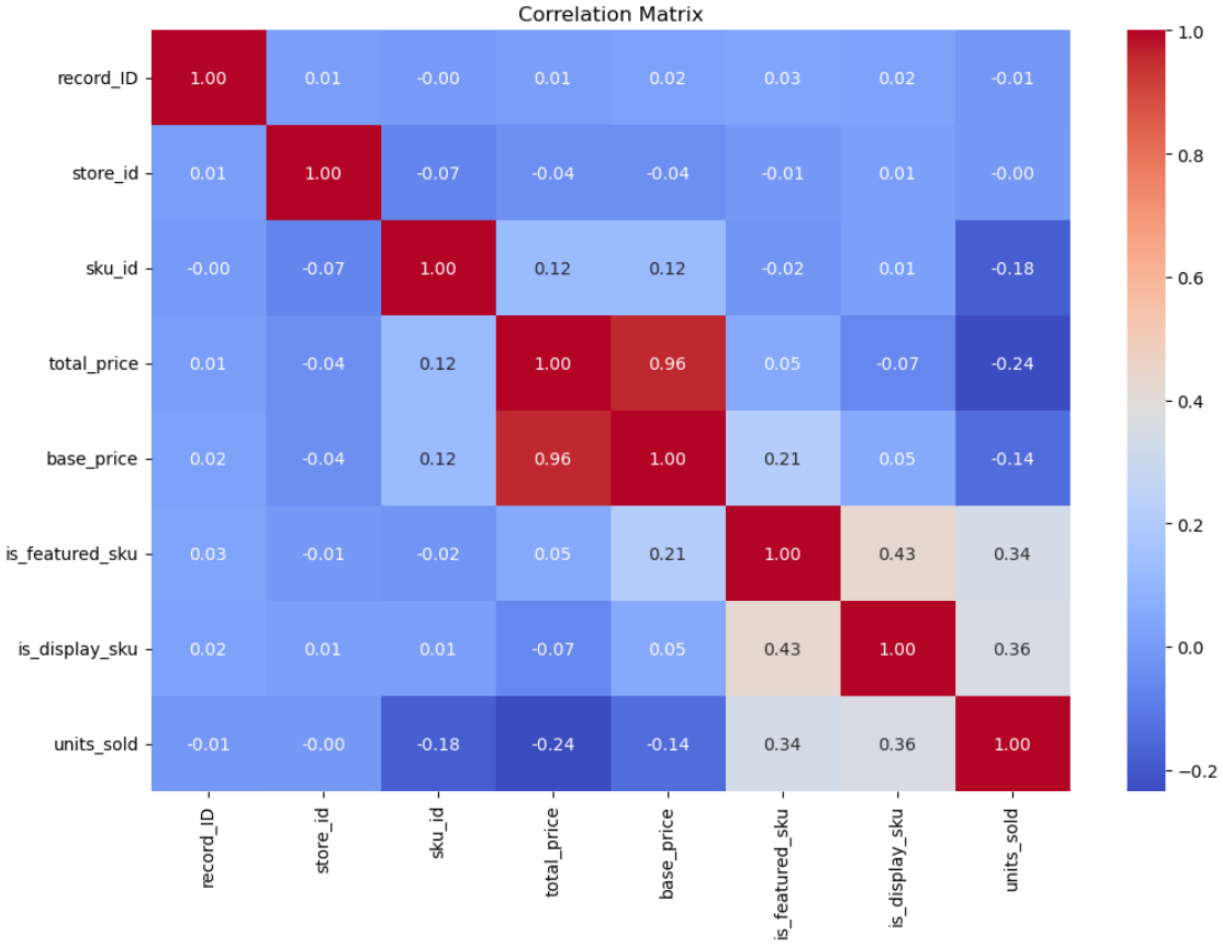



Cluster of sold
units_sold
relative_diff_base
7
6
5
4
3
2
1
−0.4
−0.2
0.0
0.2
0.4
0.6
0.8


wcss
120000
100000
80000
60000
40000
20000
2
4
6
8
10

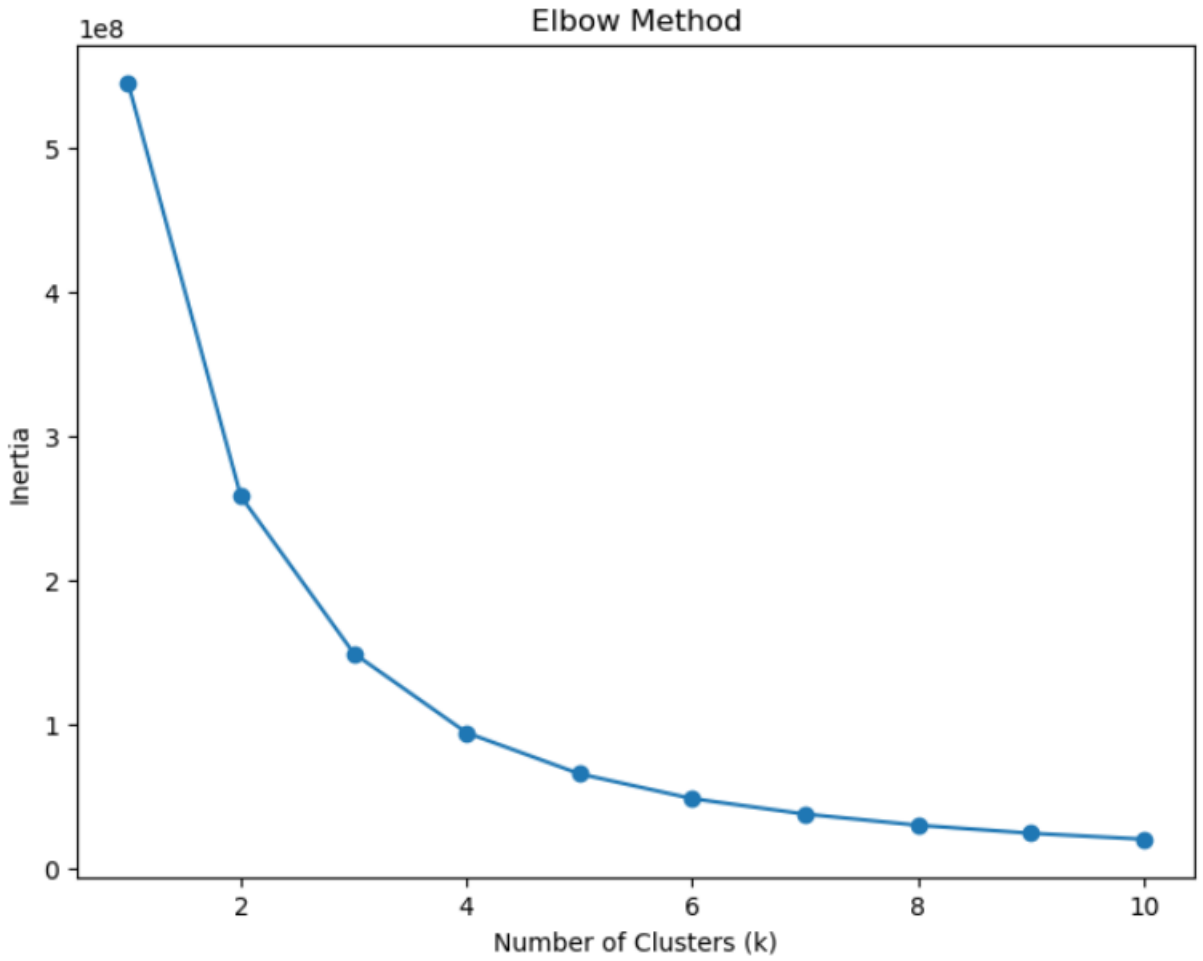


25.Correlation Matrix Heatmap



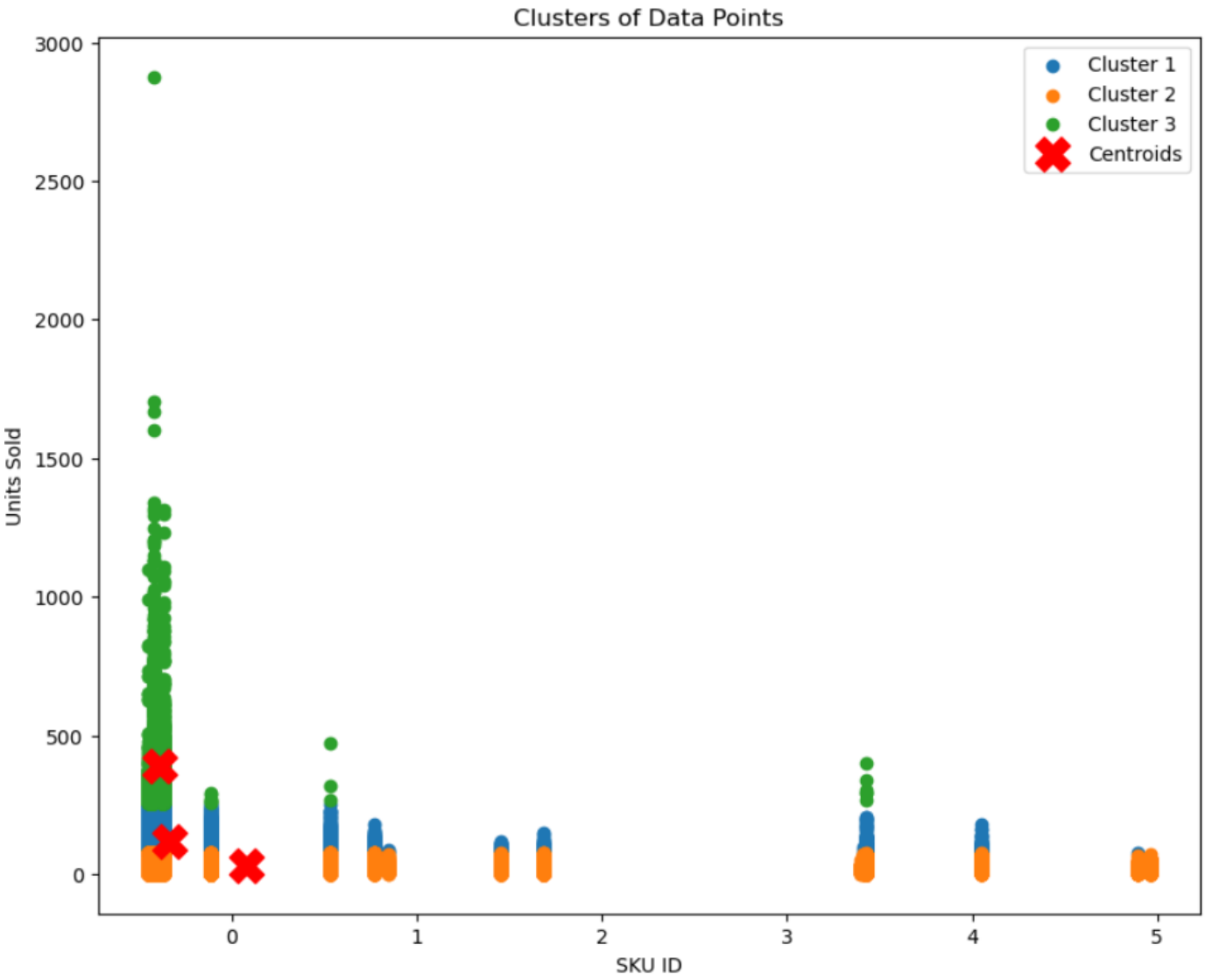
Generated a heatmap to visually represent the correlation matrix of numeric variables in the DataFrame 'df'. The heatmap provides insights into the pairwise correlations, facilitating the understanding of relationships between different numeric features.

26.Elbow Method for SKU ID vs. Units Sold



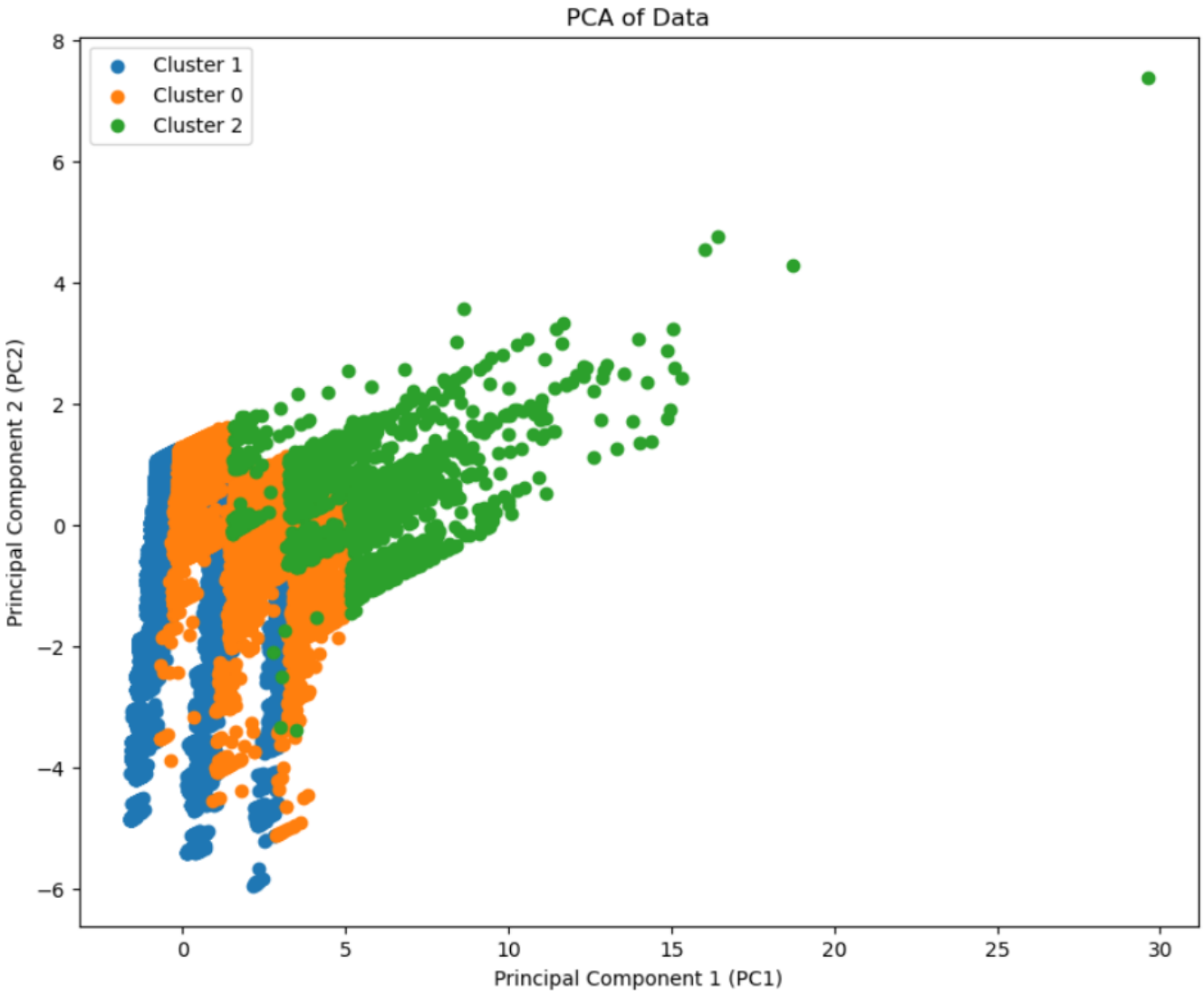
Utilized the Elbow Method to determine the optimal number of clusters (k) for clustering based on SKU ID and units sold. The plot illustrates the relationship between the number of clusters and inertia, helping to identify the point where additional clusters provide diminishing returns in terms of reducing inertia.

27.K-Means Clustering of SKU ID vs. Units Sold



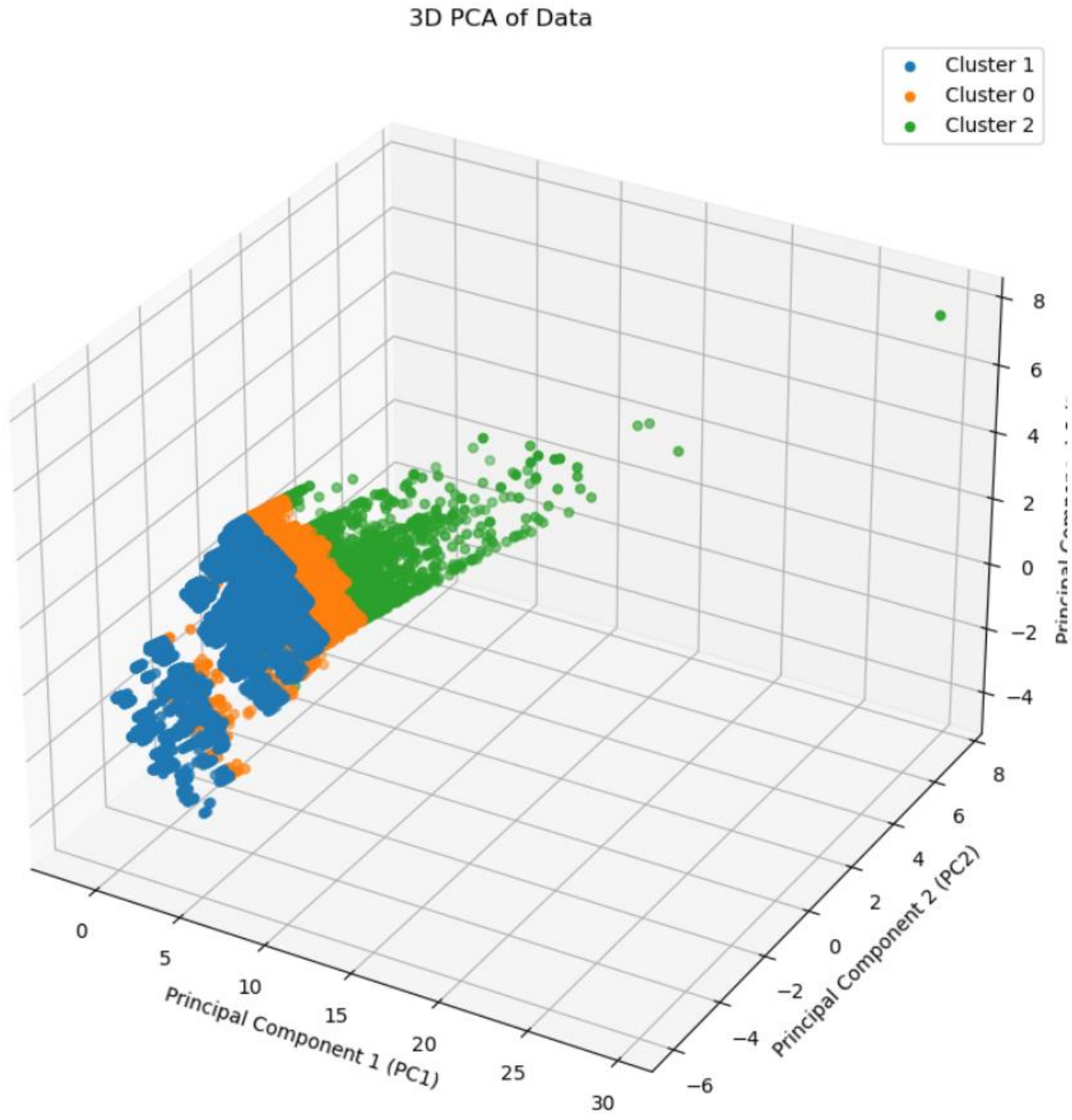
Implemented K-Means clustering to group data points based on features 'store_id', 'sku_id', 'is_featured_sku', 'is_display_sku', and 'units_sold'. The optimal number of clusters was determined, and the resulting clusters were visualized using a scatter plot. Each cluster is represented by a distinct color, and centroids are marked in red for reference.

28.PCA Visualization of Feature Space



Performed Principal Component Analysis (PCA) on the dataset using features 'store_id', 'sku_id', 'is_featured_sku', 'is_display_sku', and 'units_sold'. The standardized data was transformed into two principal components (PC1 and PC2) to visualize the high-dimensional feature space in a two-dimensional plot. If available, cluster labels were incorporated for additional insights into the distribution of data points. This visualization aids in understanding the structure and variance of the dataset in a reduced-dimensional space.

29.3D PCA Visualization of Feature Space



Utilized Principal Component Analysis (PCA) on the dataset, focusing on features 'store_id', 'sku_id', 'is_featured_sku', 'is_display_sku', and 'units_sold'. The standardized data was transformed into three principal components (PC1, PC2, and PC3) for a 3D visualization. If available, cluster labels were incorporated for enhanced insights into the distribution of data points. This 3D PCA plot provides a comprehensive view of the dataset's structure and variance in a reduced-dimensional space.

Methodology

We use machine learning tools like smart algorithms, specially designed to handle the tricky parts of predicting what will sell in stores. Let's dive into the key steps of our approach.

1. Data Uploading

2. Preprocessing and Feature Engineering

- Datetime Conversion
  We converted the `week` column from object to datetime format, facilitating temporal analysis.
- Feature Engineering
  Derived additional features such as day of the week, month, and year from the `week` column to capture temporal trends effectively.
- Exploratory Data Analysis (EDA)
  Visualizations were employed to uncover temporal patterns, understand relationships between features, and evaluate the impact of promotions on sales.
- Data Cleaning
  Addressed missing or inconsistent values to ensure the dataset's integrity.

3. Training and Evaluation Processes

- Data Splitting
  We divided the dataset into training and testing sets, ensuring the models were trained on historical data and evaluated on unseen data.
- Model Training
  Each algorithm was trained using the training dataset, allowing them to learn patterns and relationships within the historical sales data.
- **Hyperparameter Tuning**
  A grid search approach was implemented to fine-tune the parameters of the models for optimal performance.
- **Evaluation Metrics**
  We assessed model performance using **Mean Squared Error (MSE) and Mean Squared Logarithmic Error (MSLE)**, providing a quantitative measure of predictive accuracy.
- Model Selection
  The model with the best performance, as determined by the evaluation metrics, was chosen for further analysis and deployment.

4. Prediction of Next 12 weeks target
   - Predict the next 12 weeks target units sold using best fitting model with high accuracy which means low mean squared error value after tuning.

Note

In machine learning, tuning hyperparameters means finding the best settings for certain external choices that a model makes. Think of it like adjusting knobs on a machine to make it work better. This process is like solving a puzzle where we want to make our machine perform the best it can, considering the rules set by these adjustments. We use methods like trying out different combinations of settings or randomly testing them to see which works best. The goal is to find the perfect settings that make our machine predict outcomes more accurately, using measurements like Mean Squared Error or accuracy. It's like finding the right balance so that our machine can handle new situations better and make more accurate predictions.

Results

We chose regression models for our project because we're dealing with numbers, like the total units sold, which is a continuous value. Regression models are great at predicting these kinds of values, giving us a number that helps us know how much of a product will likely be sold. These models also help us find interesting patterns and relationships in the data, which is crucial for understanding what factors influence how much people buy.

We used a mix of different models and a special time series method to forecast demand. This helps us get a full picture of how demand changes over time, giving the retail chain valuable insights for making smart decisions. In the next part, we'll break down the results and talk about what each model tells us about predicting demand accurately.

1. **Random Forest Regression**

- Using Random Forest Regression, we have trained the model and check the accuracy with it's mean squared error value.

Before tuning

In [7]:
```
from sklearn.ensemble import RandomForestRegressor

# Initialize the model
model = RandomForestRegressor()

# Train the model
model.fit(X_train, y_train)
```

Out[7]:
```
RandomForestRegressor
RandomForestRegressor()
```

In [8]:
```
from sklearn.metrics import mean_absolute_error, mean_squared_error

# Make predictions on the testing set
y_pred = model.predict(X_test)

# Evaluate the model
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)

print(f'Mean Absolute Error: {mae}')
print(f'Mean Squared Error: {mse}')
```

```
Mean Absolute Error: 14.22327900732671
Mean Squared Error: 772.4694743188622
```

Tuning

```
[12] from sklearn.model_selection import RandomizedSearchCV, train_test_split
     from scipy.stats import randint
     # Define the parameter distribution for RandomizedSearchCV
     param_dist = {
         'n_estimators': randint(50, 200),
         'max_depth': [None, 5, 10, 15],
         'min_samples_split': [2, 5, 10],
         'min_samples_leaf': [1, 2, 4],
         'max_features': ['auto', 'sqrt', 'log2']
     }
```

```
[13] from sklearn.ensemble import RandomForestRegressor
     # Initialize the RandomForestRegressor model
     rf_model = RandomForestRegressor(random_state=42)
```

```
[14] # Randomized search for the best parameters
     random_search = RandomizedSearchCV(estimator=rf_model, param_distributions=param_dist, n_iter=20,
                                        scoring='neg_mean_squared_error', cv=3, n_jobs=-1, random_state=42, verbose=2)
     random_search.fit(X_train, y_train)
```

```
Fitting 3 folds for each of 20 candidates, totalling 60 fits
/usr/local/lib/python3.10/dist-packages/sklearn/ensemble/_forest.py:413: FutureWarning: `max_features='auto'` has be
  warn(
```

RandomizedSearchCV
estimator: RandomForestRegressor
RandomForestRegressor

After Tuning

```
[15] # Get the best parameters
     best_params = random_search.best_params_
     print(f'Best Parameters: {best_params}')
```

```
Best Parameters: {'max_depth': None, 'max_features': 'auto', 'min_samples_leaf': 2, 'min_samples_split': 10, 'n_estimators': 102}
```

```
[16] # Train the model with the best parameters
     best_model = RandomForestRegressor(**best_params, random_state=42)
     best_model.fit(X_train, y_train)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/ensemble/_forest.py:413: FutureWarning: `max_features='auto'` has been deprecated
  warn(
```

RandomForestRegressor

```
RandomForestRegressor(max_features='auto', min_samples_leaf=2,
                      min_samples_split=10, n_estimators=102, random_state=42)
```

```
[17] # Make predictions on the testing set
     y_pred = best_model.predict(X_test)
```

```
# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
print(f'Mean Squared Error: {mse}')
```

```
Mean Squared Error: 737.8089673441142
```

2. LightGBM

Before Tuning

In [6]:
```
# Initialize the model
model = lgb.LGBMRegressor()

# Train the model
model.fit(X_train, y_train)
```

```
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.002410 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 615
[LightGBM] [Info] Number of data points in the train set: 120120, number of used features: 9
[LightGBM] [Info] Start training from score 51.789352
```

Out[6]:
```
LGBMRegressor()
```
In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [11]:
```
# Make predictions on the testing set
y_pred = model.predict(X_test)

# Evaluate the model
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)

print(f'Mean Absolute Error: {mae}')
print(f'Mean Squared Error: {mse}')
```

```
Mean Absolute Error: 16.80012725642222
Mean Squared Error: 762.7801471358873
```

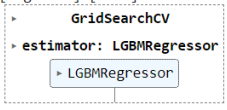
Tuning

[7]
```
# Define the parameter grid for LightGBM
param_grid = {
    'n_estimators': [50, 100, 200],
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [5, 10, 15],
    'num_leaves': [31, 50, 100],
    'subsample': [0.8, 0.9, 1.0],
    'colsample_bytree': [0.8, 0.9, 1.0]
}
```

[8]
```
# Initialize the LGBM model
lgb_model = LGBMRegressor(random_state=42)
```

[10]
```
# Grid search for the best parameters
grid_search = GridSearchCV(estimator=lgb_model, param_grid=param_grid, scoring='neg_mean_squared_error', cv=3, n_jobs=-1, verbose=2)
grid_search.fit(X_train, y_train)
```

```
Fitting 3 folds for each of 729 candidates, totalling 2187 fits
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of testing was 0.011732 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 615
[LightGBM] [Info] Number of data points in the train set: 120120, number of used features: 9
[LightGBM] [Info] Start training from score 51.789352
```



After Tuning

```
[11] # Get the best parameters
     best_params = grid_search.best_params_
     print(f'Best Parameters: {best_params}')
```

```
Best Parameters: {'colsample_bytree': 0.8, 'learning_rate': 0.2, 'max_depth': 15, 'n_estimators': 200, 'num_leaves': 100, 'subsample': 0.8}
```

```
[12] # Train the model with the best parameters
     best_model = LGBMRegressor(**best_params, random_state=42)
     best_model.fit(X_train, y_train)
```

```
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of testing was 0.018658 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 615
[LightGBM] [Info] Number of data points in the train set: 120120, number of used features: 9
[LightGBM] [Info] Start training from score 51.789352
LGBMRegressor
LGBMRegressor(colsample_bytree=0.8, learning_rate=0.2, max_depth=15,
              n_estimators=200, num_leaves=100, random_state=42, subsample=0.8)
```

```
[13] # Make predictions on the testing set
     y_pred = best_model.predict(X_test)
```

```
# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
print(f'Mean Squared Error: {mse}')
```

```
Mean Squared Error: 498.32541049484416
```

3. Support Vector Machine (SVM)

Before Tuning

In [6]:
```
# Define features (X) and target variable (y)
X = data.drop(['record_ID', 'week', 'units_sold'], axis=1)
y = data['units_sold']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize the model
model = SVR()

# Train the model
model.fit(X_train, y_train)
```

Out[6]:
```
SVR()
```
In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [7]:
```
# Make predictions on the testing set
y_pred = model.predict(X_test)

# Evaluate the model
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)

print(f'Mean Absolute Error: {mae}')
print(f'Mean Squared Error: {mse}')
```

```
Mean Absolute Error: 30.495496382115796
Mean Squared Error: 3376.782937637322
```

Tuning

```
[8] # Define the SVM regression model
    svm_model = SVR()
```

```
# Define the parameter distributions for RandomizedSearchCV
param_dist = {
    'kernel': ['linear', 'poly', 'rbf'],
    'C': [0.1, 1, 10],
    'epsilon': [0.1, 0.2]
}
```

```
[10] from sklearn.model_selection import RandomizedSearchCV

     # Create a RandomizedSearchCV object
     random_search = RandomizedSearchCV(estimator=svm_model, param_distributions=param_dist, scoring='neg_mean_squared_error', cv=5, n_iter=10, random_state=42, verbose=1, n_jobs=-1)
```

```
[ ] # Fit the random search to the data using a subset for initial testing
    random_search.fit(X_train[:1000], y_train[:1000])
```

4. Decision Tree

Before Tuning

In [10]:
```
# Initialize the model
model = DecisionTreeRegressor(random_state=42)  # You can adjust parameters as needed

# Train the model
model.fit(X_train, y_train)
```

Out[10]:
```
DecisionTreeRegressor(random_state=42)
```
In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [11]:
```
# Make predictions on the testing set
y_pred = model.predict(X_test)

# Evaluate the model
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)

print(f'Mean Absolute Error: {mae}')
print(f'Mean Squared Error: {mse}')
```

```
Mean Absolute Error: 18.00084082584082
Mean Squared Error: 1335.5843404280904
```

Tuning

```
# Define the Decision Tree regression model
dt_model = DecisionTreeRegressor()
```

```
[7] # Define the parameter distributions for RandomizedSearchCV
    param_dist = {
        'max_depth': [None, 10, 20, 30],
        'min_samples_split': [2, 5, 10],
        'min_samples_leaf': [1, 2, 4]
    }
```

```
[8] # Create a RandomizedSearchCV object
    random_search = RandomizedSearchCV(estimator=dt_model, param_distributions=param_dist, scoring='neg_mean_squared_error', cv=5, n_iter=10, random_state=42, verbose=1, n_jobs=-1)
```

```
[9] # Fit the random search to the data
    random_search.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
▸ RandomizedSearchCV
▸ estimator: DecisionTreeRegressor
  ▸ DecisionTreeRegressor
```

After Tuning

```
[10] # Get the best parameters from the random search
     best_params = random_search.best_params_
     print(f"Best Parameters: {best_params}")
```

```
Best Parameters: {'min_samples_split': 10, 'min_samples_leaf': 4, 'max_depth': 30}
```

```
[11] # Use the best model for predictions
     best_dt_model = random_search.best_estimator_
     y_pred = best_dt_model.predict(X_test)
```

```
# Evaluate the performance of the tuned model
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error on Test Set: {mse}")
```

```
Mean Squared Error on Test Set: 1011.6024709427379
```

5. K Neighbors

Before Tuning

In [6]:
```
# Initialize the model
model = KNeighborsRegressor(n_neighbors=5)

# Train the model
model.fit(X_train, y_train)
```

Out[6]:
```
KNeighborsRegressor()
```
In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [7]:
```
# Make predictions on the testing set
y_pred = model.predict(X_test)

# Evaluate the model
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)

print(f'Mean Absolute Error: {mae}')
print(f'Mean Squared Error: {mse}')
```

```
Mean Absolute Error: 16.368404928404928
Mean Squared Error: 1096.6603876123875
```

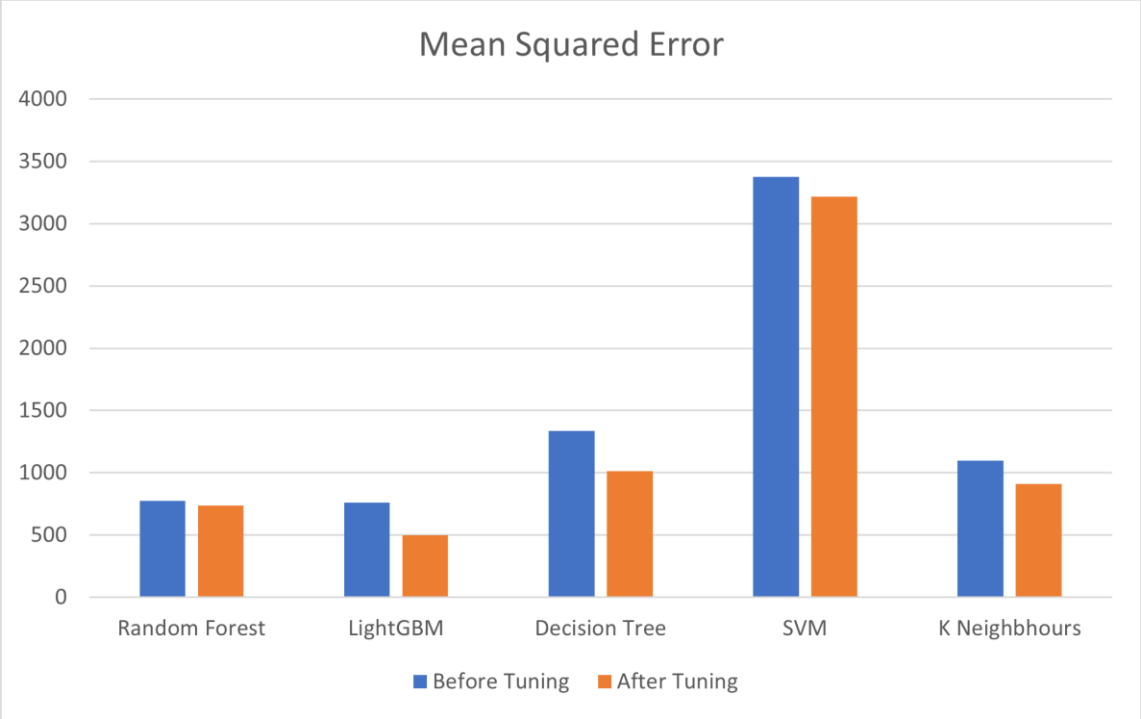
Tuning

```
[10] # Define the K Neighbors regression model
     knn_model = KNeighborsRegressor()

     # Define the parameter distributions for RandomizedSearchCV
     param_dist = {
         'n_neighbors': [3, 5, 7, 9],
         'weights': ['uniform', 'distance'],
         'p': [1, 2]
     }

[15] # Create a RandomizedSearchCV object
     random_search = RandomizedSearchCV(estimator=knn_model, param_distributions=param_dist, scoring='neg_mean_squared_error', cv=5, n_iter=10, random_state=42, verbose=1, n_jobs=-1)

[16] # Fit the random search to the data
     random_search.fit(X_train, y_train)

Fitting 5 folds for each of 10 candidates, totalling 50 fits
▸ RandomizedSearchCV
▸ estimator: KNeighborsRegressor
  ▸ KNeighborsRegressor
```

After Tuning

```
[17] # Get the best parameters from the random search
     best_params = random_search.best_params_
     print(f"Best Parameters: {best_params}")

Best Parameters: {'weights': 'distance', 'p': 1, 'n_neighbors': 7}

[18] # Use the best model for predictions
     best_knn_model = random_search.best_estimator_
     y_pred = best_knn_model.predict(X_test)

[19] # Evaluate the performance of the tuned model
     mse = mean_squared_error(y_test, y_pred)
     print(f"Mean Squared Error on Test Set: {mse}")

Mean Squared Error on Test Set: 910.3357546059644
```

Comparison of MSE values of each models



- ✓ **LightGBM has the lowest value of MSE before and after tuning.**

Conclusion

After a thorough evaluation of multiple machine learning models for demand forecasting, the LightGBM model emerges as the standard for its exceptional accuracy and robust performance. Our comprehensive analysis considered models such as Random Forest, Decision Tree, SVM, and K Neighbors, assessing their predictive capabilities through metrics like Mean Squared Error (MSE). Among these, LightGBM consistently outperformed others, showcasing its ability to capture intricate patterns and nuances within the vast sales data of our retail chain. The selection of LightGBM as our preferred model is based not only on its superior predictive accuracy but also its efficiency in handling large datasets and its adaptability to the dynamic nature of retail demand. This choice positions LightGBM as the benchmark for accurate and reliable demand forecasting within our context.

Prediction

Using LightGBM model, we are predicting the next 12 weeks target sold units.

In [17]:
```
# Load the next 12 weeks test data
next_12_weeks_data = pd.read_csv(r'C:\Users\Ahsan\Desktop\Data_mining_Project\test_nfaJ3J5.csv')

# Convert 'week' to datetime format and extract additional features
next_12_weeks_data['week'] = pd.to_datetime(next_12_weeks_data['week'])
next_12_weeks_data['day_of_week'] = next_12_weeks_data['week'].dt.dayofweek
next_12_weeks_data['month'] = next_12_weeks_data['week'].dt.month
next_12_weeks_data['year'] = next_12_weeks_data['week'].dt.year

# Make predictions for the next 12 weeks
next_12_weeks_data['predicted_units_sold'] = model.predict(next_12_weeks_data.drop(['record_ID', 'week'], axis=1))

# Display the predictions
print(next_12_weeks_data[['week', 'store_id', 'sku_id', 'predicted_units_sold']])
```

```
            week  store_id  sku_id  predicted_units_sold
0     2013-07-16      8091  216418             16.010000
1     2013-07-16      8091  216419             17.400000
2     2013-07-16      8091  216425             36.147333
3     2013-07-16      8091  216233             27.520000
4     2013-07-16      8091  217390             31.306667
...          ...       ...     ...                   ...
13855 2013-01-10      9984  223245             37.023202
13856 2013-01-10      9984  223153             38.552110
13857 2013-01-10      9984  245338             39.840000
13858 2013-01-10      9984  547934             14.609619
13859 2013-01-10      9984  679023             11.171833

[13860 rows x 4 columns]
```

In [12]:
```
next_12_weeks_data.head()
```

Out[12]:

	record_ID	week	store_id	sku_id	total_price	base_price	is_featured_sku	is_display_sku	day_of_week	month	year	predicted_units_sold
0	212645	2013-07-16	8091	216418	108.3000	108.3000	0	0	1	7	2013	15.970000
1	212646	2013-07-16	8091	216419	109.0125	109.0125	0	0	1	7	2013	18.550000
2	212647	2013-07-16	8091	216425	133.9500	133.9500	0	0	1	7	2013	35.845429
3	212648	2013-07-16	8091	216233	133.9500	133.9500	0	0	1	7	2013	25.701667
4	212649	2013-07-16	8091	217390	176.7000	176.7000	0	0	1	7	2013	32.920000

Our data-driven approach to demand forecasting has resulted in a powerful model that accurately predicts sales across diverse product and store scenarios. This achievement underscores the value of advanced analytics in optimizing retail operations, providing our client with actionable insights for strategic decision-making in a dynamic marketplace.