WiseAnalytics Assessment

In this notebook, I will be exploring an online retail dataset in order to get insights from it.

In [1]:
```
# import all the required modules
import pandas as pd
import numpy as np

import seaborn as sns
import matplotlib.pyplot as plt

from mlxtend.frequent_patterns import apriori, fpgrowth
from mlxtend.frequent_patterns import association_rules

from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler, PolynomialFeatures
from sklearn.model_selection import train_test_split, GridSearchCV, cross_val_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.linear_model import LinearRegression, Ridge
from sklearn.svm import LinearSVR
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.pipeline import make_pipeline
from sklearn.decomposition import PCA

from scipy.stats import randint
from xgboost import XGBClassifier
from scipy.sparse import csr_matrix
import pickle
```

Part 1: Data Exploration and Preprocessing

Task 1: Exploratory Data Analysis

In [2]:
```
#Load the dataset
retail_df = pd.read_excel("../Online Retail.xlsx")
```

In [3]:
```
#shape of our dataset
print("The shape of our dataset is: ", retail_df.shape)
```

```
The shape of our dataset is:  (541909, 8)
```

We can deduce that we have 541909 rows and 8 columns from the above step.

In [4]:
```
#check the head of the dataset
retail_df.head(10)
```

Out[4]:

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Cou
0	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	2010-12-01 08:26:00	2.55	17850.0	Un Kingc
1	536365	71053	WHITE METAL LANTERN	6	2010-12-01 08:26:00	3.39	17850.0	Un Kingc
2	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	2010-12-01 08:26:00	2.75	17850.0	Un Kingc
3	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	2010-12-01 08:26:00	3.39	17850.0	Un Kingc
4	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	6	2010-12-01 08:26:00	3.39	17850.0	Un Kingc
5	536365	22752	SET 7 BABUSHKA NESTING BOXES	2	2010-12-01 08:26:00	7.65	17850.0	Un Kingc
6	536365	21730	GLASS STAR FROSTED T-LIGHT HOLDER	6	2010-12-01 08:26:00	4.25	17850.0	Un Kingc
7	536366	22633	HAND WARMER UNION JACK	6	2010-12-01 08:28:00	1.85	17850.0	Un Kingc
8	536366	22632	HAND WARMER RED POLKA DOT	6	2010-12-01 08:28:00	1.85	17850.0	Un Kingc
9	536367	84879	ASSORTED COLOUR BIRD ORNAMENT	32	2010-12-01 08:34:00	1.69	13047.0	Un Kingc

Let's verify what our dataset looks like by getting more information about rows and columns.

In [5]:
```
retail_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 541909 entries, 0 to 541908
Data columns (total 8 columns):
 #   Column       Non-Null Count   Dtype         
---  ------       --------------   -----         
 0   InvoiceNo    541909 non-null  object        
 1   StockCode    541909 non-null  object        
 2   Description  540455 non-null  object        
 3   Quantity     541909 non-null  int64         
 4   InvoiceDate  541909 non-null  datetime64[ns]
 5   UnitPrice    541909 non-null  float64       
 6   CustomerID   406829 non-null  float64       
 7   Country      541909 non-null  object        
dtypes: datetime64[ns](1), float64(2), int64(1), object(4)
memory usage: 33.1+ MB
```

As deduced earlier our dataset consists of **541909** rows and **8** columns.

When going through the columns, we notice that we have some missing values in the **CustomerID** column, as it only has **406829** values. Similarly, **Description** has missing values too. So, that should be considered when applying exploration to our dataset.

In [6]:
```
#exploring the unique values of each attribute
print("Number of transactions: ", retail_df['InvoiceNo'].nunique())
print("Number of products: ",retail_df['StockCode'].nunique())
print("Number of customers:", retail_df['CustomerID'].nunique() )
print("Percentage of customers NA: ", round(retail_df['CustomerID'].isnull().sum()
print('Number of countries: ',retail_df['Country'].nunique())
```

```
Number of transactions:  25900
Number of products:  4070
Number of customers: 4372
Percentage of customers NA:  25.0 %
Number of countries:  38
```

We can identify that 25% of the customers do not have a CustomerID. This could impact our analysis.

In [7]:
```
retail_df.describe()
```

Out[7]:

	Quantity	InvoiceDate	UnitPrice	CustomerID
count	541909.000000	541909	541909.000000	406829.000000
mean	9.552250	2011-07-04 13:34:57.156386048	4.611114	15287.690570
min	-80995.000000	2010-12-01 08:26:00	-11062.060000	12346.000000
25%	1.000000	2011-03-28 11:34:00	1.250000	13953.000000
50%	3.000000	2011-07-19 17:17:00	2.080000	15152.000000
75%	10.000000	2011-10-19 11:27:00	4.130000	16791.000000
max	80995.000000	2011-12-09 12:50:00	38970.000000	18287.000000
std	218.081158	NaN	96.759853	1713.600303

We can see that the minimum values are in negatives for quantities. Based on the dataset description we were provided these could be the cancelled orders.

Dataset description:

InvoiceNo: Invoice number. ***Nominal***, a 6-digit integral number uniquely assigned to each transaction. **If this code starts with letter 'c', it indicates a cancellation.**
StockCode: Product (item) code. ***Nominal***, a 5-digit integral number uniquely assigned to each distinct product.
Description: Product (item) name. ***Nominal***.
Quantity: The quantities of each product (item) per transaction. ***Numeric***.
InvoiceDate: Invoice Date and time. ***Numeric***, the day and time when each transaction was generated.
UnitPrice: Unit price. ***Numeric***, Product price per unit in sterling.
CustomerID: Customer number. ***Nominal***, a 5-digit integral number uniquely assigned to each customer.
Country: Country name. ***Nominal***, the name of the country where each customer resides.

Let's verify our hypothesis of these negative values being cancelled orders and if the value of -80995 is a cancelled order

In [8]:
```
#get cancelled transactions
cancelled_orders = retail_df[retail_df['InvoiceNo'].astype(str).str.startswith('C')
cancelled_orders.head()
```

Out[8]:

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Co
141	C536379	D	Discount	-1	2010-12-01 09:41:00	27.50	14527.0	Kir
154	C536383	35004C	SET OF 3 COLOURED FLYING DUCKS	-1	2010-12-01 09:49:00	4.65	15311.0	Kir
235	C536391	22556	PLASTERS IN TIN CIRCUS PARADE	-12	2010-12-01 10:24:00	1.65	17548.0	Kir
236	C536391	21984	PACK OF 12 PINK PAISLEY TISSUES	-24	2010-12-01 10:24:00	0.29	17548.0	Kir
237	C536391	21983	PACK OF 12 BLUE PAISLEY TISSUES	-24	2010-12-01 10:24:00	0.29	17548.0	Kir

In [9]:
```
#search for transaction where quantity == -80995
cancelled_orders[cancelled_orders['Quantity']==-80995]
```

Out[9]:

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID
540422	C581484	23843	PAPER CRAFT , LITTLE BIRDIE	-80995	2011-12-09 09:27:00	2.08	16446.0

As expected, these negatives values of the cancelled orders and -80995 does support our hypothesis in this regards.

Let's make sure if there are no positive values in quantities among these cancelled orders

In [10]:
```
cancelled_orders[cancelled_orders['Quantity']>0]
```

Out[10]:

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Count

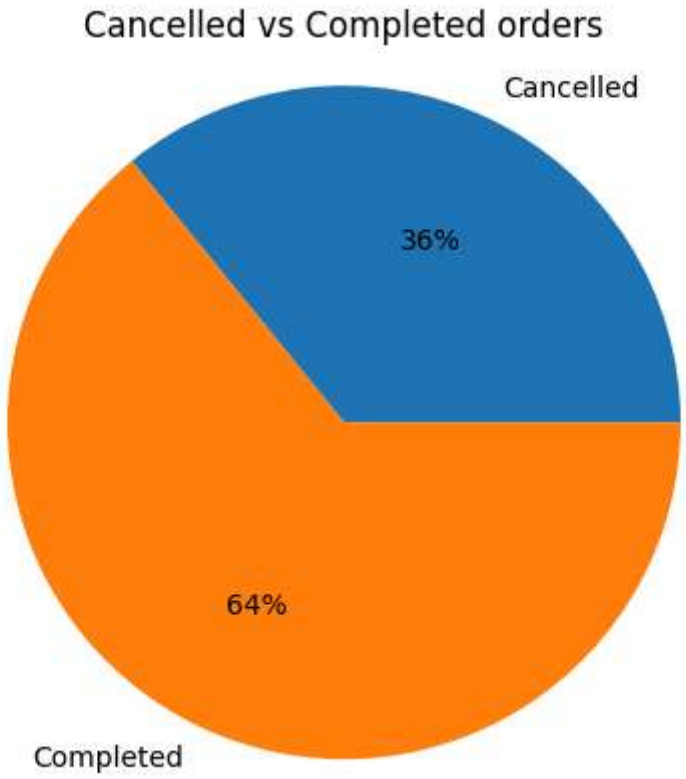
The above steps confirm it that all the negative quantities belong to cancelled orders.

Let's check how many cancelled orders we have and what percentage they entail to in our dataset?

In [11]:
```
#check how many rows our dataframe of cancelled orders contain
print("We have ",len(cancelled_orders), " cancelled orders.")
#percentage of cancelled orders in total orders
total_orders = retail_df['InvoiceNo'].nunique()
cancelled_number = len(cancelled_orders)
completed_orders = total_orders - cancelled_number
print("we have ", completed_orders, " completed orders")
print('Percentage of orders canceled: {}/{} ({:.0f}%) '.format(cancelled_number, to
```

```
We have  9288  cancelled orders.
we have  16612  completed orders
Percentage of orders canceled: 9288/25900 (36%)
```

In [12]:
```
labels = ['Cancelled','Completed']
sizes = [cancelled_number,completed_orders]

fig, ax = plt.subplots()
ax.pie(sizes, labels = labels, autopct ='%1.0f%%')
ax.axis('equal')
ax.set_title('Cancelled vs Completed orders')
plt.show()
```



A significant 36% of the overall orders have been cancelled. Doing a deep dive into these orders can help us reduce future cancellations.

Let's look into customers purchase behavior which can help us understand our patterns and trends

In [13]:
```
# get unique InvoiceNo number per customer
groupby_customers = pd.DataFrame(retail_df.groupby('CustomerID')['InvoiceNo'].nuniq
```

```
groupby_customers.head()
```

Out[13]:

CustomerID	InvoiceNo
12346.0	2
12347.0	7
12348.0	4
12349.0	1
12350.0	1

In [14]:
```
groupby_customers.describe()
```

Out[14]:

	InvoiceNo
count	4372.000000
mean	5.075480
std	9.338754
min	1.000000
25%	1.000000
50%	3.000000
75%	5.000000
max	248.000000

Invoice-wise Order Analysis

1. The mean value of 5.08 suggests that, on average, each customer has placed around 5 orders.
2. The std (standard deviation) of 9.34 is quite high, meaning there is significant variation in the number of orders placed by different customers. Some customers place very few orders, while others place a lot.
3. 50% of customers (median) placed 3 or fewer orders. 25% of customers placed only 1 order (first quartile, Q1 = 1). 75% of customers placed 5 or fewer orders (third quartile, Q3 = 5). This shows that a small number of customers drive a large number of orders, while most customers place only a few orders.
4. The max value of 248 suggests that the most active customer has placed 248 orders—a significantly higher number than the median (3 orders). This indicates that a few high-frequency buyers contribute disproportionately to total sales.

In [15]:
```
groupby_invoice = pd.DataFrame(retail_df.groupby('InvoiceNo')['StockCode'].nunique(
groupby_invoice.columns=['productsNumber']
```

```
groupby_invoice.head()
```

Out[15]:

InvoiceNo	productsNumber
536365	7
536366	2
536367	12
536368	4
536369	1

In [16]:
```
groupby_invoice.describe()
```

Out[16]:

	productsNumber
count	25900.000000
mean	20.510618
std	42.500488
min	1.000000
25%	2.000000
50%	10.000000
75%	23.000000
max	1110.000000

Order Size Analysis

1. The mean value of 20.51 suggests that, on average, each invoice contains around 20 products.
2. The std (standard deviation) of 42.50 is very high, indicating that there is a large variation in the number of products per order. Some orders contain just one product, while others have hundreds or even over a thousand.
3. 50% (median) of invoices contain 10 or fewer products. 25% (first quartile, Q1) contain only 2 products. 75% (third quartile, Q3) contain 23 or fewer products. This means that the majority of orders are relatively small, with a few very large orders driving up the average.
4. The max value of 1,110 suggests that the largest invoice contained 1,110 products in a single transaction. This indicates that bulk purchases exist in the dataset, likely from business customers or wholesalers.

In [17]:
```
retail_df.groupby(['InvoiceNo','CustomerID'])['StockCode'].nunique().describe()
```

Out[17]:
```
count    22190.000000
mean        17.876566
std         22.872614
min          1.000000
25%          3.000000
50%         12.000000
75%         24.000000
max        541.000000
Name: StockCode, dtype: float64
```

Unique Product Diversity per Order

1. The mean value of 17.88 suggests that, on average, each invoice contains about 18 unique products (different StockCode values). This means that most customers buy a variety of different products per order, rather than just a few repeated ones.
2. The std (standard deviation) of 22.87 is quite high, meaning that some orders contain only a few products, while others include a large number of different products. This suggests a diverse customer base, with some making small, specific purchases and others buying a wide range of items.
3. 25% (Q1) of invoices contain 3 or fewer unique products. 50% (median) of invoices contain 12 unique products. 75% (Q3) of invoices contain 24 or fewer unique products. This indicates that while the median number of unique products is 12 per invoice, a significant portion of invoices contain only a few unique items.
4. The max value of 541 shows that the largest invoice contained 541 unique products—a huge number compared to the median (12). This suggests that some bulk buyers or resellers make large purchases, significantly impacting the dataset.

In [18]:
```
temp_df =retail_df.groupby(['InvoiceNo','CustomerID'],as_index=False)['InvoiceDate'
transaction_df = temp_df.rename(columns = {'InvoiceDate':'Number of products'})
transaction_df.head()
```

Out[18]:

	InvoiceNo	CustomerID	Number of products
0	536365	17850.0	7
1	536366	17850.0	2
2	536367	13047.0	12
3	536368	13047.0	4
4	536369	13047.0	1

In [19]:
```
transaction_df.describe()
```

Out[19]:

	CustomerID	Number of products
count	22190.000000	22190.000000
mean	15238.498738	18.333889
std	1733.149624	23.892111
min	12346.000000	1.000000
25%	13755.000000	3.000000
50%	15136.000000	12.000000
75%	16746.000000	24.000000
max	18287.000000	542.000000

Transaction-wise Analysis

1. The mean value of 18.33 suggests that, on average, each transaction contains 18 products. This aligns with previous findings that most invoices contain a moderate number of items, with a few large orders skewing the average.
2. The std (standard deviation) of 23.89 indicates that there is a wide spread in the number of products per transaction. This suggests that some customers purchase only a few items, while others place much larger orders.
3. 25% (Q1) of transactions contain only 3 or fewer products. 50% (median) of transactions contain 12 products. 75% (Q3) of transactions contain 24 or fewer products. This confirms that while the median transaction size is 12 items, a significant number of transactions involve only a few products.
4. The max value of 542 shows that the largest transaction contained 542 products—a huge order compared to the median (12).This suggests the presence of bulk buyers or wholesalers, as seen in previous analyses.

Business Insights

1. **Invoice-wise Order Analysis:** Most customers place small orders: 50% of customers placed 3 or fewer orders. High-value customers exist: Some customers placed up to 248 orders, meaning a few loyal customers contribute significantly to total sales. Retention strategy needed: Since 25% of customers ordered only once, businesses should focus on converting them into repeat buyers using loyalty programs and targeted promotions.
2. **Order Size Analysis:** Most invoices contain relatively few products: 50% of orders contain 10 or fewer products. Bulk buyers exist: Some invoices contain hundreds of products, suggesting wholesalers or business customers. Opportunity for cross-selling: Since many orders are small, encouraging customers to buy complementary products can increase revenue.

3. **Unique Product Diversity per Order:** Customers buy a variety of products: The median number of unique products per order is 12, meaning customers typically purchase multiple different items. Wholesale buyers make large purchases: Some invoices contain up to 541 unique products, showing the presence of bulk buyers. Segmented marketing needed: Business should differentiate marketing efforts for casual shoppers vs. bulk buyers.
4. **Transaction-wise Analysis:** Most transactions contain small to moderate product counts: 50% of transactions include 12 or fewer products. High variation in transaction sizes: Some transactions include over 500 products, indicating business customers. Upselling & bundling potential: Encouraging customers to add more items per transaction through bundle deals can increase revenue.

Customers by Country

In [20]:
```
retail_df['total_cost'] = retail_df['Quantity'] * retail_df['UnitPrice']
```

In [21]:
```
retail_df.head()
```

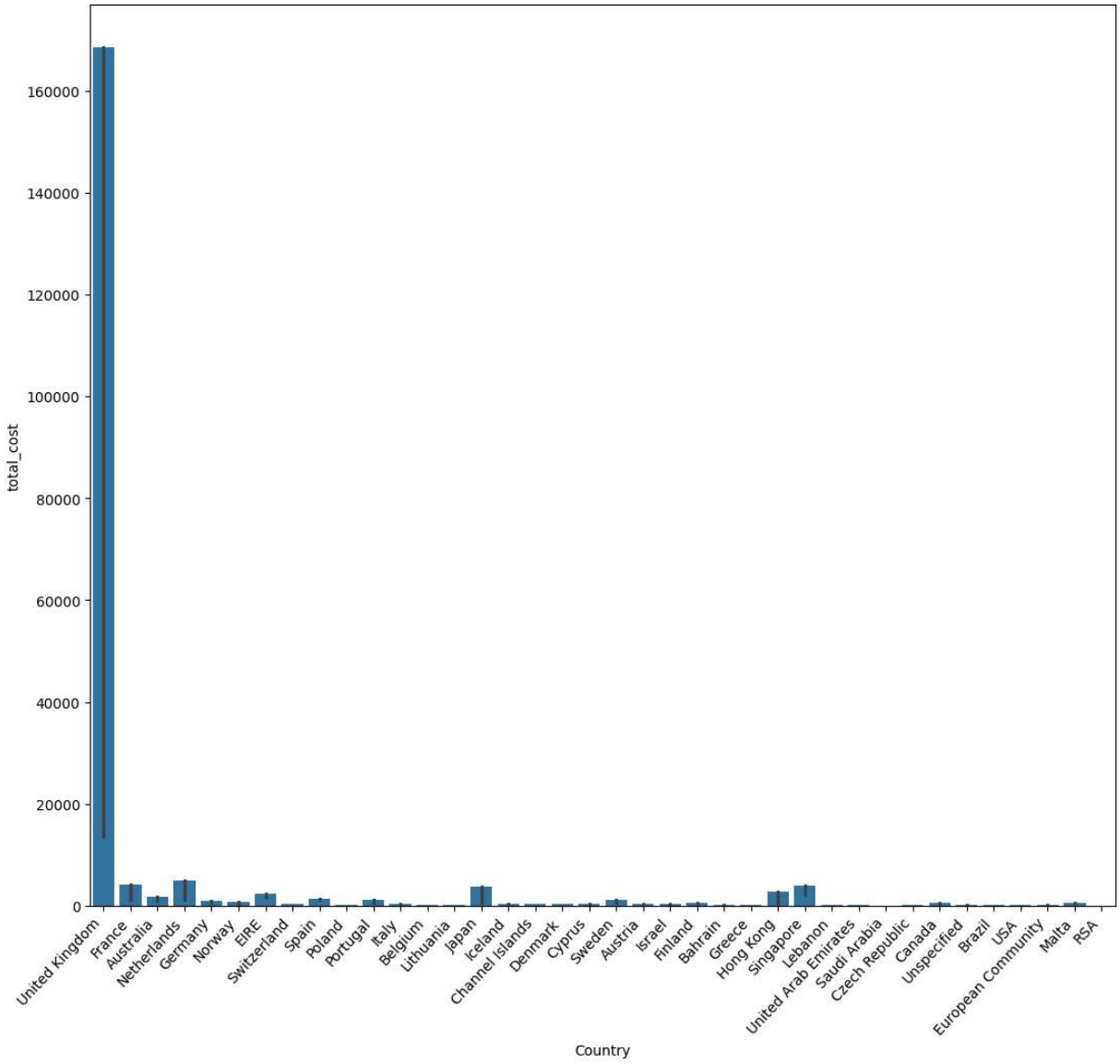
Out[21]:

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Cou
0	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	2010-12-01 08:26:00	2.55	17850.0	Un Kingc
1	536365	71053	WHITE METAL LANTERN	6	2010-12-01 08:26:00	3.39	17850.0	Un Kingc
2	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	2010-12-01 08:26:00	2.75	17850.0	Un Kingc
3	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	2010-12-01 08:26:00	3.39	17850.0	Un Kingc
4	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	6	2010-12-01 08:26:00	3.39	17850.0	Un Kingc

In [22]:
```
import seaborn as sns
import matplotlib.pyplot as plt
```

```
%matplotlib inline
fig, ax = plt.subplots()
fig.set_size_inches(13, 11.5)
ax=sns.barplot(x='Country', y='total_cost',data=retail_df,estimator=max,ax=ax)
ax.set_xticklabels(ax.get_xticklabels(), rotation=47, ha="right")
plt.show()
```

```
C:\Users\abhil\AppData\Local\Temp\ipykernel_7512\3222805382.py:7: UserWarning: set_t
icklabels() should only be used with a fixed number of ticks, i.e. after set_ticks()
or using a FixedLocator.
  ax.set_xticklabels(ax.get_xticklabels(), rotation=47, ha="right")
```



In [23]:
```
retail_df.groupby('Country').sum(numeric_only=True).sort_values(by='total_cost', as
```

Out[23]:

Country	Quantity	UnitPrice	CustomerID	total_cost
United Kingdom	4263829	2245715.474	5.626433e+09	8187806.364
Netherlands	200128	6492.550	3.419054e+07	284661.540
EIRE	142637	48447.190	1.103917e+08	263276.820
Germany	117448	37666.000	1.200751e+08	221698.210
France	110480	43031.990	1.076489e+08	197403.900
Australia	83653	4054.750	1.569300e+07	137077.270
Switzerland	30325	6813.690	2.377592e+07	56385.350
Spain	26824	12633.450	3.268929e+07	54774.580
Belgium	23152	7540.130	2.571829e+07	40910.960
Sweden	35637	1806.830	6.790083e+06	36595.910
Japan	25218	814.860	4.567292e+06	35340.620
Norway	19247	6529.060	1.350765e+07	35163.460
Portugal	16180	13037.540	1.886480e+07	29367.020
Finland	10666	3786.850	8.699324e+06	22326.740
Channel Islands	9479	3738.550	1.128522e+07	20086.290
Denmark	8188	1266.950	4.876734e+06	18768.140
Italy	7999	3879.390	1.015666e+07	16890.510
Cyprus	6317	3920.070	7.715880e+06	12946.290
Austria	4827	1701.520	5.021102e+06	10154.320
Hong Kong	4769	12241.500	0.000000e+00	10117.040
Singapore	5234	25108.890	2.918376e+06	9120.390
Israel	4353	1079.040	3.164467e+06	7907.820
Poland	3653	1422.270	4.341972e+06	7213.140
Unspecified	3300	1204.010	3.348046e+06	4749.790
Greece	1556	713.290	2.008584e+06	4710.520
Iceland	2458	481.210	2.247154e+06	4310.000
Canada	2763	910.580	2.615483e+06	3666.380
Malta	944	666.010	2.158496e+06	2505.470
United Arab Emirates	982	229.890	1.018952e+06	1902.280

Country	Quantity	UnitPrice	CustomerID	total_cost
USA	1034	644.980	3.672086e+06	1730.920
Lebanon	386	242.440	5.743800e+05	1693.880
Lithuania	652	99.440	5.366200e+05	1661.060
European Community	497	294.050	9.215880e+05	1291.750
Brazil	356	142.600	4.086080e+05	1143.600
RSA	352	248.100	7.218680e+05	1002.310
Czech Republic	592	88.150	3.834300e+05	707.720
Bahrain	260	86.570	2.100270e+05	548.400
Saudi Arabia	75	24.110	1.256500e+05	131.170

We can identify that majority of the orders were placed from the United Kingdom.

In [24]:
```
retail_uk = retail_df[retail_df['Country']=='United Kingdom']
retail_uk.describe()
```

Out[24]:

	Quantity	InvoiceDate	UnitPrice	CustomerID	total_cost
count	495478.000000	495478	495478.000000	361878.000000	495478.000000
mean	8.605486	2011-07-04 05:01:41.098131456	4.532422	15547.871368	16.525065
min	-80995.000000	2010-12-01 08:26:00	-11062.060000	12346.000000	-168469.600000
25%	1.000000	2011-03-27 12:06:00	1.250000	14194.000000	3.290000
50%	3.000000	2011-07-19 11:47:00	2.100000	15514.000000	8.290000
75%	10.000000	2011-10-20 10:41:00	4.130000	16931.000000	16.630000
max	80995.000000	2011-12-09 12:49:00	38970.000000	18287.000000	168469.600000
std	227.588756	NaN	99.315438	1594.402590	394.839116

In [25]:
```
print(retail_df[retail_df['Country']=='United Kingdom']['CustomerID'].nunique(), "u
print(retail_uk['InvoiceNo'].nunique(),"unique orders were placed from UK")
print(retail_uk['StockCode'].nunique(),"unique products were bought")
print(round(retail_uk['CustomerID'].isnull().sum() * 100 / len(retail_uk),2),"%","o
```

```
3950 unique customers from UK
23494 unique orders were placed from UK
4065 unique products were bought
26.96 % of anonymous/guest customers
```

In [26]:
```
retail_uk = retail_df[retail_df['Country']=='United Kingdom']
retail_uk.describe()
```

Out[26]:

	Quantity	InvoiceDate	UnitPrice	CustomerID	total_cost
count	495478.000000	495478	495478.000000	361878.000000	495478.000000
mean	8.605486	2011-07-04 05:01:41.098131456	4.532422	15547.871368	16.525065
min	-80995.000000	2010-12-01 08:26:00	-11062.060000	12346.000000	-168469.600000
25%	1.000000	2011-03-27 12:06:00	1.250000	14194.000000	3.290000
50%	3.000000	2011-07-19 11:47:00	2.100000	15514.000000	8.290000
75%	10.000000	2011-10-20 10:41:00	4.130000	16931.000000	16.630000
max	80995.000000	2011-12-09 12:49:00	38970.000000	18287.000000	168469.600000
std	227.588756	NaN	99.315438	1594.402590	394.839116

Business Insights from UK Transaction Data

1. The UK data contains 495,478 transactions from customers in the United Kingdom. Out of these, 361,878 transactions have an associated CustomerID, suggesting that some purchases were made by anonymous or guest buyers.
2. Average purchase quantity: 8.61 items per transaction, meaning customers typically buy in small batches.
3. High variability: The standard deviation (227.59) is very high, indicating a wide range of purchase quantities.
4. Pricing & Revenue (Unit Price & Total Cost): Average unit price: £4.53, but high variability (std = 99.32) suggests a mix of cheap and high-end products.

In [27]:
```
groupedProduct = retail_uk.groupby('StockCode',as_index= False)['Quantity'].sum().s
groupedProduct.head(10)
#check how to show product description instead of StockCode
```

Out[27]:

	StockCode	Quantity
1068	22197	52928
2620	84077	48326
3655	85099B	43167
3666	85123A	36706
2733	84879	33519
1451	22616	25307
375	21212	24702
1049	22178	23242
39	17003	22801
887	21977	20288

In [28]:
```
invoice_quantity= retail_uk.groupby('InvoiceNo', as_index=False)['Quantity'].sum().
invoice_quantity.head()
```

Out[28]:

	InvoiceNo	Quantity
20090	581483	80995
2136	541431	74215
17136	574941	14149
17765	576365	13956
13770	567423	12572

Since UK contains most number of orders in the data and since it might skew the data a bit. Let's look at all the others countries than UK

In [29]:
```
retail_non_uk = retail_df[retail_df['Country']!='United Kingdom']
retail_non_uk.describe()
```

Out[29]:

	Quantity	InvoiceDate	UnitPrice	CustomerID	total_cost
count	46431.000000	46431	46431.000000	44951.000000	46431.000000
mean	19.655424	2011-07-08 08:52:10.383149056	5.450852	13193.105203	33.596984
min	-624.000000	2010-12-01 08:45:00	0.000000	12347.000000	-8322.120000
25%	4.000000	2011-04-07 09:28:00	1.250000	12484.000000	12.500000
50%	10.000000	2011-07-28 14:08:00	1.950000	12658.000000	17.400000
75%	16.000000	2011-10-12 09:50:00	3.750000	14156.000000	30.000000
max	2400.000000	2011-12-09 12:50:00	4161.060000	17844.000000	4992.000000
std	47.233708	NaN	63.360527	1085.609495	104.425951

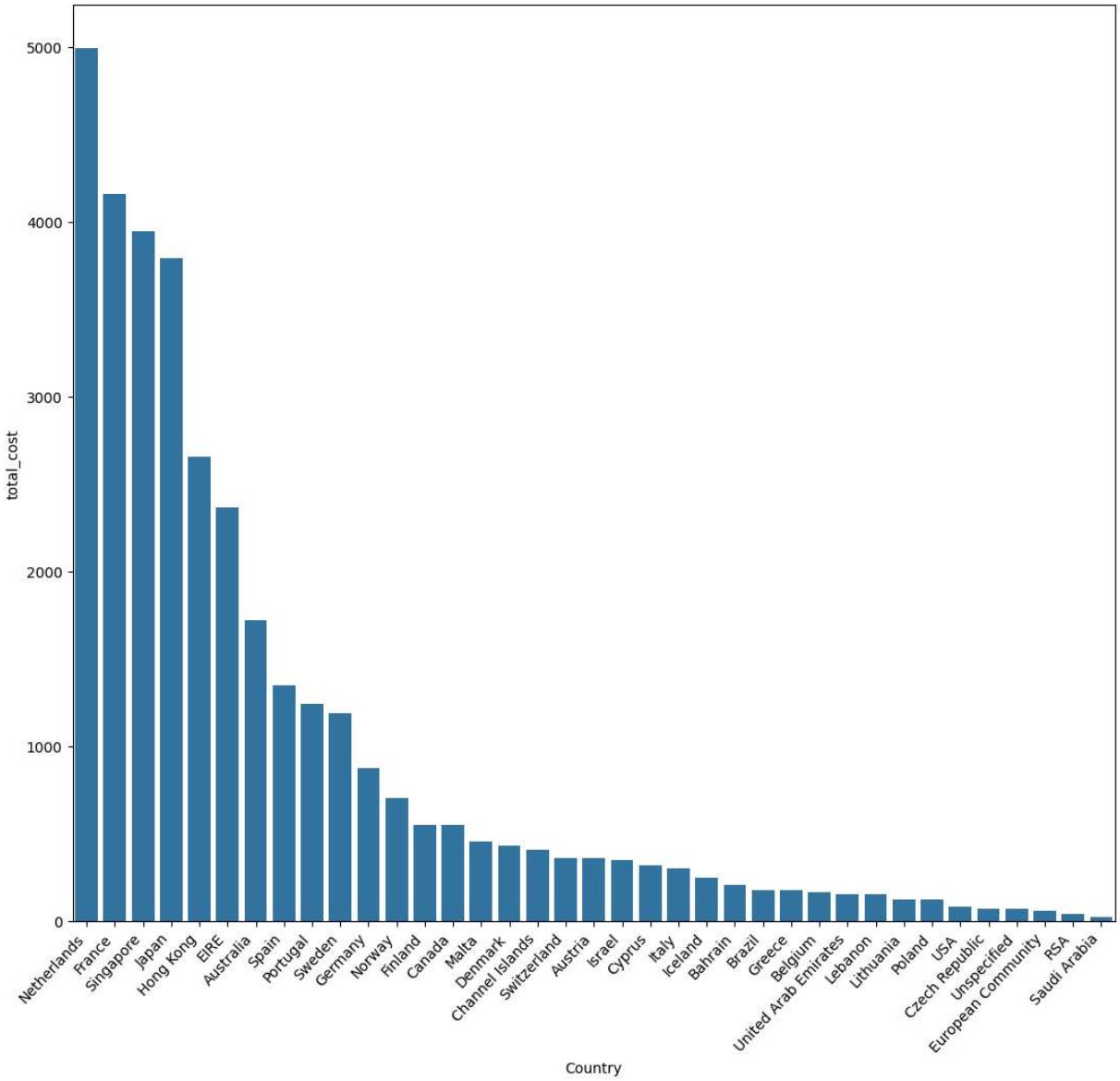
Business Insights from Non-UK Transactions

1. International Sales Volume: The dataset contains 46,431 transactions from non-UK customers, compared to 495,478 UK transactions. This means only ~8.6% of all transactions come from outside the UK.
2. Quantity Insights: Average purchase quantity: 19.65 items per transaction, which is more than double the UK average (8.61).
3. High variability (std = 47.23) suggests that while many international orders are moderate-sized, some are bulk purchases.
4. Pricing & Revenue Insights: Average unit price: £5.45, slightly higher than the UK average (£4.53). High variability: Standard deviation is £63.36, suggesting the presence of both low-cost and high-end products.

Market Analysis

In [30]:
```
fig, ax = plt.subplots()
fig.set_size_inches(13, 11.5)

# Sorting the data before plotting
sorted_data = retail_non_uk.groupby("Country")["total_cost"].max().reset_index()
sorted_data = sorted_data.sort_values(by="total_cost", ascending=False)

# Creating the bar plot with sorted data
ax = sns.barplot(x="Country", y="total_cost", data=sorted_data, estimator=max, ax=a

# Rotating x-axis labels for better visibility
ax.set_xticklabels(ax.get_xticklabels(), rotation=47, ha="right")

plt.show()
```

```
C:\Users\abhil\AppData\Local\Temp\ipykernel_7512\1242522765.py:12: UserWarning: set_
ticklabels() should only be used with a fixed number of ticks, i.e. after set_ticks
() or using a FixedLocator.
  ax.set_xticklabels(ax.get_xticklabels(), rotation=47, ha="right")
```



In [31]:
```
print(retail_non_uk['CustomerID'].nunique(), "unique customers from other than UK")
print(retail_non_uk['InvoiceNo'].nunique(),"unique orders were placed from other th
print(retail_non_uk['StockCode'].nunique(),"unique products were bought")
print(round(retail_non_uk['CustomerID'].isnull().sum() * 100 / len(retail_non_uk),2
```

```
422 unique customers from other than UK
2406 unique orders were placed from other than UK
2807 unique products were bought
3.19 % of anonymous/guest customers
```

In [32]:
```
groupedProduct = retail_non_uk.groupby('StockCode',as_index= False)['Quantity'].sum
groupedProduct.head(10)
#check how to show product description instead of StockCode
```

Out[32]:

	StockCode	Quantity
1648	23084	15478
1111	22492	12517
279	21212	11337
1241	22629	8374
1242	22630	7161
967	22326	6840
559	21731	6774
2217	84077	5521
1168	22554	5180
1166	22551	4954

Based on the unskewed data let's look at the following top 5 markets:

1. Netherlands
2. France
3. Singapore
4. Japan
5. Hong Kong

Task 2: Data Cleaning and Feature Engineering

In [33]:
```
#load the dataset
market_retail_df = pd.read_excel("../Online Retail.xlsx")
market_retail_df.head()
# dropna
market_retail_df = market_retail_df.dropna()
market_retail_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 406829 entries, 0 to 541908
Data columns (total 8 columns):
 #   Column       Non-Null Count   Dtype         
---  ------       --------------   -----         
 0   InvoiceNo    406829 non-null  object        
 1   StockCode    406829 non-null  object        
 2   Description  406829 non-null  object        
 3   Quantity     406829 non-null  int64         
 4   InvoiceDate  406829 non-null  datetime64[ns]
 5   UnitPrice    406829 non-null  float64       
 6   CustomerID   406829 non-null  float64       
 7   Country      406829 non-null  object        
dtypes: datetime64[ns](1), float64(2), int64(1), object(4)
memory usage: 27.9+ MB
```

In [34]:
```
# Remove negative or zero quantity and price
market_retail_df = market_retail_df[(market_retail_df["Quantity"] > 0) & (market_re

# Create TotalPrice column
market_retail_df["TotalPrice"] = market_retail_df["Quantity"] * market_retail_df["U

# Convert InvoiceDate to datetime format
market_retail_df["InvoiceDate"] = pd.to_datetime(market_retail_df["InvoiceDate"])

# Extract meaningful time-based features
market_retail_df["Year"] = market_retail_df["InvoiceDate"].dt.year
market_retail_df["Month"] = market_retail_df["InvoiceDate"].dt.month
market_retail_df["Day"] = market_retail_df["InvoiceDate"].dt.day
market_retail_df["Hour"] = market_retail_df["InvoiceDate"].dt.hour
market_retail_df["Weekday"] = market_retail_df["InvoiceDate"].dt.weekday

# Drop unnecessary columns
market_retail_df = market_retail_df.drop(columns=["InvoiceNo", "StockCode", "Descri
market_retail_df.head()
```

Out[34]:

	Quantity	UnitPrice	CustomerID	TotalPrice	Year	Month	Day	Hour	Weekday
0	6	2.55	17850.0	15.30	2010	12	1	8	2
1	6	3.39	17850.0	20.34	2010	12	1	8	2
2	8	2.75	17850.0	22.00	2010	12	1	8	2
3	6	3.39	17850.0	20.34	2010	12	1	8	2
4	6	3.39	17850.0	20.34	2010	12	1	8	2

In [35]:
```
# Split data into features and target
X = market_retail_df.drop(columns=["TotalPrice"])
y = market_retail_df["TotalPrice"]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

# Standardize numerical features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Part 2: Machine Learning Model Deployment

Task 3: Model Selection and Training

In [36]:
```
models = {
    "Linear Regression": LinearRegression(),
    "Ridge Regression": Ridge(alpha=10.0),
    "Random Forest": RandomForestRegressor(n_estimators=100, random_state=42),
    "Gradient Boosting": GradientBoostingRegressor(n_estimators=100, random_state=4
    "Support Vector Regressor": LinearSVR()
}
```

```
best_model_name = None
best_r2_score = float('-inf')
best_model = None

for name, model in models.items():
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    r2 = r2_score(y_test, y_pred)
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    print(f"{name} Performance:")
    print(f"R^2 Score: {r2:.4f}")
    print(f"RMSE: {rmse:.4f}\n")

    if r2 > best_r2_score:
        best_r2_score = r2
        best_model_name = name
        best_model = model

print(f"Best Model: {best_model_name} with R^2 Score: {best_r2_score:.4f}\n")
```

```
Linear Regression Performance:
R^2 Score: 0.5226
RMSE: 53.0035

Ridge Regression Performance:
R^2 Score: 0.5226
RMSE: 53.0030

Random Forest Performance:
R^2 Score: 0.9801
RMSE: 10.8161

Gradient Boosting Performance:
R^2 Score: 0.9711
RMSE: 13.0434

Support Vector Regressor Performance:
R^2 Score: 0.5084
RMSE: 53.7820

Best Model: Random Forest with R^2 Score: 0.9801
```

Task 4: Model Evaluation and Interpretation

In [37]:
```
# Hyperparameter tuning for the best model
if best_model_name == "Random Forest":
    param_grid = {
        'n_estimators': [50, 100, 200],
        'max_depth': [None, 10, 20],
        'min_samples_split': [2, 5, 10]
    }
    grid_search = GridSearchCV(RandomForestRegressor(random_state=42), param_grid,
else:
```

```
    grid_search = None

if grid_search:
    grid_search.fit(X_train, y_train)
    best_model = grid_search.best_estimator_

    # Final model evaluation
    y_pred_best = best_model.predict(X_test)
    print(f"Best {best_model_name} Model Performance:")
    print(f"R^2 Score: {r2_score(y_test, y_pred_best):.4f}")
    print(f"RMSE: {np.sqrt(mean_squared_error(y_test, y_pred_best)):.4f}\n")
```

```
Best Random Forest Model Performance:
R^2 Score: 0.9762
RMSE: 11.8255
```

In [38]:
```
# Feature Importance
if best_model_name == "Random Forest":
    importances = best_model.feature_importances_
    feature_names = X.columns
    feature_importance = pd.DataFrame({'Feature': feature_names, 'Importance': impo
    feature_importance = feature_importance.sort_values(by='Importance', ascending=
    print("Feature Importances:")
    print(feature_importance)
```

```
Feature Importances:
      Feature  Importance
0    Quantity    0.873614
1   UnitPrice    0.123662
5         Day    0.001236
2  CustomerID    0.000998
4       Month    0.000295
7     Weekday    0.000095
6        Hour    0.000094
3        Year    0.000007
```

In [40]:
```
import joblib

# Save the best model
joblib.dump(best_model, 'best_model.pkl')

# Save the scaler and PCA if they are not saved yet
joblib.dump(scaler, 'scaler.pkl')
# joblib.dump(pca, 'pca.pkl')
```

Out[40]:
```
['scaler.pkl']
```

Part 3: Deployment and Business impact Analysis

Task 5: Model Deployment

In [41]:
```
%%writefile app.py
from flask import Flask, render_template, request
```

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score

app = Flask(__name__)

model = None
scaler = None
features = None

@app.route('/')
def index():
    return """
    <h1> Upload Online Retail Data as .xlsx file. </h1>
    <form action = '/train' method = 'post' enctype = 'multipart/form-data'>
    <input type = 'file' name = 'file' required>
    <input type='submit' value='Proceed'>
    </form>
    """

@app.route('/train', methods=['POST'])
def train():
    global model, scaler, features  # Ensure global access
    file = request.files['file']
    df = pd.read_excel(file)
    df = df.dropna()

    # Remove negative or zero quantity and price
    df = df[(df["Quantity"] > 0) & (df["UnitPrice"] > 0)]

    # Create TotalPrice column
    df["TotalPrice"] = df["Quantity"] * df["UnitPrice"]

    # Convert InvoiceDate to datetime format
    df["InvoiceDate"] = pd.to_datetime(df["InvoiceDate"])

    # Extract meaningful time-based features
    df["Year"] = df["InvoiceDate"].dt.year
    df["Month"] = df["InvoiceDate"].dt.month
    df["Day"] = df["InvoiceDate"].dt.day
    df["Hour"] = df["InvoiceDate"].dt.hour
    df["Weekday"] = df["InvoiceDate"].dt.weekday

    # Drop unnecessary columns
    df = df.drop(columns=["InvoiceNo", "StockCode", "Description", "InvoiceDate", "

    # Store feature names
    features = list(df.drop(columns=["TotalPrice"]).columns)  # <-- FIXED!

    # Split data into features and target
    X = df[features]
    y = df["TotalPrice"]
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random
```

```
    # Standardize numerical features
    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train)
    X_test = scaler.transform(X_test)

    # Train the model
    model = RandomForestRegressor(n_estimators=100, random_state=42)
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)

    # Model evaluation
    r2 = r2_score(y_test, y_pred)
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))

    return f"""
    <h1>Model Training Complete</h1>
    <p>R^2 Score: {r2:.4f}</p>
    <p>RMSE: {rmse:.4f}</p>
    <a href='/predict'>Go to test prediction</a>
    """


@app.route('/predict', methods=['GET', 'POST'])
def predict():
    global model, scaler, features

    # Ensure model, scaler, and features are trained before prediction
    if model is None or scaler is None or features is None:
        return "<h1>Error: Model has not been trained yet. Please upload data and t

    if request.method == 'POST':
        try:
            input_values = [float(request.form[feature]) for feature in features]
            input_array = np.array(input_values).reshape(1, -1)

            input_scaled = scaler.transform(input_array)
            predicted_price = model.predict(input_scaled)[0]

            return f"""
            <h1>Predicted Total Price</h1>
            <p><strong>Estimated Value:</strong> {predicted_price:.2f}</p>
            <a href='/predict'>Predict another test case</a> | <a href='/'>Back to
            """
        except Exception as e:
            return f"<h1>Error: {str(e)}</h1><a href='/predict'>Try again</a>"

    # Display input form for GET request
    form_html = "<h1>Enter Feature Values</h1><form method='post'>"
    for feature in features:
        form_html += f"<label>{feature}: <input type='text' name='{feature}' requir
    form_html += "<input type='submit' value='Predict'></form>"

    return form_html
```

```
if __name__ == '__main__':
    app.run(debug=True)
```

```
Overwriting app.py
```

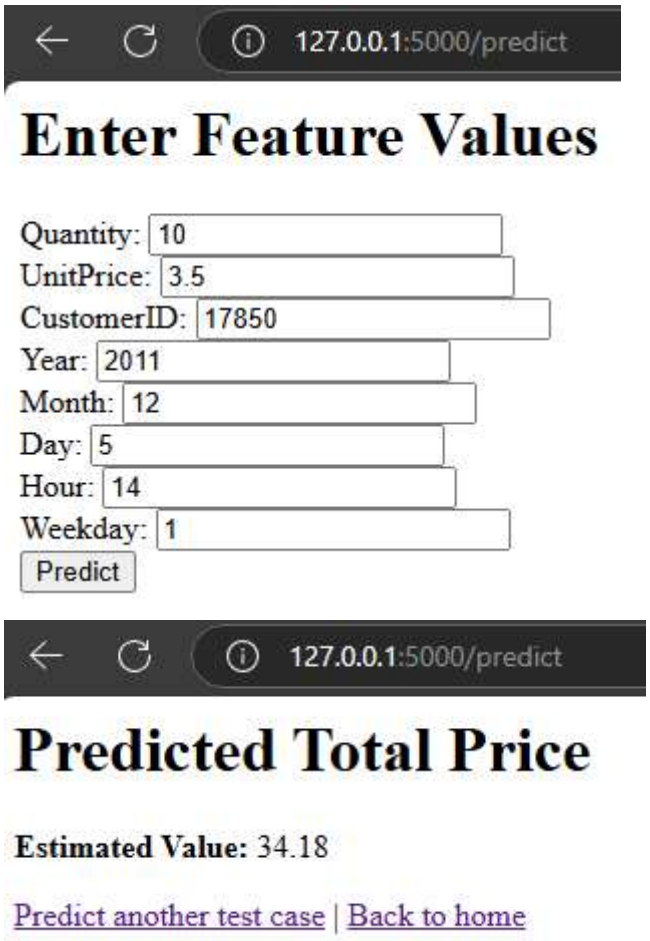
In [ ]:
```
!python app.py
```

Once the above line is submitted open http://127.0.0.1:5000/ in another page and upload the "Online Retail.xlsx" file and click on proceed. It will take a few mins to run the prediction model and return R^2 score ans rmse results.

Home Page:

Train Page:

Predict Page:



Task 6: Business Insights and Recommendations

In [ ]: