National College of Ireland

A hybrid approach towards identifying optimal prices by segmenting customers using active and inactive criteria

MSc Research Project
Data Analytics

Tejali Gangane
Student ID: 21148872

School of Computing
National College of Ireland

Supervisor: Qurrat Ul Ain

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Tejali Gangane
Student ID:	21148872
Programme:	Data Analytics
Year:	2023
Module:	MSc Research Project
Supervisor:	Qurrat Ul Ain
Submission Due Date:	14/08/2023
Project Title:	A hybrid approach towards identifying optimal prices by segmenting customers using active and inactive criteria
Word Count:	
Page Count:	10

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	17th September 2023

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

A hybrid approach towards identifying optimal prices by segmenting customers using active and inactive criteria

Tejali Gangane
21148872

1 Introduction

This document is used to reflect on the steps that were taken to implement my thesis project. The aim of my study is to provide a hybrid approach of generating optimal prices that take into consideration segmentation and customer lifetime value scores.

2 Requirements - Software

The study was implemented using Colab for the coding part and a tool called PowerBI for visualization purposes.

- Reasons for choosing Colab *Google Colaboratory* (n.d.)
  - Allows to write Python code directly into the browser.
  - No requirements for configurations
- Reasons for choosing PowerBI
  - Easy to use
  - Easy to load data and build dashboards

3 Code Execution

3.1 Importing libraries

Figure 1 shows the necessary libraries required for the conducted study. Along with that it also shows the data that was loaded called the 'Online Retail.csv'. This dataset was obtained from UCI ML repository. It shows the steps required to read the data from the CSV file.

```
## Importing required libraries

import pandas as pd
import matplotlib.pyplot as plt
from datetime import date

import scipy.stats as st
import statsmodels.api as sm
import numpy as np
import matplotlib.pyplot as pyplot
import pickle
import pandas as pd
from sklearn.metrics import silhouette_score

import re

## Required for calculating maximum profits i.e. price optimization
from sklearn.preprocessing import MinMaxScaler

## Required for product segmentation, grouping products based on similarity
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity


# Loading the Online retail dataset - downloaded from the UCI ML repository
retail_data = pd.read_csv('Online Retail.csv')
retail_data.shape
```

Figure 1: Importing libraries and loading the data

3.2 Data Preparation

Figures 2 and 3, show the data preparation stage. Figure 4, shows the data preparation results. Certain columns were converted to the respective datatype. Initially, the CustomerID was converted to a String datatype. The InvoiceDate was converted to datetime datatype. Date was extracted from the column. Furthermore, the rows where the Country was 'Unspecified' were filtered out.

```
# Dropping rows where the CustomerID colum has no values
# --- As the study revolves around understanding behavior of customer, and the only identification in the dataset concerning customers is CustomerID,
# --- Rows with no CustomerID values are removed
retail_data.dropna(subset=['CustomerID'], inplace=True)
retail_data.reset_index(drop=True, inplace=True)

# Converting CustomerID to String datatype
retail_data['CustomerID'] = retail_data['CustomerID'].astype(int).astype(str)


# Converting the 'InvoiceDate' Column to date datatype to ensure the date is processed correctly further in the study
retail_data['InvoiceDate'] = pd.to_datetime(retail_data['InvoiceDate'])


# Extract the date from the 'InvoiceDate' column
retail_data['InvoiceDate'] = retail_data['InvoiceDate'].dt.date

# Checking the shape of the dataframe
# --- First value represents the number of rows and second value represents the number of columns
print(retail_data.shape)


# Dropping rows where the Country column is 'unspecified'
# --- Rows with no values or incorrect value for country serves no purpose, hence removing them
retail_data = retail_data[retail_data['Country'] != 'Unspecified']
# Resetting the index of the dataframe as it was filtered out in previous step
retail_data.reset_index(drop=True, inplace=True)
print("Shape of dataframe after removing inconsistent country values = ",retail_data.shape)
```

Figure 2: Data Preparation part 1

Further filtration of rows was performed by filtering out the UnitPrice by ensuring it was not equal to 0.

```
# Dropping rows where the UnitPrice column is 0
# --- Rows where UnitPrice is 0, makes no sense and has no purpose for towards the study as no price manipulation can be performed on such data. Therefore, these are removed
retail_data = retail_data[retail_data['UnitPrice']  != 0].reset_index(drop=True)

# dropping duplicate values
# --- Removing duplicate data to avoid data redundancy as it can skew the results or provide inaccurate outcomes
retail_data = retail_data.drop_duplicates().reset_index(drop=True)


## checking for missing values
print("Checing for missing values\n",retail_data.isna().sum())



# Remove values with non-alphanumeric characters (except digits and letters)
# --- Filtering out inconsistent StockCode values
filtered_stock_codes = [code for code in retail_data['StockCode']  if not code.isalpha()]
retail_data = retail_data[retail_data['StockCode'].isin(filtered_stock_codes)].reset_index(drop=True)

filtered_stock_codes = [sc for sc in retail_data['StockCode'] if sc not in ['BANK CHARGES', 'C2']]
retail_data = retail_data[retail_data['StockCode'].isin(filtered_stock_codes)].reset_index(drop=True)


# new variable called AmountSpent will be calculated as a result of the product of UnitPrice and Quantity.
retail_data['AmountSpent'] = retail_data['UnitPrice'] * retail_data['Quantity']
```

Figure 3: Data Preparation part 2

3.3 Data Segregation

Figures 5 and 6 show data segregation steps for cancelled and valid transactions. Transactions that were cancelled and the ones that went through were grouped together respectively. This was done based on the InvoiceNo attribute. The transactions having InvoiceNo prefixed with a 'C' were deemed to be the cancelled transactions.

```
(406829, 8)
Shape of dataframe after removing inconsistent country values =  (406585, 8)
Checing for missing values
 InvoiceNo      0
StockCode      0
Description    0
Quantity       0
InvoiceDate    0
UnitPrice      0
CustomerID     0
Country        0
dtype: int64
```

Figure 4: Data Preparation results

DATA SEGGREGATION

Seggregating cancelled transactions

```
# Extracting cancelled transactions for further anlaysis

cancelled_inv = ['A']

# Filtering out invoice numbers that are prefixed by 'C'
canceled_transactions = retail_data[retail_data["InvoiceNo"].str.startswith("C")]
canceled_transactions_val = canceled_transactions.copy()


canceled_transactions_val['Quantity'] = canceled_transactions_val['Quantity']*-1
canceled_transactions_val['AmountSpent'] = canceled_transactions_val['AmountSpent']*-1

# Aggregating the data from transactions that were cancelled to get an overview of the data
canceled_transactions_agg = canceled_transactions_val.groupby('CustomerID').agg({
    'InvoiceNo': 'nunique', # number of transactions
    'AmountSpent': 'sum', # total amount spent
    'InvoiceDate' : 'max'
}).reset_index()


canceled_transactions_val.to_csv('Cancelled-Transactions.csv')
canceled_transactions_val.head()
```

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country	AmountSpent
152	C536383	35004C	SET OF 3 COLOURED FLYING DUCKS	1	2010-01-12	4.65	15311	United Kingdom	4.65
233	C536391	22556	PLASTERS IN TIN CIRCUS PARADE	12	2010-01-12	1.65	17548	United Kingdom	19.80
234	C536391	21984	PACK OF 12 PINK PAISLEY TISSUES	24	2010-01-12	0.29	17548	United Kingdom	6.96
235	C536391	21983	PACK OF 12 BLUE PAISLEY TISSUES	24	2010-01-12	0.29	17548	United Kingdom	6.96

Figure 5: Data Segregation - Cancelled transactions

Seggregating valid transactions

```
# Filtering out invoice numbers that are prefixed by 'C' to get the transactions that were not cancelled
valid_transactions_val = retail_data[~retail_data["InvoiceNo"].str.startswith("C")]

# Aggregating the data from transactions that were successful to get an overview of the data
valid_transactions_agg = valid_transactions_val.groupby('CustomerID').agg({
    'InvoiceNo': 'nunique', # number of transactions
    'AmountSpent': 'sum', # total amount spent
    'InvoiceDate' : 'max'
}).reset_index()

valid_transactions_val.to_csv('Valid-Transactions.csv')
```

Figure 6: Data Segregation - Valid transactions

3.4 Data Generation

Figure 7 shows the data generation steps for creating a dataframe to store unique product information and product pricing history. Various products were sold at different prices throughout history and hence, need to be grouped together for further analysis and understanding.

DATA GENERATION

```
# subset of the retail_data will be taken to store unique information about the products which is - StockCOde, Description.
product_info = valid_transactions_val[['StockCode','Description']].drop_duplicates().reset_index(drop=True)
product_info.to_csv('Product-Information.csv')

# product price history -DateProductWasSold, PreviousProductPrice, ProductWasSold,
price_history = valid_transactions_val[['StockCode','Description','UnitPrice','InvoiceDate']].drop_duplicates().reset_index(drop=True)
#import date
price_history['InvoiceDate'] = pd.to_datetime(price_history['InvoiceDate'])
price_history = price_history.sort_values(by=['StockCode','InvoiceDate'], ascending = True).reset_index(drop=True)


# Drop duplicate rows based on 'UnitPrice', keeping only the first occurrence (latest date)
unique_product_prices = price_history.drop_duplicates(subset='UnitPrice', keep='first').reset_index(drop=True)
unique_product_prices.to_csv('Product-Price-Changes.csv')
```

Figure 7: Data generation

3.5 Data Aggregation

Figure 8 shows the steps taken to generate RFM data (Recency, frequency and monetary value). Recency was calculated on the basis of the number of days that had passed since the customer had purchased based on the ”cut-off” date. Frequency was calculated as the

number of times the customer made a certain transaction. MonetaryValue was computed considering the total amount of money spent by the customers throughout their lifetime.

DATA AGGREGATION

```
# Generating RFM model

rfm_retail_data = valid_transactions_val[valid_transactions_val['Quantity'] >=0]
# Aggregate data by customer id
rfm_retail_data = rfm_retail_data.groupby('CustomerID').agg({
    'InvoiceNo': 'nunique', # number of transactions
    'AmountSpent': 'sum', # total amount spent
    'InvoiceDate': 'max' # last transaction date
})

# Find the maximum date in the 'InvoiceDate' column
max_date = rfm_retail_data['InvoiceDate'].max()

# Calculate one day ahead of the maximum date
# --- This is done to calculate the number of days since the customer last purchased
target_date = max_date + pd.DateOffset(days=1)


# Calculate time since last purchase
rfm_retail_data['LastPurchaseDays'] = (pd.to_datetime(target_date) - pd.to_datetime(rfm_retail_data['InvoiceDate'])).dt.days

# Calculate average basket size
rfm_retail_data['AvgBasketSize'] = rfm_retail_data['InvoiceNo']

# Drop unnecessary columns
rfm_retail_data = rfm_retail_data.drop(['InvoiceDate','InvoiceNo'], axis=1)

rfm_retail_data = rfm_retail_data.reset_index(drop=False)
rfm_retail_data.to_csv('RFM-Retail-Data.csv')
```

Figure 8: Data Aggregation

3.6 Modelling

MODELLING

Customer Segmentation

Creating dataframe with customer ids and months in which the customers made a transaction for valid transactions

```
# Extract 'CustomerID' and 'InvoiceDate' columns
customer_data = valid_transactions_val[['CustomerID', 'InvoiceDate']]

# Create a new column 'Month'
customer_data['Month'] = pd.to_datetime(customer_data['InvoiceDate']).dt.to_period('M')

# Group the data by 'CustomerID' and 'Month', and calculate the count of purchases
customer_purchases = customer_data.groupby(['CustomerID', 'Month']).size().reset_index(name='PurchaseCount')

# Pivot the table to reshape the data
customer_purchases = customer_purchases.pivot(index='CustomerID', columns='Month', values='PurchaseCount')

# Replace count of purchases with 1's and fill missing values with 0's
customer_purchases = customer_purchases.notnull().astype(int)


# Convert the period values to their corresponding numerical values
customer_purchases.columns = customer_purchases.columns.astype(str)

customer_purchases.columns = [str(c) for c in customer_purchases.columns]

# Reset the index
customer_purchases.reset_index(inplace=True)
customer_purchases.head()
```

Figure 9: Modelling - customer segmentation - part 1

Identifying the number of times customer made transactions across the month (to check if the customer changed states from inactive to active)

```
# Exclude the 'CustomerID' column before performing the sum
customer_sums = customer_purchases.drop('CustomerID', axis=1).sum(axis=1)

# Create a new DataFrame to store the sums per customer ID
customer_activity_data = pd.DataFrame({
    'CustomerID': customer_purchases['CustomerID'],
    'AttendanceThroughYear': customer_sums
})

customer_activity_data.columns = ['CustomerID','AttendanceThroughYear']
```

```
customer_activity_data
```

	CustomerID	AttendanceThroughYear
0	12346	1
1	12347	6
2	12348	4
3	12349	1
4	12350	1
...	...	...
4325	18280	1
4326	18281	1
4327	18282	2
4328	18283	8

Figure 10: Modelling - customer segmentation - part 2

3.7 Price & Demand Curve

Figure 17 and 18, shows the price and demand curve generated based on the quantities and unit prices available for the products.

CLV

CLV = (Average Purchase Value) x (Average Purchase Frequency) x (Average Customer Lifespan)

Double-click (or enter) to edit

```
# Datafrmae storing the date when customer first made a transaction
first_purchase_date = valid_transactions_val.groupby('CustomerID').agg({
    'InvoiceDate' : 'min', #First transaction made
}).reset_index()
first_purchase_date.columns = ['CustomerID','FirstPurchaseDate']

# Datafrmae storing the date when customer's last transaction
last_purchase_date = valid_transactions_val.groupby('CustomerID').agg({
    'InvoiceDate': 'max', # last transaction date
}).reset_index()
last_purchase_date.columns = ['CustomerID','LastPurchaseDate']
```

Figure 11: Modelling - customer lifetime value - part 1

```
# Merging data of first and the last purchase made to create a dataframe consisting of purchase timeline
purchase_timeline = first_purchase_date.merge(last_purchase_date)

# Convert the date columns to datetime format
purchase_timeline['Start_Date'] = pd.to_datetime(purchase_timeline['FirstPurchaseDate'])
purchase_timeline['End_Date'] = pd.to_datetime(purchase_timeline['LastPurchaseDate'])

# Calculate the difference in months between the two date columns
purchase_timeline['Months_Difference'] = (purchase_timeline['End_Date'].dt.to_period('M')- purchase_timeline['Start_Date'].dt.to_period('M')).apply(lambda x: x.n) + 1
purchase_timeline.drop(['Start_Date','End_Date'], axis = 1, inplace=True)

# Merging the customers sum data with the purchase timeline data
purchase_timeline = purchase_timeline.merge(customer_activity_data)
purchase_timeline['LoyaltyRate'] = round(purchase_timeline['AttendanceThroughYear']/purchase_timeline['Months_Difference'],2)

purchase_timeline = purchase_timeline.merge(rfm_retail_data)



purchase_timeline['CLV'] = purchase_timeline['AmountSpent'] * purchase_timeline['LastPurchaseDays'] * purchase_timeline['AvgBasketSize'] * purchase_timeline['LoyaltyRate']
purchase_timeline.to_csv('Purchase-Timeline.csv')

purchase_timeline.head()
```

	CustomerID	FirstPurchaseDate	LastPurchaseDate	Months_Difference	AttendanceThroughYear	LoyaltyRate	AmountSpent	LastPurchaseDays	AvgBasketSize	CLV
0	12346	2011-01-18	2011-01-18	1	1	1.00	77183.60	327	1	2.523904e+07
1	12347	2010-07-12	2011-10-31	16	6	0.38	4310.00	41	7	4.700486e+05
2	12348	2010-12-16	2011-09-25	10	4	0.40	1437.24	77	4	1.770680e+05
3	12349	2011-11-21	2011-11-21	1	1	1.00	1457.55	20	1	2.915100e+04
4	12350	2011-02-02	2011-02-02	1	1	1.00	294.40	312	1	9.185280e+04

Figure 12: Modelling - customer segmentation - part 2

```
# Define the cutoff date
cutoff_date = pd.to_datetime('2011-11-30')
purchase_timeline = purchase_timeline[purchase_timeline['FirstPurchaseDate'] <= cutoff_date]
# Create the 'segment' column based on the conditions
purchase_timeline['Segment'] = purchase_timeline['LastPurchaseDate'].apply(lambda x: 'Inactive' if x > cutoff_date else 'Active')
purchase_timeline = purchase_timeline.reset_index(drop=True)
```

Figure 13: Modelling - customer segmentation - part 3

```
#product descriptions
product_descriptions = product_info['Description']

# Create a TF-IDF vectorizer
vectorizer = TfidfVectorizer()

# Transform the product descriptions into TF-IDF vectors
tfidf_matrix = vectorizer.fit_transform(product_descriptions)

# Calculate cosine similarity between descriptions
cosine_similarities = cosine_similarity(tfidf_matrix, tfidf_matrix)

# Print the cosine similarity matrix
print(cosine_similarities)

from sklearn.cluster import KMeans

# Number of segments/clusters
num_clusters = 3

# Perform K-Means clustering
kmeans = KMeans(n_clusters=num_clusters, random_state=0)
clusters = kmeans.fit_predict(cosine_similarities)

# Assign cluster labels to product descriptions
segmented_products = {cluster: [product_descriptions[i] for i, c in enumerate(clusters) if c == cluster]
                      for cluster in range(num_clusters)}
```

Figure 14: Modelling - Product segmentation - part 1

```
# Create a list to store rows
rows = []

# Iterate over the dictionary and create rows for each value
for segment, products in segmented_products.items():
    for product in products:
        rows.append({'Segment': segment, 'Description': product})

# Convert the list of rows to a DataFrame
segmented_products = pd.DataFrame(rows)

unique_product_prices = unique_product_prices.merge(segmented_products)
```

```
[[1.         0.21673257 0.         ... 0.         0.76850508 0.        ]
 [0.21673257 1.         0.         ... 0.         0.         0.        ]
 [0.         0.         1.         ... 0.         0.18842232 0.        ]
 ...
 [0.         0.         0.         ... 1.         0.         0.        ]
 [0.76850508 0.         0.18842232 ... 0.         1.         0.        ]
 [0.         0.         0.         ... 0.         0.         1.        ]]
```

Figure 15: Modelling - Product segmentation - part 2

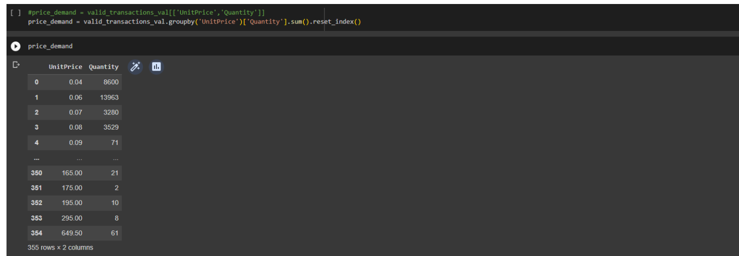
```
# Calculate the silhouette score for the clustering
silhouette_avg = silhouette_score(cosine_sim, clust)
print(f"Silhouette Score: {silhouette_avg}")
```

```
Silhouette Score: 0.07303118852710831
```

Figure 16: Modelling - Product segmentation - Evaluating clusters

3.8 Price Elasticity

Figure 20-21 shows the price elasticity scores generated for the products.



	UnitPrice	Quantity
0	0.04	8600
1	0.06	13963
2	0.07	3280
3	0.08	3529
4	0.09	71
...	...	...
350	165.00	21
351	175.00	2
352	195.00	10
353	295.00	8
354	649.50	61

355 rows × 2 columns

Figure 17: Price and Demand curve - part 1

Figure 18: Price and Demand curve - part 2

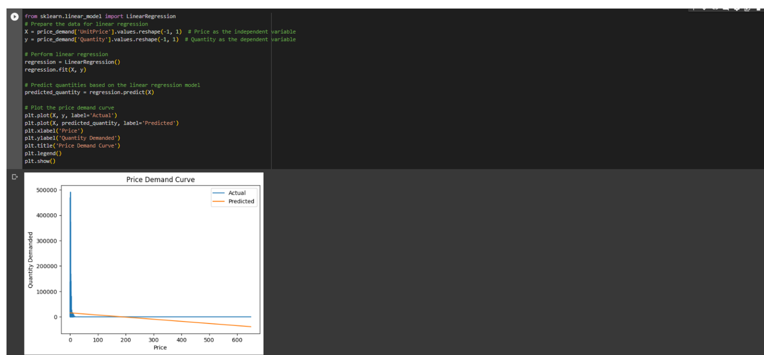


Figure 19: Price and Demand curve - part 3

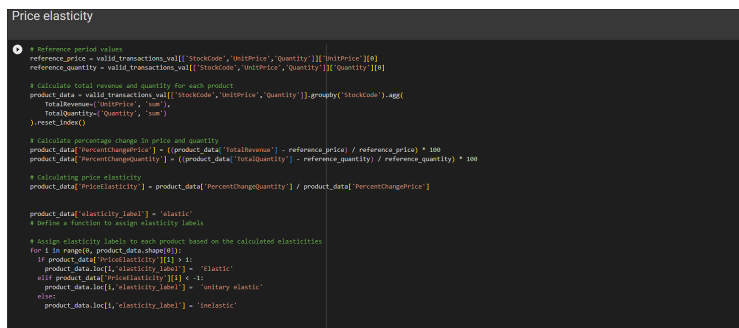


Figure 20: Price Elasticity

3.9 Price Optimization

Figures 22 to 25 shows the steps performed to generate optimal prices for the segments.

```
product_data
```

	StockCode	TotalRevenue	TotalQuantity	PercentChangePrice	PercentChangeQuantity	PriceElasticity	elasticity_label
0	10002	41.65	823	1533.333333	13616.666667	8.880435	Elastic
1	10080	8.65	291	239.215686	4750.000000	19.856557	Elastic
2	10120	6.09	192	138.823529	3100.000000	22.330508	Elastic
3	10123C	1.95	5	-23.529412	-16.666667	0.708333	inelastic
4	10124A	2.10	16	-17.647059	166.666667	-9.444444	unitary elastic
...	...	...	...	...	...	...	...
3654	90214U	0.29	12	-88.627451	100.000000	-1.128319	unitary elastic
3655	90214V	2.79	14	9.411765	133.333333	14.166667	Elastic
3656	90214W	0.29	12	-88.627451	100.000000	-1.128319	unitary elastic
3657	90214Y	1.83	61	-28.235294	916.666667	-32.465278	unitary elastic
3658	90214Z	0.29	12	-88.627451	100.000000	-1.128319	unitary elastic

3659 rows × 7 columns

Figure 21: Price Elasticity - part 2

Maximum Profit generation - Price optimization

Profit calculation

```
import scipy.optimize as optimize
retail_datac = purchase_timeline[['CustomerID','Segment']].merge(valid_transactions_val[valid_transactions_val['InvoiceDate']<=cutoff_date][['CustomerID','StockCode', 'UnitPrice', 'Quantity','Ar
def find_optimal_price_each_segment(stockid, segment, profit_multipliers):
    # Defining profit  calculation function
    def calculate_profit(price, customer_segment):
        # Calculation of profit based on the price and customer segment
        # Customize this calculation based on your business logic and historical data


        base_profit = price * base_profit_multiplier
        segment_multiplier = profit_multipliers.get(customer_segment, 1.0)
        profit= base_profit * segment_multiplier

        return profit

    # Define objective function
    def objective_function(price):
        return -calculate_profit(price, segment)

    # Define data-driven constraints based on historical patterns

    segment_data = retail_datac[(retail_datac['StockCode'] == stockid) & (retail_datac['Segment'] == segment)]
    #print("Segment = ", segment_data)
    min_price = segment_data['UnitPrice'].min()  # Minimum price allowed
    max_price = segment_data['UnitPrice'].max()  # Maximum price allowed
    #print("Min = ", min_price, " ----- MAX = ", max_price)
    # Setting bounds for optimization
```

Figure 22: Price optimization - part 1

```
    ## If there does not exists any products under this segment
    if ((np.isnan(min_price) or np.isnan(max_price)) and segment == 'Inactive'):
      #print("YESSSSSSSSSSSSSSSSSSS")
    # Handle NaN values by setting default bounds
      temp = retail_datac[(retail_datac['StockCode'] == stockid) & (retail_datac['Segment'] == 'Active')]
      temp_min = temp['UnitPrice'].min()  # Minimum price allowed
      temp_max = temp['UnitPrice'].max()  # Maximum price allowed
      bounds = (temp_min, temp_max)  # setting bounds
      #print(bounds)
    elif (np.isnan(min_price) or np.isnan(max_price) and segment == 'Active'):
      temp = retail_datac[(retail_datac['StockCode'] == stockid) & (retail_datac['Segment'] == 'Inactive')]
      temp_min = temp['UnitPrice'].min()  # Minimum price allowed
      temp_max = temp['UnitPrice'].max()  # Maximum price allowed
      bounds = (temp_min, temp_max)  # setting bounds
    else:
        if min_price > max_price:
            bounds = (min_price , max_price)
        elif min_price == max_price:
          bounds = (min_price, min_price + 1.0)
        else:
            bounds = (min_price, max_price)  # Usual bounds
    print(bounds)
    # Perform optimization
    #result = optimize.minimize_scalar(objective_function, method='Brent')
    #print(bounds)
    result = optimize.minimize_scalar(objective_function, bounds=bounds, method='bounded')

    # Extract optimal price
    optimal_price = result.x
    return optimal_price
```

Figure 23: Price optimization - part 2

```
#################### PROFIT MULTIPLIER CALCULATION
# Calculate average amount spent for each segment
segment_amount_spent = retail_datac.groupby('Segment')['AmountSpent'].mean()

# Finding the segment which has the highest average amount spent
reference_segment = segment_amount_spent.idxmax()

# Calculating profit multipliers based on relative amount that was spent in comparison to the reference segment
profit_multipliers = segment_amount_spent / segment_amount_spent.loc[reference_segment]
profit_multipliers = profit_multipliers.to_dict()

# Base profit multiplier (can be adjusted based on business logic)
base_profit_multiplier = 1

# Initialize list of segments
segments = ['Active', 'Inactive']

# Initialize dictionary to store optimal prices
optimal_prices = {}

# Iterate over products
for stockid in unique_product_prices['StockCode']:
    segment_prices = {}
    for segment in segments:
        print(stockid, segment)
        if(stockid!='BANK CHARGES'):
          optimal_price = find_optimal_price_each_segment(stockid, segment, profit_multipliers)
          segment_prices[segment] = optimal_price
    optimal_prices[stockid] = segment_prices
```

```
(0.42, 0.73)
82580 Active
(0.42, 1.25)
82580 Inactive
(0.42, 0.92)
84347 Active
(1.74, 4.96)
```

Figure 24: Price optimization - part 3

3.10 Simulation

Figures 26, 27 and 28 show the simulation conducted to see how well optimal prices were with respect to already existing product quantities.

```python
optimal_prices
predicted_prod_segment_price = round(pd.DataFrame(optimal_prices).T, 4).reset_index()
predicted_prod_segment_price.columns = ['StockCode','Active','Inactive']
predicted_prod_segment_price[0:20]
```

	StockCode	Active	Inactive
0	10002	1.85	1.85
1	10080	0.85	1.39
2	10120	1.21	1.21
3	10123C	1.65	1.65
4	10124A	1.42	0.42
5	10135	2.46	1.25
6	11001	3.29	2.69
7	15030	1.29	1.29
8	15034	0.14	1.14
9	15036	0.83	0.83
10	15044A	5.79	3.95
11	15056BL	12.46	5.95
12	15056P	12.46	6.95
13	15058A	7.95	7.95
14	15058C	7.95	8.95
15	15060B	8.29	4.75
16	16008	0.25	0.25
17	16014	0.42	1.42

Figure 25: Price optimization - part 4

SIMULATION

```python
import pandas as pd
import numpy as np



optimal_prices_df = predicted_prod_segment_price.merge(unique_product_prices.groupby('StockCode').mean()['UnitPrice'].reset_index()) #predicted_prod_segment_price.copy()
temp = valid_transactions_val[['StockCode','Quantity']].groupby('StockCode').median().reset_index()
optimal_prices_df = optimal_prices_df.merge(temp)

# Simulating different scenarios based on the quantity already provided in the dataset
simulation_results = []
for scenario in ['Base', 'PriceChange']:
    simulated_sales = []

    for index, row in optimal_prices_df.iterrows():
        if scenario == 'Base':
            price = row['UnitPrice']
            #print("From Base, price = ", price)
        elif scenario == 'PriceChange':
            price = row['Active']
            #print("From price change, price = ", price)


        simulated_revenue = row['Quantity'] * price

        # Calculating simulated profit
        simulated_profit = simulated_revenue

        simulated_sales.append(simulated_profit)

    simulation_results.append((scenario, simulated_sales))
```

Figure 26: Simulating revenue - part 1

```python
# Display simulation results
priceactchange = []
orginialrev = []
for scenario, results in simulation_results:
    #print(f"Scenario: {scenario}")

    #print("Simulated Profits:", results)
    if(scenario == 'PriceChange'):
      priceactchange.extend(results)
    else:
      orginialrev.extend(results)

optimal_prices_df['OriginalRev'] = orginialrev
optimal_prices_df['PriceChangeActiveRev'] = priceactchange
```

Figure 27: Simulating revenue - part 2

```python
optimal_prices_df[['StockCode','Active','UnitPrice','OriginalRev','PriceChangeActiveRev']][0:20]
```

	StockCode	Active	UnitPrice	OriginalRev	PriceChangeActiveRev
0	10002	1.85	0.850000	10.200000	22.200
1	10080	0.85	0.390000	4.680000	10.200
2	10120	1.21	0.210000	1.050000	6.050
3	10123C	1.65	0.650000	0.650000	1.650
4	10124A	1.42	0.420000	1.680000	5.680
5	10135	2.46	1.255000	12.550000	24.600
6	11001	3.29	2.083333	33.333333	52.640
7	15030	1.29	0.290000	1.450000	6.450
8	15034	0.14	0.105000	2.520000	3.360
9	15036	0.83	0.707500	8.490000	9.960
10	15044A	5.79	3.763333	22.580000	34.740
11	15056BL	12.46	5.912000	17.736000	37.380
12	15056P	12.46	4.600000	13.800000	37.380
13	15058A	7.95	7.350000	14.700000	15.900
14	15058C	7.95	3.950000	9.875000	19.875
15	15060B	8.29	6.020000	24.080000	33.160
16	16008	0.25	0.100000	2.400000	6.000
17	16014	0.42	0.320000	6.400000	8.400
18	16015	0.50	0.430000	4.300000	5.000
19	16045	0.12	0.040000	4.000000	12.000

Figure 28: Simulating revenue - part 3

3.11 Final Results

Fig 29. shows the revenue generated for the original price points and revenue generated for optimal prices for the active segment.

```
print("Original Revenue based on the original UnitPrice = ",optimal_prices_df['OriginalRev'].sum(), "\nRevenue after optimal prices were depicted for active segment = ",optimal_prices_df['Price(
```

```
Original Revenue based on the original UnitPrice =  7733.47975
Revenue after optimal prices were depicted for active segment =  10281.605
```

Figure 29: Final result

4 Data Analysis using PowerBI

Figure 30 indicates the data analysis performed in PowerBI to get an overview of the data.

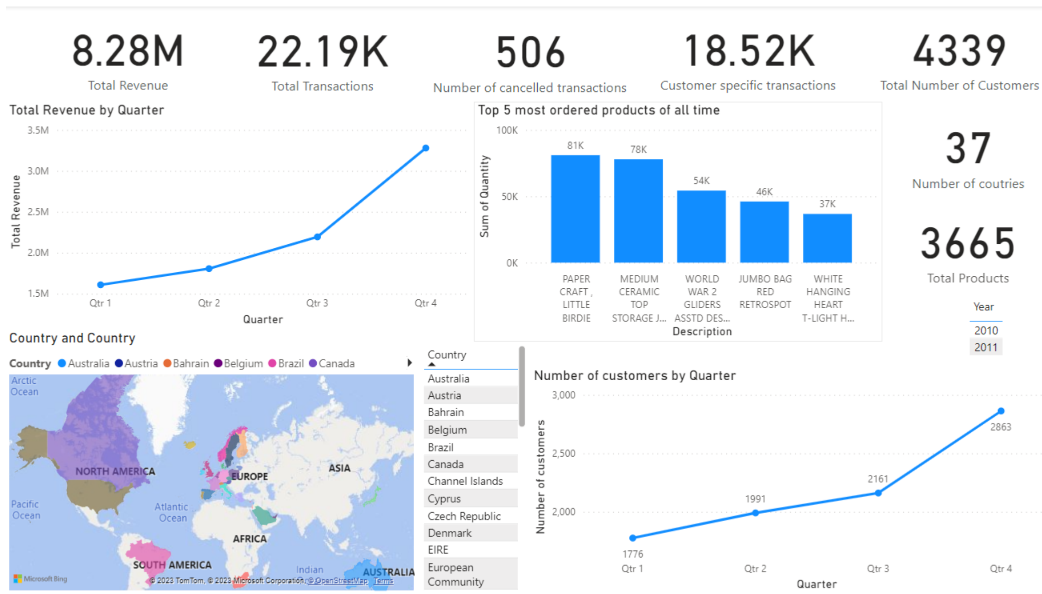


Figure 30: Data Analysis to get an overview of the data

References

Google Colaboratory (n.d.). https://colab.research.google.com/?utm_source=scs-index.