```
In [262…  import pandas as pd
          import numpy as np
          from sklearn.preprocessing import StandardScaler, LabelEncoder
          import matplotlib.pyplot as plt
          import seaborn as sns
```

```
In [279…  %matplotlib inline
```

```
In [280…  d1 = pd.read_csv('/Users/tolulopejohnson/Downloads/All_Data_Aldi.csv')
```

```
In [81]:  d2 = pd.read_csv('/Users/tolulopejohnson/Downloads/All_Data_ASDA.csv')
```

```
/var/folders/04/6sptnwmx1pd72p7w91rnf6100000gn/T/ipykernel_40701/2105526031.py:1: DtypeWarning: Columns (7) 
have mixed types. Specify dtype option on import or set low_memory=False.
  d2 = pd.read_csv('/Users/tolulopejohnson/Downloads/All_Data_ASDA.csv')
```

```
In [83]:  d3 = pd.read_csv('/Users/tolulopejohnson/Downloads/All_Data_Morrisons.csv')
```

```
In [11]:  d4 = pd.read_csv('/Users/tolulopejohnson/Downloads/All_Data_Sains.csv')
```

```
In [12]:  d5 = pd.read_csv('/Users/tolulopejohnson/Downloads/All_Data_Tesco.csv')
```

1st step

```
In [39]:  # List of individual DataFrames
          dataframes = [d1, d2, d3, d4, d5]
          df_names = ["d1", "d2", "d3", "d4", "d5"]

          # Loop through each DataFrame and display its shape before merging
          for name, df in zip(df_names, dataframes):
              rows, cols = df.shape
              print(f"Before Merging → DataFrame: {name} → Rows: {rows}, Columns: {cols}")

          # Merge all DataFrames into one
```

```python
df = pd.concat(dataframes, ignore_index=True)

# Print shape of merged DataFrame
print(f"\nAfter Merging → Merged DataFrame → Rows: {df.shape[0]}, Columns: {df.shape[1]}")
```

```
Before Merging → DataFrame: d1 → Rows: 464863, Columns: 8
Before Merging → DataFrame: d2 → Rows: 2456414, Columns: 8
Before Merging → DataFrame: d3 → Rows: 1794065, Columns: 8
Before Merging → DataFrame: d4 → Rows: 2600289, Columns: 8
Before Merging → DataFrame: d5 → Rows: 2213611, Columns: 8

After Merging → Merged DataFrame → Rows: 9529242, Columns: 8
```

In [37]:

```python
# Check missing values
print("Missing values per column before handling:\n", df.isnull().sum())
```

```
Missing values per column before handling:
 supermarket          0
prices_(£)          7
prices_unit_(£)    524
unit               524
names               26
date                0
category            0
own_brand           26
dtype: int64
```

In [43]:

```python
# Fill missing values and assign back to the original DataFrame


df["prices_(£)"] = df["prices_(£)"].fillna(df["prices_(£)"].median())
df["prices_unit_(£)"] = df["prices_unit_(£)"].fillna(df["prices_unit_(£)"].median())
df["unit"] = df["unit"].fillna(df["unit"].mode()[0])
df["names"] = df["names"].fillna("Unknown Product")
df["own_brand"] = df["own_brand"].fillna(False)

# Display missing values after handling
print("Missing values per column after handling:\n", df.isnull().sum())
```

```
Missing values per column after handling:
 supermarket        0
prices_(£)         0
prices_unit_(£)    0
unit               0
names              0
date               0
category           0
own_brand          0
dtype: int64
```

In [47]:
```python
# merge all data into a single data frame

df.to_csv("merged_supermarket_data.csv", index=False)
```

In [52]:
```python
# Export to my downloads
df.to_csv("/Users/tolulopejohnson/Downloads/merged_supermarket_data.csv", index=False)
```

2nd step

✅ It converts the 'date' column from a string or number format (like 20240413) ➡️ into proper datetime format (like 2024-04-13), which is essential for any time-based analysis.

In [56]:
```python
df['date'] = pd.to_datetime(df['date'], format='%Y%m%d', errors='coerce')
```

3rd step

0.14 is a numeric value if it's already stored as a float or integer inside your DataFrame.

However, sometimes when loading large CSVs (especially from web scrapers), values that look like numbers (e.g. 0.14) are actually stored as strings (like '0.14'). That’s why i explicitly run:

In [62]:
```python
# Convert price columns to numeric
```

```
df["prices_(£)"] = pd.to_numeric(df["prices_(£)"], errors="coerce")
df["prices_unit_(£)"] = pd.to_numeric(df["prices_unit_(£)"], errors="coerce")
```

In [64]:
```
### to check data type

df.dtypes
```

Out[64]:
```
supermarket                 object
prices_(£)                 float64
prices_unit_(£)            float64
unit                        object
names                       object
date                datetime64[ns]
category                    object
own_brand                     bool
dtype: object
```

In [66]:
```
from sklearn.preprocessing import LabelEncoder

label_cols = ['supermarket', 'category', 'unit']
le = LabelEncoder()

for col in label_cols:
    df[col] = le.fit_transform(df[col])
```

In [67]:
```
df = pd.get_dummies(df, columns=['supermarket', 'category', 'unit'], drop_first=True)
```

4th step scale the numerical features so they're on a similar range.

This helps many machine learning models perform better (especially distance- or gradient-based ones like logistic regression, KNN, or SVM).

In [ ]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
```

```
# Select numeric columns to scale (excluding target variable)
numeric_cols = ['prices_(£)', 'prices_unit_(£)']
df[numeric_cols] = scaler.fit_transform(df[numeric_cols])
```

nicely cleaned and fully preprocessed!

In [72]:
```
df.isnull().sum()
```

Out[72]:
```
prices_(£)         0
prices_unit_(£)    0
names              0
date               0
own_brand          0
supermarket_1      0
supermarket_2      0
supermarket_3      0
supermarket_4      0
category_1         0
category_2         0
category_3         0
category_4         0
category_5         0
category_6         0
category_7         0
category_8         0
category_9         0
category_10        0
unit_1             0
unit_2             0
unit_3             0
dtype: int64
```

✅ Dependent variable = what you're trying to predict (e.g., consumer choice, own brand, demand) - They are the outcomes, the result i am trying to predict. What you're trying to explain.

✅ Independent variables = what influences that outcome (like price, category, date, etc.) - Factors that affects the result.What influences the outcome

Independent Variables (What influences the outcome):

These are the factors that could affect the purchase decision:

prices_(£) → actual product price prices_unit_(£) → price per unit (value-for-money) supermarket → brand/store loyalty category → type of product unit → product format (e.g., per kg, per unit) date → captures price trends/seasonal changes

Independent Variables (What influences the outcome):

These are the factors that could affect the purchase decision:

prices_(£) → actual product price prices_unit_(£) → price per unit (value-for-money) supermarket → brand/store loyalty unit → product format (e.g., per kg, per unit) date → captures price trends/seasonal changes category

Dependent Variable

own_brand

In [89]:
```python
from sklearn.model_selection import train_test_split

# Define target (y) and features (X)
y = df['own_brand']
X = df.drop(['own_brand', 'names', 'date'], axis=1)  # Drop date & names for now

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
```

The first result (accuracy 91.87%) is from the Random Forest model(this can be found down my code).

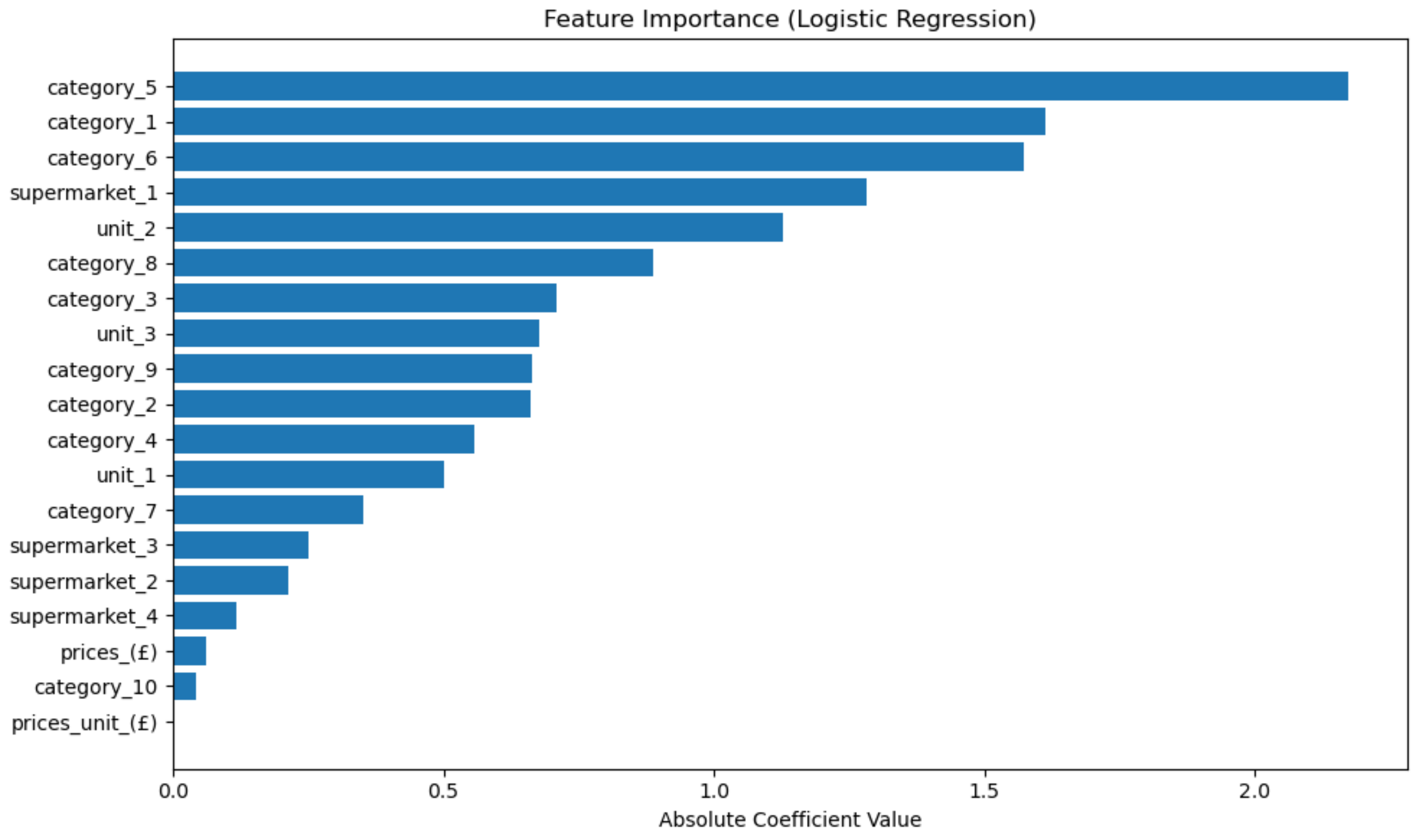
The second result (accuracy 77.30%) is from the Logistic Regression model.

Random Forest is better here because it handles non-linear relationships and complex feature interactions more effectively than Logistic Regression, leading to higher accuracy and better performance in predicting own-brand products.

In [94]:
```python
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# Step 1: Train the model
model = LogisticRegression(max_iter=1000)
model.fit(X_train, y_train)

# Step 2: Predict on test set
y_pred = model.predict(X_test)

# Step 3: Evaluate performance
print("Accuracy:", accuracy_score(y_test, y_pred))
print("\nConfusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))
```

```
Accuracy: 0.7729542057109456

Confusion Matrix:
 [[1332720   93421]
 [ 339294  140414]]

Classification Report:
               precision    recall  f1-score   support

       False       0.80      0.93      0.86   1426141
        True       0.60      0.29      0.39    479708

    accuracy                           0.77   1905849
   macro avg       0.70      0.61      0.63   1905849
weighted avg       0.75      0.77      0.74   1905849
```

In [95]:
```python
# Feature importance from logistic regression
coeffs = pd.DataFrame({
    'Feature': X.columns,
    'Importance': np.abs(model.coef_[0])
}).sort_values(by='Importance', ascending=False)

# Plot
plt.figure(figsize=(10, 6))
plt.barh(coeffs['Feature'], coeffs['Importance'])
plt.gca().invert_yaxis()
plt.title('Feature Importance (Logistic Regression)')
plt.xlabel('Absolute Coefficient Value')
plt.tight_layout()
plt.show()
```



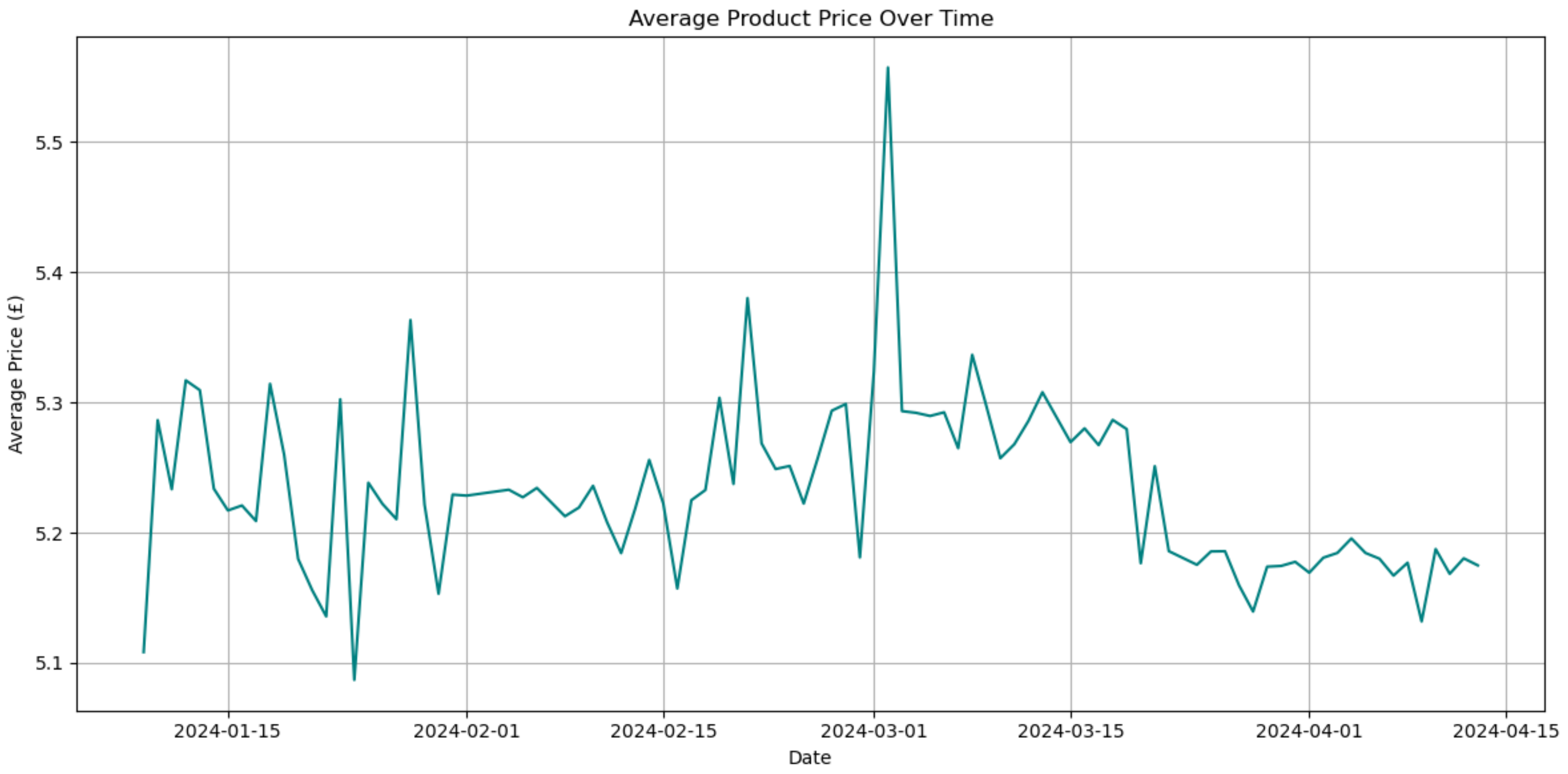
In [91]:
```
from sklearn.tree import DecisionTreeClassifier
model_dt = DecisionTreeClassifier(random_state=42)
model_dt.fit(X_train, y_train)
```

```
Out[91]:
▾ DecisionTreeClassifier
DecisionTreeClassifier(random_state=42)
```

The graph shows that the average product price slightly fluctuated daily but generally declined over time from January to April 2024, with a few noticeable spikes, suggesting occasional short-term price increases.

The x-axis in matplotlib auto-formats labels to avoid clutter, so it only shows a few date labels, not that it only used 5 points. But behind the scenes, all 92 days are plotted!

```
In [107…
# 1. Price Trends Over Time

# Group by date and calculate mean price
daily_avg = df.groupby('date')['prices_(£)'].mean()

# Plot the time series
plt.figure(figsize=(12, 6))
plt.plot(daily_avg, color='teal')
plt.title("Average Product Price Over Time")
plt.xlabel("Date")
plt.ylabel("Average Price (£)")
plt.grid(True)
plt.tight_layout()
plt.show()
```



```
In [118… print(daily_avg.shape)
```

```
(92,)
```

```
In [126… # Step to reconstruct the 'category' column from one-hot encoding

category_columns = [col for col in df.columns if col.startswith('category_')]
df['category'] = df[category_columns].idxmax(axis=1).str.replace('category_', '')
```

This graph shows how the average price of each product category (coded 1–10) changed daily over time.

Interpretation (short): Some categories (like 10 and 6) consistently have higher prices, while others (like 1 and 5) remain lower. The lines show relatively stable trends, but there are occasional price spikes and dips, especially in mid-March, suggesting price volatility varies across categories.

X-axis (Date): This represents the daily timeline from early January to mid-April 2024. Y-axis (Average Price £): This shows the average price per product in each category on each day — ranging from around £2 to over £10. Lines (Category 0–10): Each line represents a product category (like drinks, baby products, etc., though now encoded as numbers).

Stands out with consistently high prices (~£10+).

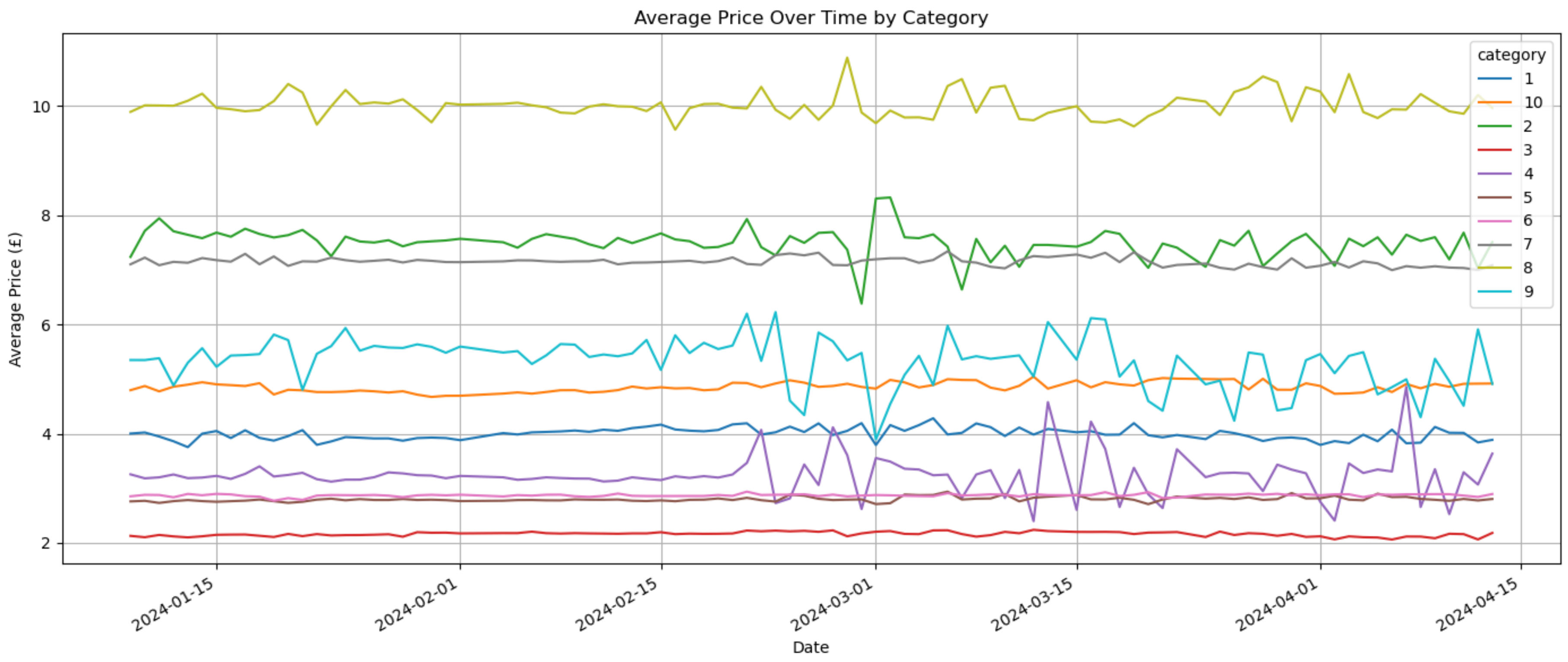
The lines show relatively stable trends, but there are occasional price spikes and dips, especially in mid-March, suggesting price volatility varies across categories.

The colors in the graph represent different product categories and show how their average prices varied over time:

Yellow-green (Category 8): Highest-priced category, very stable around £10–£11. Green (Category 2): Mid-to-high range, steady around £7–£8. Light Blue (Category 9): More price fluctuations, ranges between £5–£6. Orange (Category 10): Stable mid-range prices, around £5. Blue (Category 1): Slight variations, steady between £4–£5. Purple (Category 4): Moderate ups and downs, around £3.5–£4. Brown (Category 5) and Grey (Category 7): Low and stable prices near £3. Light Purple (Category 6): Some jumps, around £2.5–£3.5. Red (Category 3): Lowest-priced category, consistently around £2.

In [142…

```
#2. Price Changes by Category Over Time

category_trends = df.groupby(['date', 'category'])['prices_(£)'].mean().unstack()
category_trends.plot(figsize=(14, 6))
plt.title("Average Price Over Time by Category")
plt.xlabel("Date")
plt.ylabel("Average Price (£)")
plt.grid(True)
plt.tight_layout()
plt.show()
```



This graph compares the average price trends of own-brand (green) vs branded (orange) products over time:

Branded products consistently cost more, averaging around £5.7–£6.0. Own-brand products are cheaper, averaging around £3.6–£4.0. Both show mild fluctuations, but the price gap remains steady, reinforcing that own-brand options are more affordable. A small price spike appears for both types in early March, likely due to temporary market conditions or promotions.

This line plot compares the average daily prices of own-brand (green) and branded (orange) products over time. Branded products consistently maintain a higher price point, around £5.70–£5.90, while own-brand products hover lower, around £3.60–£3.90.

Despite some minor fluctuations, own-brand pricing remains more stable, whereas branded product prices show occasional spikes (notably in early March). This supports the idea that own-brand options offer consistent affordability during price volatility periods.

🟧 Branded Products (Orange Line): Average price consistently higher, around £5.6–£5.9. Minor fluctuations with occasional upward spikes (especially around early March). Indicates price stability, but still more expensive overall.

🟩 Own-Brand Products (Green Line): Prices range from £3.5–£3.9, noticeably cheaper. Shows a few sharper downward dips (e.g., mid-Jan, late March), possibly discount-driven. Generally stable, but slightly more reactive (spikes suggest price adjustments or promo effects).

How This Own-Brand vs Branded Chart Informs It: Own-brand is a proxy for purchasing behavior Since you don't have actual transaction data, you're using whether a product is own-brand (own_brand=True) as a proxy for consumer choice. By examining price trends for own-brand vs branded, you're indirectly observing what pricing patterns consumers respond to.

Price variation affects brand choice The chart shows own-brand products are consistently cheaper, with occasional dips or volatility. If consumers were switching to cheaper alternatives during branded price spikes, it supports the idea that price variation influences consumer behavior.

Evidence of consumer price sensitivity The clearer reaction in own-brand pricing (more frequent small dips or spikes) suggests price sensitivity is higher in this category. That aligns with the idea that during periods of price volatility, consumers are likely to opt for cheaper, own-brand products.

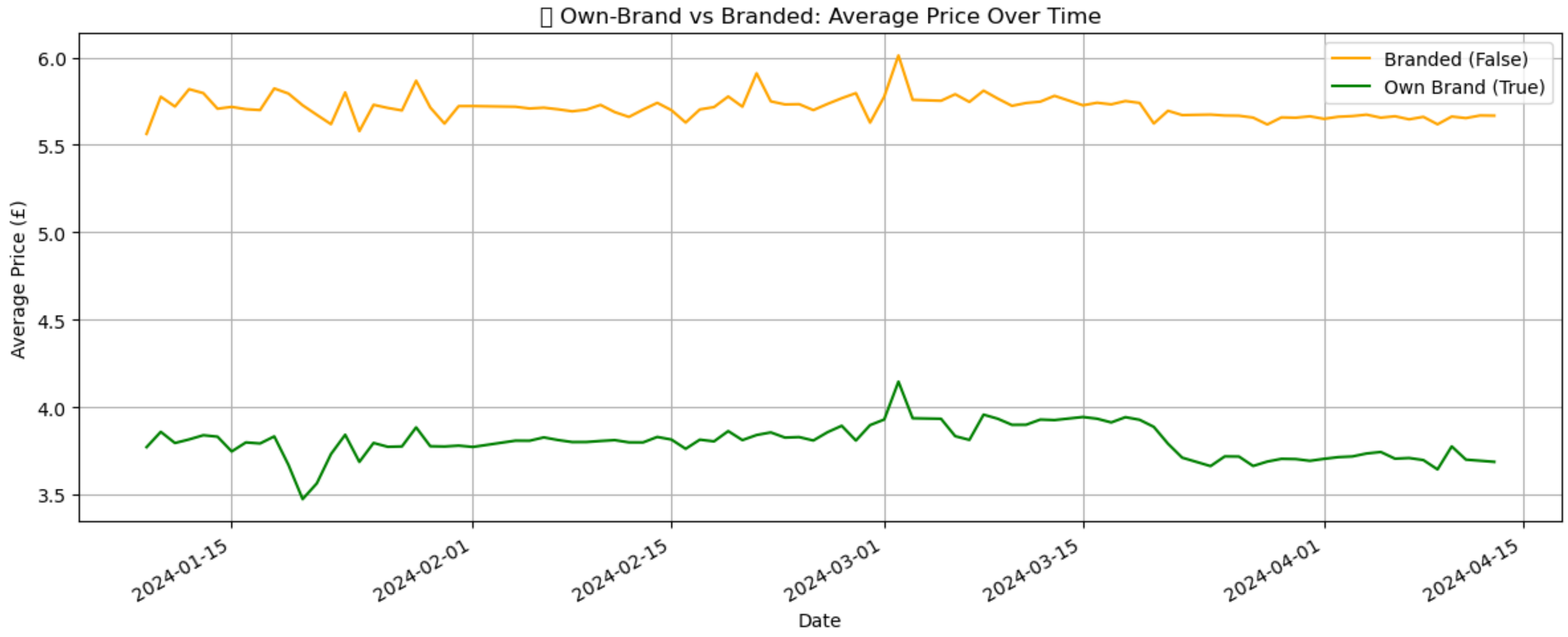
Brand segmentation strategy insights for retailers This pattern tells supermarkets that during volatile pricing periods, strategically discounting own-brand items could boost consumer loyalty and maintain sales — a critical insight for price and promotion planning.

📌 Bottom Line: Your visualization supports the hypothesis that price variations (volatility) do influence consumer choice — and that own-brand products become more attractive when prices rise or fluctuate for branded goods.

In [157…
```python
# 3. Own-Brand vs Branded Pricing Over Time


own_brand_trends = df.groupby(['date', 'own_brand'])['prices_(£)'].mean().unstack()
```

```python
own_brand_trends.plot(figsize=(12, 5), color=['orange', 'green'])
plt.title("🏷️ Own-Brand vs Branded: Average Price Over Time")
plt.xlabel("Date")
plt.ylabel("Average Price (£)")
plt.legend(['Branded (False)', 'Own Brand (True)'])
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
/var/folders/04/6sptnwmx1pd72p7w91rnf6100000gn/T/ipykernel_40701/3739143781.py:12: UserWarning: Glyph 12799
1 (\N{LABEL}) missing from current font.
  plt.tight_layout()
/opt/anaconda3/lib/python3.12/site-packages/IPython/core/pylabtools.py:170: UserWarning: Glyph 127991 (\N{L
ABEL}) missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
```

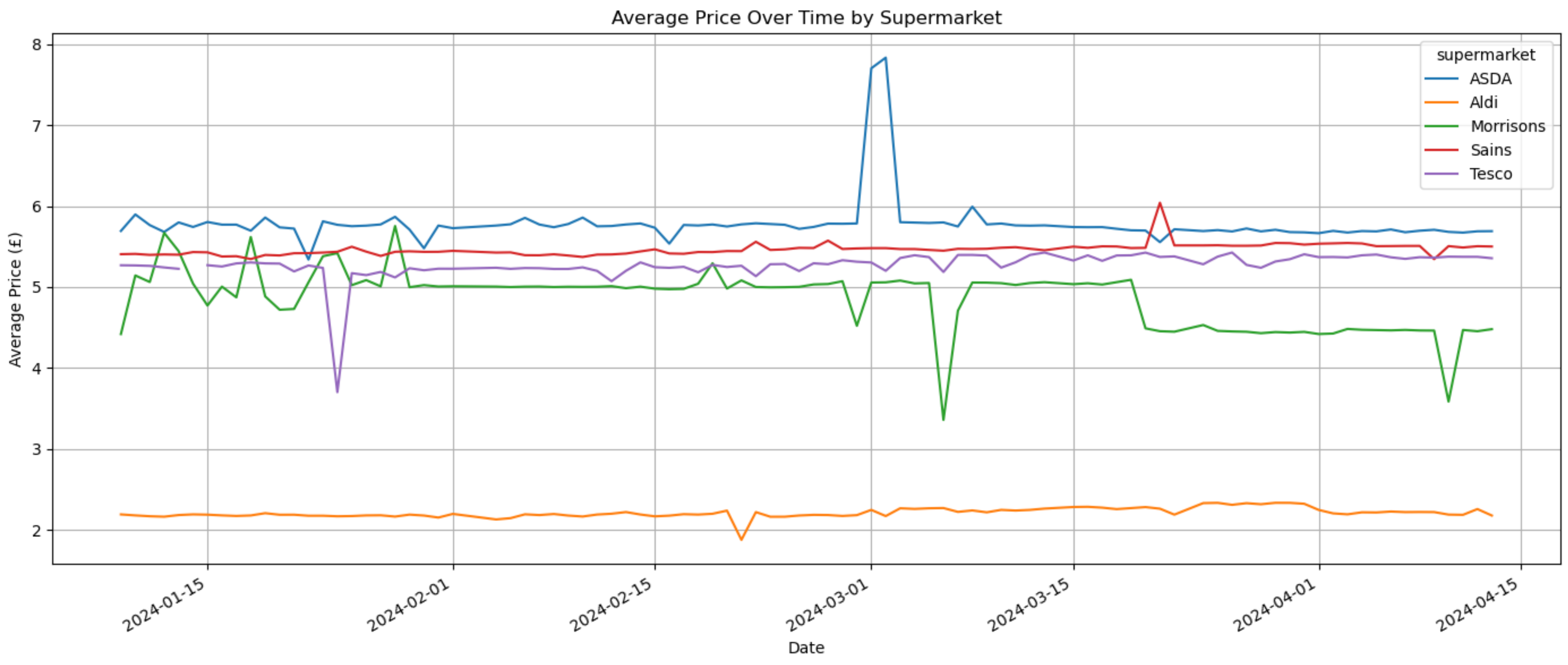


Interpretation:

ASDA (blue) shows the most volatility, with noticeable spikes and drops — especially a sharp peak around early March. Aldi (orange) remains the most stable and lowest-priced, consistently hovering just above £2. Morrisons (green) fluctuates more than others, with prices dipping sharply at multiple points. Sainsbury's (red) and Tesco (purple) have relatively steady pricing, both maintaining mid-range levels close to £5–£5.5.

In [313…

```
#4. Supermarket-Wise Price Comparison Over Time


# Load the original unencoded data
df_original = pd.read_csv("merged_supermarket_data.csv")

# Convert 'date' column to datetime format
df_original['date'] = pd.to_datetime(df_original['date'], format='%Y%m%d', errors='coerce')

# Group by date and supermarket, calculate average price
supermarket_trends = df_original.groupby(['date', 'supermarket'])['prices_(£)'].mean().unstack()

# Plot the trends
supermarket_trends.plot(figsize=(14, 6))
plt.title("Average Price Over Time by Supermarket")
plt.xlabel("Date")
plt.ylabel("Average Price (£)")
plt.grid(True)
plt.tight_layout()
plt.show()
```



```
In [ ]: # Optional: Extract day/month from the date
        df['month'] = df['date'].dt.month
        df['day_of_week'] = df['date'].dt.dayofweek
```

```
In [ ]: print(df.columns)
```

Predict consumer brand choice (own vs branded) — and use the own_brand column as a proxy proxy for consumer behaviour which might give a clue to customer's purchasing behaviour based on purchasing own brand or branded products.

In other words, you’re not predicting how prices change — you're testing if price differences (between products, over time, across supermarkets, etc.) impact brand choice.

Trying to find out if the shoppers will go for the cheaper own-brand or the more expensive branded option? which is a shopping decision influenced by price.

So the model can learn:

When price gaps are large, do people switch to own-brand? Does it depend on product category? Does this behavior vary across supermarkets?

```
In [ ]: y = df['own_brand']  # Proxy for consumer behavior
```

```
In [224…  # Step 1
          X = df[['prices_(£)', 'prices_unit_(£)',
                  'supermarket_1', 'supermarket_2', 'supermarket_3', 'supermarket_4',
                  'category_1', 'category_2', 'category_3', 'category_4', 'category_5',
                  'category_6', 'category_7', 'category_8', 'category_9', 'category_10',
                  'unit_1', 'unit_2', 'unit_3',
                  'month', 'day_of_week']]
```

```
In [ ]: print(X.columns)
```

Interpretation:

The chart shows the top 10 most influential features that impact whether a product is own-brand or not (your proxy for consumer purchasing behavior).

prices_unit_(£) and prices_(£) are by far the most important features. → This means price sensitivity plays the biggest role in predicting own-brand preference. → Consumers tend to prefer own-brand products based on their unit price and overall price.

category_5 and unit_1 also have some influence. → Certain product categories or packaging units may push consumers toward own-brand options.

Supermarket-related and less important category/unit dummy variables (like supermarket_2, category_3, etc.) show minimal influence, but still slightly affect the outcome.

Business Insight:

Price (especially per unit) is the strongest driver of own-brand preference — this supports the idea that budget-conscious consumers lean toward own brands during periods of price volatility.

Let me know if you'd like to turn this into a written paragraph for your report.

Feature Importance Plot

It visualizes how important each feature is in making predictions according to the Random Forest model.

So to clarify:

The model used is Random Forest Classifier. The plot you generated is a Feature Importance Plot from the Random Forest.

```
In [ ]:
# Use X_train.columns to ensure consistency
feature_importance = pd.Series(rf_model.feature_importances_, index=X_train.columns)

# Plot top 10
feature_importance.nlargest(10).plot(kind='barh', figsize=(8, 6), color='skyblue')
plt.title("Top 10 Feature Importance")
plt.xlabel("Importance Score")
plt.tight_layout()
plt.show()
```

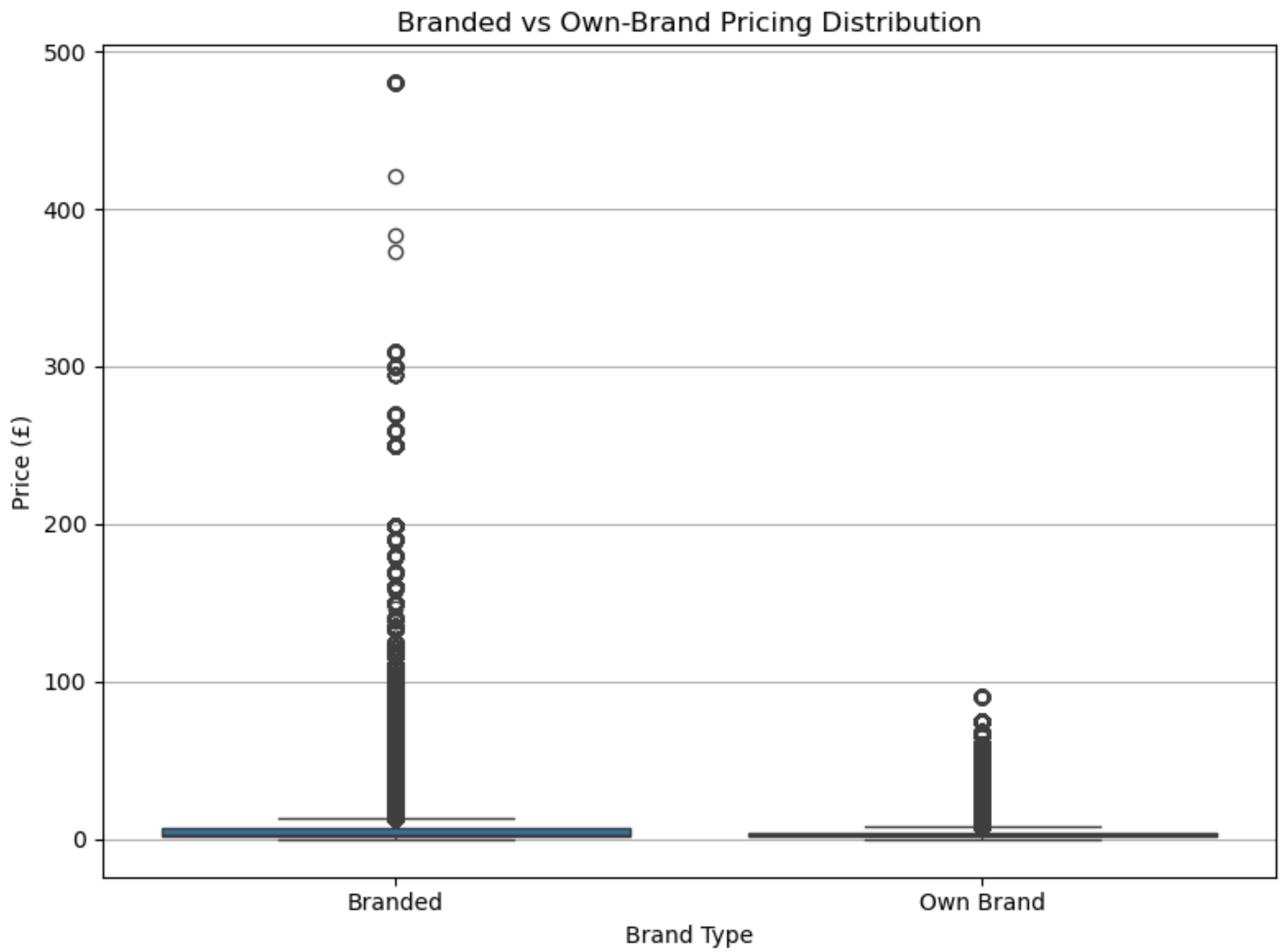
Trained and evaluated Random Forest model, the next step is to analyze and interpret results and draw business insights from them. Here's what to do next:

```
In [246…
from sklearn.metrics import accuracy_score, classification_report

# Predict using the trained Random Forest model
y_pred_rf = rf_model.predict(X_test)

# Print evaluation results
print("Random Forest Accuracy:", accuracy_score(y_test, y_pred_rf))
print("\nClassification Report:\n", classification_report(y_test, y_pred_rf))
```

```
Random Forest Accuracy: 0.9186719409564976

Classification Report:
               precision    recall  f1-score   support

       False       0.93      0.96      0.95   1426141
        True       0.87      0.80      0.83    479708

    accuracy                           0.92   1905849
   macro avg       0.90      0.88      0.89   1905849
weighted avg       0.92      0.92      0.92   1905849
```

In [259…

```
# Map True/False to labels
df['Brand Type'] = df['own_brand'].map({True: 'Own Brand', False: 'Branded'})

# Create the boxplot
plt.figure(figsize=(8, 6))
sns.boxplot(data=df, x='Brand Type', y='prices_(£)')
plt.title('Branded vs Own-Brand Pricing Distribution')
plt.ylabel('Price (£)')
plt.xlabel('Brand Type')
plt.grid(True, axis='y')
plt.tight_layout()
plt.show()
```

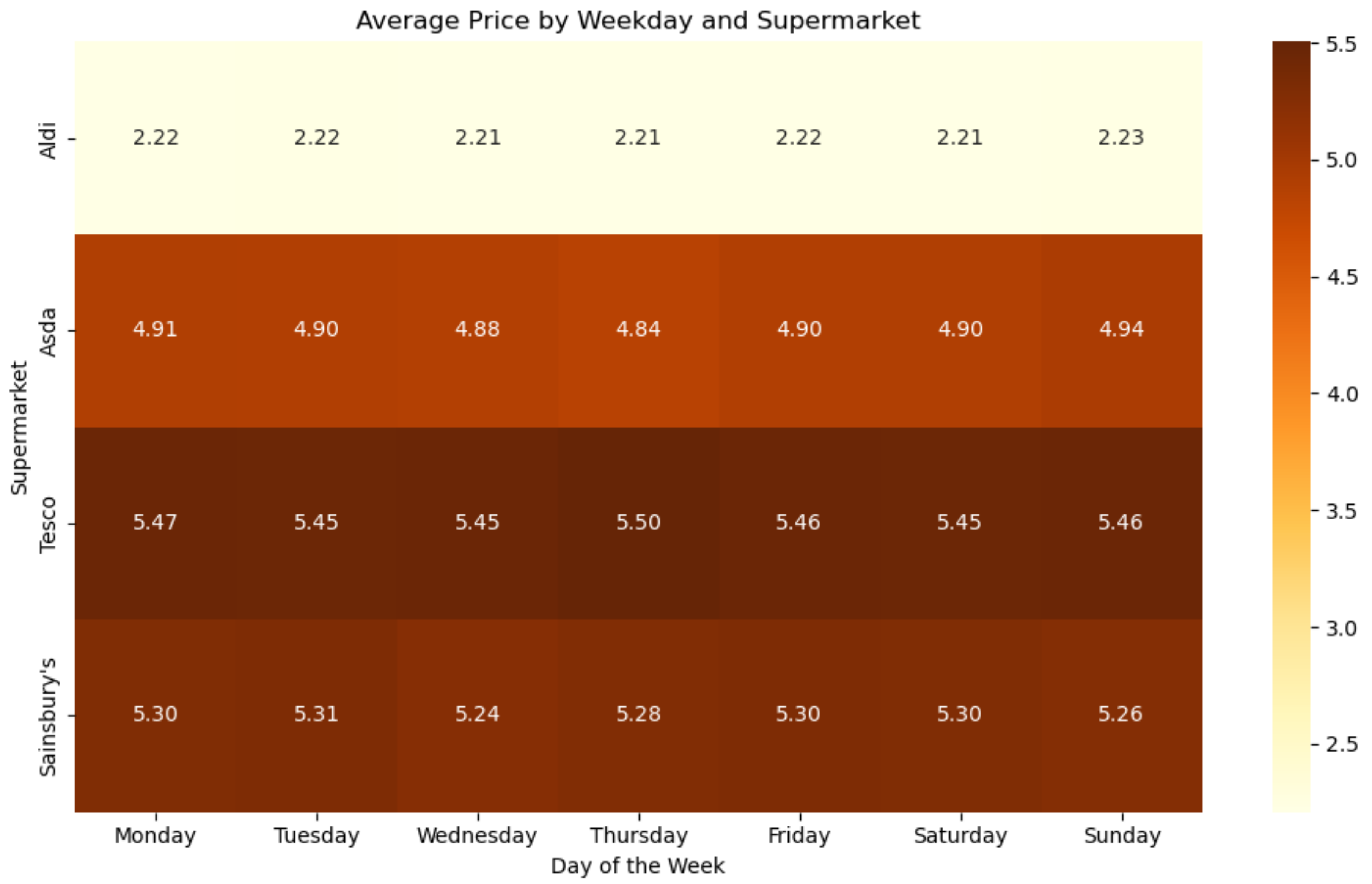


In [301…
```
supermarket_map = {
    'Supermarket 1': 'Aldi',
    'Supermarket 2': 'Asda',
    'Supermarket 3': 'Tesco',
    'Supermarket 4': 'Sainsbury\'s'
}
```

In [302…
```
heatmap_data.rename(index=supermarket_map, inplace=True)
```

In [310…
```
# Step 1: Identify the supermarket for each row
def get_supermarket(row):
    for i in range(1, 5):
        if row[f'supermarket_{i}']:
            return f'Supermarket {i}'
    return 'Unknown'

df['Supermarket'] = df.apply(get_supermarket, axis=1)

# Step 2: Map day_of_week numbers to weekday names
weekday_map = {
    0: 'Monday', 1: 'Tuesday', 2: 'Wednesday',
    3: 'Thursday', 4: 'Friday', 5: 'Saturday', 6: 'Sunday'
}
df['Weekday'] = df['day_of_week'].map(weekday_map)

# Step 3: Group by supermarket and weekday, calculate average price
heatmap_data = df.groupby(['Supermarket', 'Weekday'])['prices_(£)'].mean().unstack()

# Step 4: Reorder weekdays
weekday_order = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
heatmap_data = heatmap_data[weekday_order]

heatmap_data.rename(index=supermarket_map, inplace=True)

heatmap_data = heatmap_data.drop('Unknown', errors='ignore')



# Step 5: Plot heatmap
```

```python
plt.figure(figsize=(10, 6))
sns.heatmap(heatmap_data, cmap='YlOrBr', annot=True, fmt=".2f")
plt.title('Average Price by Weekday and Supermarket')
plt.xlabel('Day of the Week')
plt.ylabel('Supermarket')
plt.tight_layout()
plt.show()
```



Include it in your results section as:

Figure X: Heatmap showing average product price per weekday across UK supermarkets

Mention insights like: “Supermarket 1 maintains the lowest pricing consistency throughout the week, which could be a strategic competitive positioning.”

In [284…

```
print(heatmap_data)
```

```
Weekday           Monday   Tuesday  Wednesday  Thursday    Friday  Saturday  \
Supermarket
Supermarket 1   2.220107  2.221260   2.207604  2.208248  2.219546  2.209349
Supermarket 2   4.910060  4.895185   4.878013  4.835564  4.896243  4.903616
Supermarket 3   5.465683  5.448396   5.453805  5.504918  5.462726  5.450771
Supermarket 4   5.295142  5.305230   5.236643  5.283854  5.303701  5.296157
Unknown         5.712291  5.731758   5.759162  5.717662  5.787757  5.864365

Weekday           Sunday
Supermarket
Supermarket 1   2.225230
Supermarket 2   4.941741
Supermarket 3   5.464002
Supermarket 4   5.258722
Unknown         5.745003
```

In [287…

```
print(df[['Supermarket', 'Weekday', 'prices_(£)']].head(10))
```

```
     Supermarket   Weekday  prices_(£)
0  Supermarket 1  Saturday        3.09
1  Supermarket 1  Saturday        3.09
2  Supermarket 1  Saturday        3.59
3  Supermarket 1  Saturday        4.79
4  Supermarket 1  Saturday        4.79
5  Supermarket 1  Saturday        0.85
6  Supermarket 1  Saturday        2.79
7  Supermarket 1  Saturday        2.89
8  Supermarket 1  Saturday        2.25
9  Supermarket 1  Saturday        3.59
```

conlusion

real transaction-level data (e.g., from loyalty cards or receipts) would show actual consumer purchasing habits, making the analysis more accurate and insightful.

```
In [ ]:
```