DUNA ANALYTICS
THE ERM PRODUCT RECOMMENDATION ENGINE

June 2020

The Expected Revenue Maximization (ERM) Product Recommendation Engine

This paper describes the Expected Revenue Maximization (ERM) Product Recommendation Engine developed at Duna Analytics by Barrett Duna.

DUNA ANALYTICS
The ERM PRODUCT RECOMMENDATION ENGINE

THE EXPECTED REVENUE MAXIMIZATION (ERM) PRODUCT RECOMMENDATION ENGINE

INTRODUCTION

Executive Summary

The Expected Revenue Maximization Product Recommendation (ERM) Algorithm was developed by Barrett Duna at Duna Analytics. The ERM Algorithm suggests products to a customer already purchasing a product (this will be referred to as the *primary product)* based on past orders from other customers that included the primary product.

A naïve approach would be to show the user a product that is most often purchased with the primary product (sadly this is the strategy for many ecommerce sites – displaying frequently purchased items). We call this the Probability Maximization Product Recommendation (PM) Algorithm. While leading to the highest purchases rate (because the most likely purchased item is being suggested), the PM Algorithm ignores the *suggested product's* price thus possibly suggesting a very low-priced product (which often have higher purchase rates) leading to a small average purchase order per customer.

In contrast, the ERM Algorithm considers both probability of purchase conditional on the primary product and suggested total order revenue and multiplies the total *product bundle* (the items purchased) revenue by the probability of those items being purchased based on past product bundle purchases by previous customers.

A simulation of the two algorithms showed the ERM Algorithm drastically outperforms the PM Algorithm. Our results show simply displaying frequently purchased items to consumers is suboptimal. Rather, expected revenue (the expected value of the product bundle revenue) maximization is the optimal solution to the product recommendation problem. In general, our algorithm (ERM) generates massive improvements over the simple highest probability of purchase algorithm (PM).

Summary of ERM Methodology

Our approach differs from the PM Algorithm in that it not only considers the probability of purchase of a suggested product (in other words the frequency of purchase or "popularity" of a product), but it also considers total revenue generated by the purchase. It weights total revenue of the product combination by the probability of purchase. In probability theory this is known as expected value. We then suggest the top products with the highest expected revenue versus the products with the highest probability of purchase.

Mathematics of EMR Algorithm

Explanation of Mathematics

Expected Revenue(Primary Product, Secondary Product) = (Total Revenue of Combination) x (Probability of Purchase)

Suppose you have only two products in a hypothetical situation. Product A is purchased 75% of the time and has a price of $1.00 and product B is purchased the remaining 25% of the time with a price of $100.

The expected revenue of product A is 0.75x$1.00 = $0.75 and the expected revenue of product B is 0.25x$100 = $25.00.

The PM Algorithm will suggest product A because it has a higher probability of purchase and the ERM Algorithm will suggest product B because it has a higher expected revenue.

Now suppose 1,000 customers visit your business. You should expect 75% to purchase product A or 750 customers and 25% to purchase product B or 250 customers. This implies 750x$1.00 = $750 in revenue is generated from product A and 250x$100 = $25,000 will be generated from product B.

Here, we see the ERM Algorithm is the clear winner generating $25,000 versus $750 generated by the PM Algorithm. This is because it doesn't matter what the customer is most likely to purchase. What matters is the expected revenue from each purchase.

If we had a product X and we were to choose between recommending product A or product B and could only recommend one to the customer, we would chose product B. We would receive less sales of the recommended product, but more revenue.

Methodology of the Experiment

Dividing the Dataset

We took sales data from an electronics store and broke the dataset up into customers that purchased one product and customers that purchased two or more products. We threw out customers that purchased more than two products 1) for the purpose of the study and 2) it didn't significantly impact the results because most customers that purchased two or more products only purchased two. We also threw out customers that purchased the same product twice for the purposes of the study which was a very small minority.

There were 171,301 single product purchasing customers and 6,474 customers that purchased two products. We refer to the combination of two products purchased by the 6,474 two product customers as a *product bundle*. The single product purchasing customers was our test data for the study. Each of the single product purchasing customers were assumed to have been shown no suggested products hence only a single purchase.

Assigning Secondary Products to the Single Product Customers

Then we paired up each single product purchasing customer with a second product randomly selected from the 6,474 product bundles. We paired up bundles that included the single product purchasing customer's primary product. Thus, if a single product purchasing customer bought an iPhone, we filtered the 6,474 product bundles down to bundles including an iPhone and randomly chose one bundle as the second product (referred to as the *secondary product*). Thus, each single product purchasing customer now had a primary and secondary product. We referred to the customers reassigned with a new secondary product as the *test customers*.

Assigning secondary products this way makes logical sense. First, it obeys product bundles purchased (if the test customer's primary product was an iPhone, only past multi-purchases including an iPhone were considered when assigning a secondary product).

Second, it obeys the distribution of product bundle sales. Product bundles were repeated (more than one two product purchasers purchased the same two products) thus certain product bundles were purchased more often and other bundles less often. Thus, if a test custom's primary product was an iPhone, they are more likely to be paired with Apple Airpod Headphones than Bose SoundSmart Headphones because many more product bundles included both the Apple products than the Apple and Bose product combination.

Calculating Probabilities of Product Bundle Purchases

We calculated what are known as *conditional probabilities*. The probabilities of a random customer purchasing a random bundle such as an iPhone and Apple Airpod Headphones is different than the probability of a test customer whose primary product is an iPhone purchasing Apple Airpods Headphones as the secondary product. Knowing the test customer purchased an iPhone increases the odds that the secondary purchased product is Apple Airpod Headphones. Randomly choosing a product bundle that happens to be an iPhone and Apple Airpods Headphones has a different probability.

When you calculate the conditional probability, you filter the product bundle list down to bundles including the primary product e.g. an iPhone and find the number of bundles in this reduced set of bundles including for example Apple Airpods Headphones and divide by the total number of iPhone bundles.

In contrast, calculating the probability of a given bundle e.g. iPhone and Apple Airpods Headphones involves simply counting the number of iPhone and Apple Airpods Headphone product bundles and dividing by the total number of bundles.

Conditional probabilities are the relevant probability because we assume the customer is already purchasing the primary product and seeing recommended products to choose from thus, we filter the list of product bundles down to a list containing the primary product.

Results

The ERM Algorithm clearly and by a large margin outperforms the naïve PM Algorithm. As you can see in the in the table below and **Appendix A**, the ERM Algorithm outperforms the PM Algorithm for all recommendation counts and the gap widens as number of products shown increases. When one product is recommended to the user EMR delivers an extra $2.5m to $32.2m on $29.8m in PM revenue which is an 8.3% increase in revenue. This gap widens as number of recommendations increase. At five recommendations, EMR boosts revenue by $9.2m to $58m which is an 18.9% increase over the PM revenue number of $48.8m. Note both revenue and percentage increase in revenue are increasing in the number of recommendations.

Number of Recommendations	ERM Algorithm Revenue	PM Algorithm Revenue	Increase in Revenue	% Increase in Revenue
1	$ 32,221,406	$ 29,759,974	$ 2,461,432	8.3%
2	$ 42,977,872	$ 37,891,309	$ 5,086,563	13.4%
3	$ 49,632,049	$ 42,844,190	$ 6,787,860	15.8%
4	$ 54,107,412	$ 45,458,404	$ 8,649,008	19.0%
5	$ 58,024,077	$ 48,813,225	$ 9,210,853	18.9%

Appendix A and the table below show how average order revenue was significantly greater for ERM versus PM and the gap grew as the number of recommended products shown to the customer increased. Although, average order revenue total decreased for both EMR and PM with an increase in number of recommendations (with an increase in number of recommendations, purchase count increases dramatically reducing average order revenue totals).

Number of Recommendations	Average. Order Revenue (ERM)	Average Order Revenue (PM)	Average Order Difference
1	$ 765.63	$ 598.26	$ 167.36
2	$ 735.80	$ 518.17	$ 217.62
3	$ 741.78	$ 476.78	$ 265.00
4	$ 714.26	$ 439.08	$ 275.18
5	$ 682.80	$ 418.58	$ 264.23

For example, average order revenue total for EMR and PM with a one product recommendation was $765.63 and $598.26 respectively for a difference of $167.36. At five recommendations displayed, the EMR and PM average order revenue total was $682.80 and $418.58 producing a difference of $264.23.

Conversion rates also rose dramatically with number of recommendations shown. This is not surprising because it gives the customer more options increasing the likelihood of purchase. Below the conversion rates are shown. Notice, the conversion rates are higher for the PM Algorithm. This is by design of the algorithm. It is showing the products most likely to be purchased thus maximizing conversion rates. This analysis shows maximizing conversion rates for a product recommendation doesn't necessarily translate into maximum revenue.

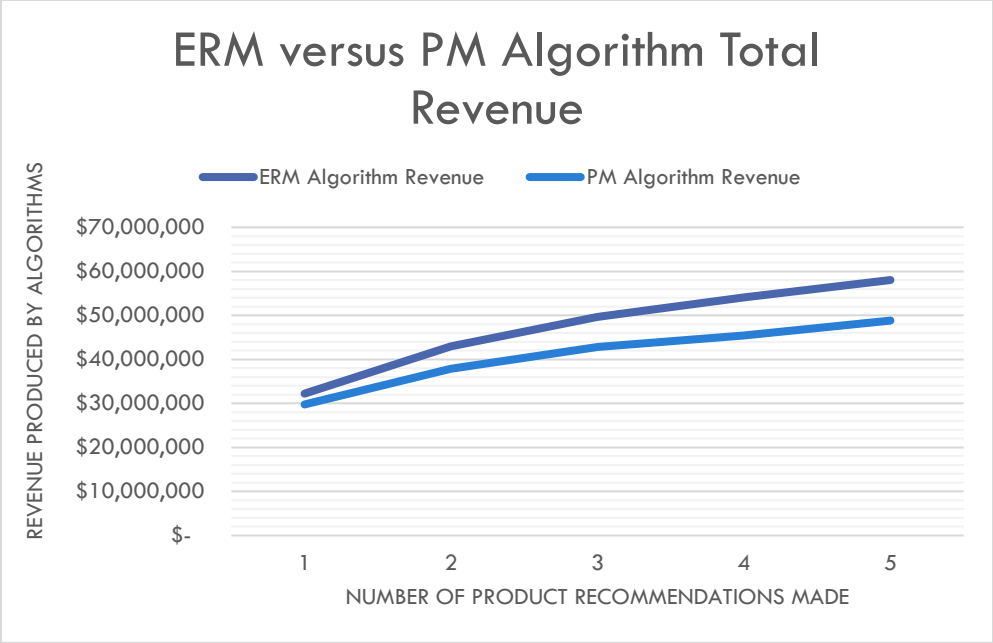
Number of Recommendations	ERM Conversion Rate	PM Conversion Rate	Change Conversion Rate
1	24.6%	29.0%	-4.5%
2	34.1%	42.7%	-8.6%
3	39.1%	52.5%	-13.4%
4	44.2%	60.4%	-16.2%
5	49.6%	68.1%	-18.5%

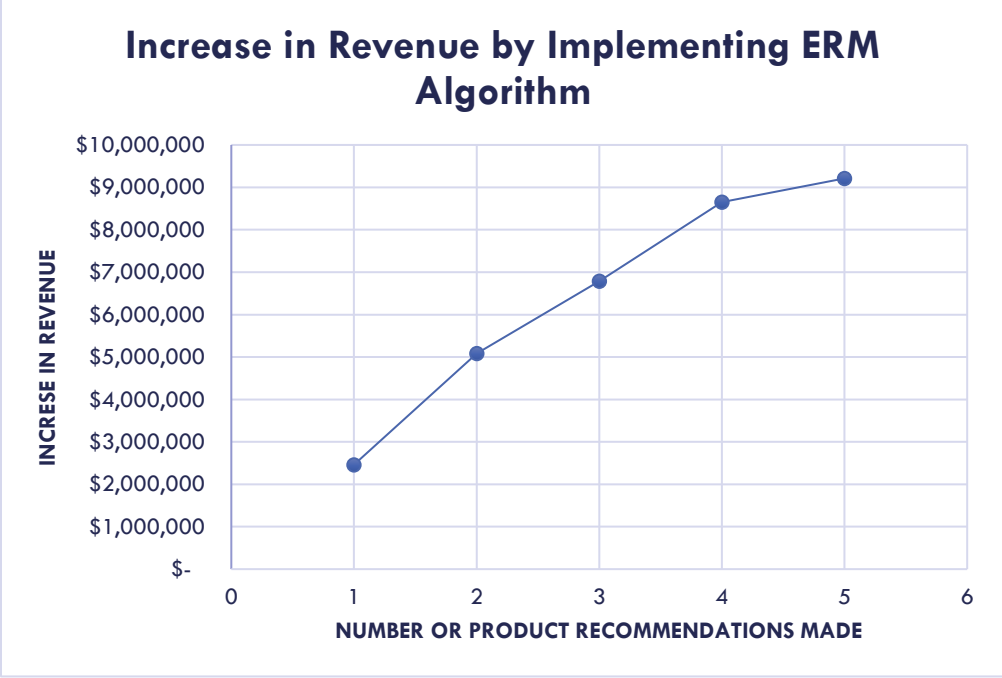
Total number of orders was lower for the ERM Algorithm since the PM Algorithm specifically delivers recommendations with the highest probability of purchase. In both cases, total orders increase with the number of recommendations.

Number of Recommendations	ERM # of Orders	PM # of Orders
1	42,085	49,744
2	58,410	73,125
3	66,909	89,861
4	75,753	103,530
5	84,979	116,617

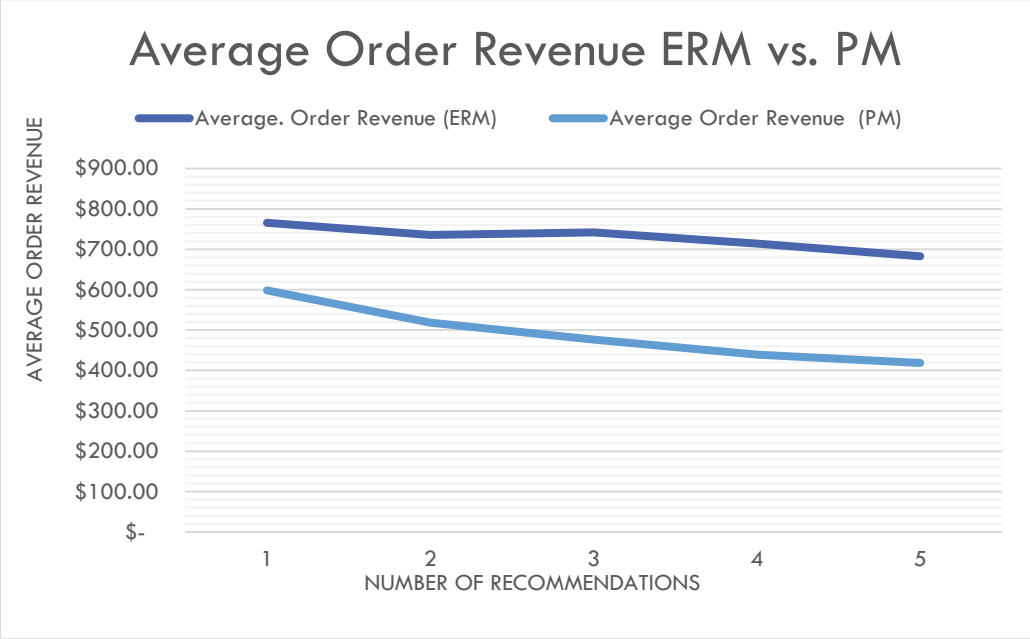
In conclusion, the ERM Algorithm is a massive improvement over the PM Algorithm. The reality is many businesses, typically ecommerce, employ the PM Algorithm showing the most popular products first rather than the expected revenue maximizing products. The most popular products get displayed on the home page without considering of expected revenue.

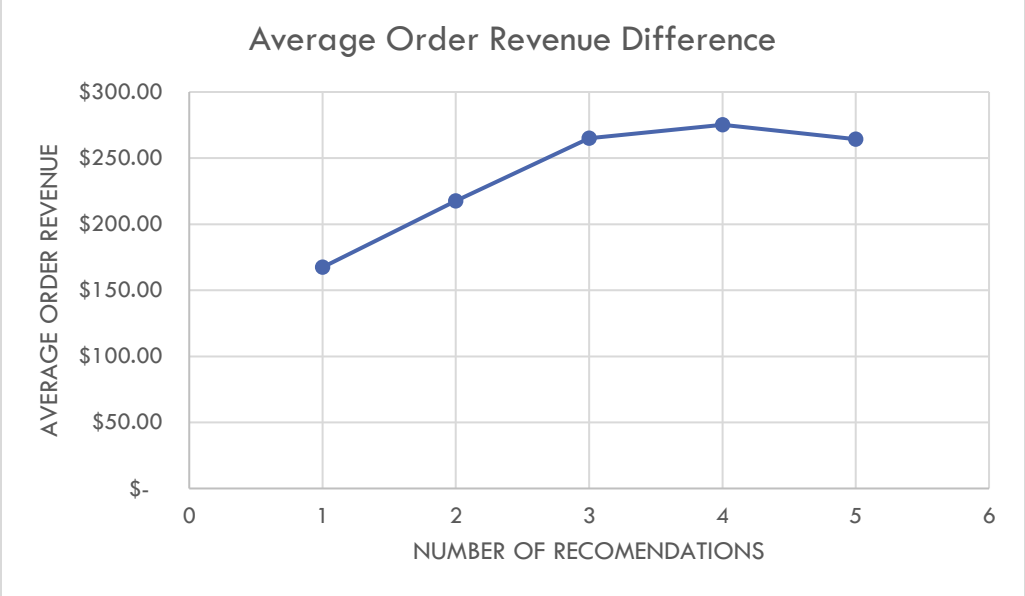
Appendix A: Plots of Results





Appendix A: Continued





Product Recommender Engine

June 23, 2020

1 ERM PRODUCT RECOMMENDER ENGINE

[325]:
```
# import packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import os
import random
```

[326]:
```
# change directory and read in sales data
DATASET_DIRECTORY = 'D:/Datasets/Sales Data/'
os.chdir(DATASET_DIRECTORY)
sales_data = pd.read_csv('sales_data.csv')
```

[327]:
```
# some of the sales data has blank lines
# drop NaN's
sales_data.dropna(inplace=True)
```

1.0.1 SEPERATE SINGLE PRODUCT PURCHASES FROM MULTI-PRODUCT PURCHASES AND PREPARE DATA

[328]:
```
# filter for orders with multiple products purchased
multi_data = sales_data.copy()[sales_data.duplicated('Order ID', keep=False)]
```

[329]:
```
# test if all multiple data
sum(~multi_data.duplicated('Order ID', keep=False))
```

[329]:
```
0
```

[330]:
```
# some rows are just the column names
# drop these rows
multi_data = multi_data[multi_data['Product'] != 'Product']
```

[331]:
```
# some price data is int and other float so convert all
# to float
multi_data['Price Each'] = multi_data['Price Each'].apply(float)
```

[332]:
```
# rename the 'Price Each' column 'Price'
multi_data = multi_data.rename({'Price Each': 'Price'}, axis=1)
```

[333]:
```
# filter for orders where the customer only bought one product
single_data = sales_data[~sales_data.duplicated('Order ID', keep=False)]
```

[334]:
```
# test for duplicated 'Order ID'
sum(single_data.duplicated('Order ID', keep=False))
```

[334]:
```
0
```

[335]:
```
# rename product column to primary column
single_data = single_data.copy().rename({'Product': 'Primary Product'}, axis=1)
```

[336]:
```
# create a price table with unique product/price rows
price_table = multi_data[['Product', 'Price']].drop_duplicates()
```

[337]:
```
# convert table to price dictionary
price_dict = dict(zip(price_table['Product'], price_table['Price']))
```

[338]:
```
# drop unnecessary columns
multi_data.drop(['Quantity Ordered', 'Price', 'Order Date', 'Purchase␣
↪Address'], axis=1, inplace=True)
```

[339]:
```
# drop unnecessary columns
single_data = single_data.copy().drop(['Quantity Ordered', 'Price Each', 'Order␣
↪Date', 'Purchase Address'], axis=1)
```

[340]:
```
# sort to order products so the same bundle doesn't get put
# in two different orders
multi_data.sort_values(['Order ID', 'Product'], inplace=True)
```

[341]:
```
# group multiple products from each order into a tuple
multi_data = multi_data.groupby('Order ID')['Product'].apply(tuple)
```

[342]:
```
# reset index to convert to dataframe
multi_data = multi_data.reset_index()
```

[343]:
```
# convert tuple of products into list of products
multi_data['Product'] = multi_data['Product'].apply(list).reset_index(drop=True)
```

[344]:
```
# calculate number of products in each order
multi_data['Order Size'] = multi_data['Product'].map(len)
```

[345]:
```
# filter out orders with size greater than two
multi_data = multi_data[multi_data['Order Size'] == 2]
```

[346]:
```
# drop 'Order Size' and 'Order ID' columns
multi_data.drop(['Order Size', 'Order ID'], axis=1, inplace=True)
```

[347]:
```
# convert multi_data to list of lists with each inner list
# containing a product bundle pair
product_bundles = multi_data['Product'].values.tolist()
```

[348]:
```
# drop unnecessary 'Order ID' column
single_data.drop('Order ID', axis=1, inplace=True)
```

1.0.2 POPULATE TEST DATA

[349]:
```
# function that takes in a single_data customer's primary product
# and returns a secondary product based on the product bundle orders
def get_secondary_product(primary_product):
    primary_product_bundles = [bundle for bundle in product_bundles if␣
 ↪primary_product in bundle]
    random_bundle = random.choice(primary_product_bundles)
    if random_bundle[0] == primary_product:
        return random_bundle[1]
    else:
        return random_bundle[0]
```

[350]:
```
# test get_secondary_product function
get_secondary_product('iPhone')
```

[350]:
```
'Lightning Charging Cable'
```

[351]:
```
# test for nulls in 'single_data'
any(single_data.isnull())
```

[351]:
```
True
```

[352]:
```
# rename 'single_data' to 'test_data'
test_data = single_data.copy()
```

[353]:
```
# drop NaN values
test_data.dropna(inplace=True)
```

[354]:
```
# create product list of unique products
product_list = list(set([p for b in product_bundles for p in b]))
```

[355]:
```
# print product list
product_list
```

[355]:
```
['Bose SoundSport Headphones',
 'Google Phone',
 'Flatscreen TV',
 'LG Dryer',
 'iPhone',
 '27in 4K Gaming Monitor',
 'Apple Airpods Headphones',
 '27in FHD Monitor',
 '34in Ultrawide Monitor',
 'Macbook Pro Laptop',
 'AAA Batteries (4-pack)',
 'AA Batteries (4-pack)',
 'Lightning Charging Cable',
 'Vareebadd Phone',
 'LG Washing Machine',
 'USB-C Charging Cable',
 'Wired Headphones',
 'ThinkPad Laptop',
 '20in Monitor']
```

[356]:
```
# check that unique products are the same between test data
# primary products and prouduct bundle data
set(test_data['Primary Product'].unique()) == set(product_list)
```

[356]:
```
True
```

[357]:
```
# remove bundles where the customer purchased the same product
# twice
product_bundles = [b for b in product_bundles if b[0] != b[1]]
```

[358]:
```
# add secondary product to test_data
test_data['Secondary Product'] = test_data['Primary Product'].apply(lambda␣
 ↪primary_product: get_secondary_product(primary_product))
```

[359]:
```
# check for any na in 'Primary Product' column
any(test_data.isna()['Primary Product'])
```

[359]:
```
False
```

[360]:
```
# check for any na in 'Secondary Product' column
any(test_data.isna()['Secondary Product'])
```

[360]:
```
False
```

[361]:
```
test_data_list = test_data.values.tolist()
```

1.0.3 PROBABILITY OF PURCHASE TABLE

[362]:
```
# calculates the perctages likelihood of
# purchase of secondary product given primary
# product
def calculate_percentages(product_name):
    percentage_list = []

    # calculate all bundles containing product 'product_name'
    product_name_bundles = []
    for row in product_bundles:
        if product_name in row:
            product_name_bundles.append(row)

    # iterate through product list
    for other_product in product_list:
        if product_name == other_product:
            percentage_list.append(0)
        else:
            # calculate count of 'other_product' appearing
            # in 'product_name_bundles'
            other_prod_bundle_count = 0
            for row in product_name_bundles:
                if other_product in row:
                    other_prod_bundle_count += 1
            percentage_list.append(other_prod_bundle_count)

    # calculate percentage likelihood of purhase
    return [bc/len(product_name_bundles) for bc in percentage_list]
```

[363]:
```
# calculate the condictional probability of purchasing each
# product in the list given the primary product is an
# iPhone
list(zip(product_list,calculate_percentages('iPhone')))
```

[363]:
```
[('Bose SoundSport Headphones', 0.006417736289381563),
 ('Google Phone', 0.003500583430571762),
 ('Flatscreen TV', 0.005250875145857643),
 ('LG Dryer', 0.0),
 ('iPhone', 0.0),
 ('27in 4K Gaming Monitor', 0.004667444574095682),
 ('Apple Airpods Headphones', 0.17444574095682613),
 ('27in FHD Monitor', 0.0011668611435239206),
 ('34in Ultrawide Monitor', 0.007001166861143524),
 ('Macbook Pro Laptop', 0.003500583430571762),
 ('AAA Batteries (4-pack)', 0.01575262543757293),
 ('AA Batteries (4-pack)', 0.01691948658109685),
```

```
 ('Lightning Charging Cable', 0.5198366394399067),
 ('Vareebadd Phone', 0.002333722287047841),
 ('LG Washing Machine', 0.0011668611435239206),
 ('USB-C Charging Cable', 0.014002333722287048),
 ('Wired Headphones', 0.21820303383897316),
 ('ThinkPad Laptop', 0.004084014002333722),
 ('20in Monitor', 0.001750291715285881)]
```

[364]:
```
# show that the conditional probabilities sum
# to one
sum(calculate_percentages('iPhone'))
```

[364]:
```
1.0
```

[365]:
```
# calculate columns of percentages table
prob_list = []
for product_name in product_list:
    prob_list.append(calculate_percentages(product_name))
```

[366]:
```
# create Python dictionary of product name keys and
# percentage list values
prob_dict = dict(zip(product_list, prob_list))
```

[367]:
```
# conditinoal product bundle probability table pandas
# dataframe creation
cpbpt = pd.DataFrame(prob_dict)
```

[368]:
```
# create 'Product' column to serve as row indexes
cpbpt['Product'] = product_list
```

[369]:
```
# set 'Product' column as row index
cpbpt.set_index('Product', inplace=True)
```

[370]:
```
# conditional product bundle probability table
# sums to one for each column
cpbpt.sum()
```

[370]:
```
Bose SoundSport Headphones    1.0
Google Phone                  1.0
Flatscreen TV                 1.0
LG Dryer                      1.0
iPhone                        1.0
27in 4K Gaming Monitor        1.0
Apple Airpods Headphones      1.0
27in FHD Monitor              1.0
34in Ultrawide Monitor        1.0
Macbook Pro Laptop            1.0
```

```
AAA Batteries (4-pack)        1.0
AA Batteries (4-pack)         1.0
Lightning Charging Cable      1.0
Vareebadd Phone               1.0
LG Washing Machine            1.0
USB-C Charging Cable          1.0
Wired Headphones              1.0
ThinkPad Laptop               1.0
20in Monitor                  1.0
dtype: float64
```

1.0.4 TOTAL BUNDLE REVENUE FOR EACH BUNDLE

[371]:
```python
# calculates bundles total revenue for each bundle
# containing product 'product_name'
def get_bundle_revenue(product_name):
    product_price = price_dict[product_name]
    price_list = []
    for other_product in product_list:
        if product_name == other_product:
            price_list.append(0)
        else:
            other_price = price_dict[other_product]
            price_list.append(product_price + other_price)
    return price_list
```

[372]:
```python
# shows bundle revenue with iPhone and listed secondary
# product
list(zip(product_list, get_bundle_revenue('iPhone')))
```

[372]:
```
[('Bose SoundSport Headphones', 799.99),
 ('Google Phone', 1300.0),
 ('Flatscreen TV', 1000.0),
 ('LG Dryer', 1300.0),
 ('iPhone', 0),
 ('27in 4K Gaming Monitor', 1089.99),
 ('Apple Airpods Headphones', 850.0),
 ('27in FHD Monitor', 849.99),
 ('34in Ultrawide Monitor', 1079.99),
 ('Macbook Pro Laptop', 2400.0),
 ('AAA Batteries (4-pack)', 702.99),
 ('AA Batteries (4-pack)', 703.84),
 ('Lightning Charging Cable', 714.95),
```

```
 ('Vareebadd Phone', 1100.0),
 ('LG Washing Machine', 1300.0),
 ('USB-C Charging Cable', 711.95),
 ('Wired Headphones', 711.99),
 ('ThinkPad Laptop', 1699.99),
 ('20in Monitor', 809.99)]
```

[373]:
```
# creates list of lists of product bundle
# revenues for product bundle
revenue_list = []
for product in product_list:
    revenue_list.append(get_bundle_revenue(product))
```

[374]:
```
# creates Python dictionary of products as keys and
# revenue lists as values
revenue_dict = dict(zip(product_list, revenue_list))
```

[375]:
```
# create Product Bundle Revenue Table pandas dataframe of product bundle␣
 ↪revenues
pbrt = pd.DataFrame(revenue_dict)
```

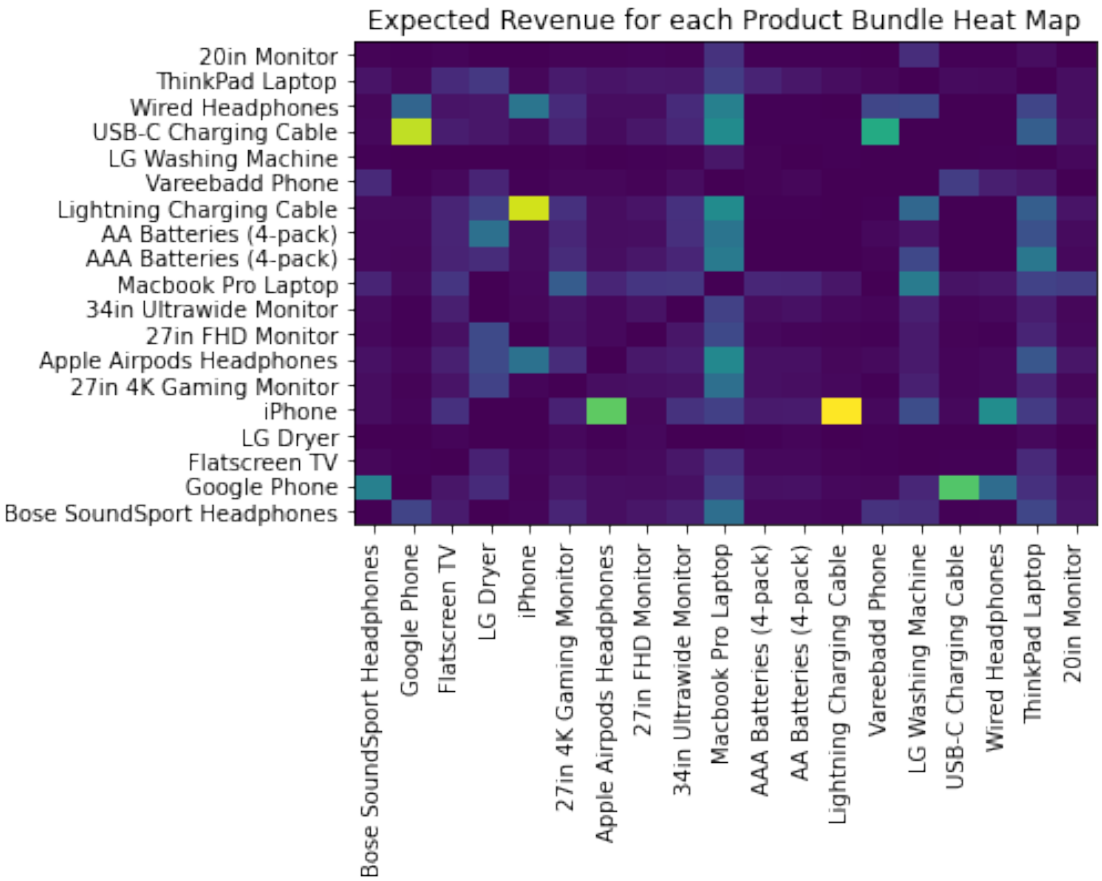
[376]:
```
# creates 'Product' column for row indexes
pbrt['Product'] = product_list
```

[377]:
```
# set 'Product' column as row indexes
pbrt = pbrt.set_index('Product')
```

1.0.5 EXPPECTED PRODUCT BUNDLE REVENUE TABLE

[378]:
```
# Expected Product Bundle Revenue Table
# multiplies entries in the revenue table
# and conditional probability table
epbrt = pbrt * cpbpt
```

[394]:
```
# plots heat map of expected revenue for each product bundle
# each row is a primary product and the column is the secondary
# product
fig=plt.figure()
plt.pcolor(epbrt)
plt.yticks(np.arange(0.5, len(epbrt.index), 1), epbrt.index)
plt.xticks(np.arange(0.5, len(epbrt.columns), 1), epbrt.columns,␣
 ↪rotation='vertical')
plt.title('Expected Revenue for each Product Bundle Heat Map')
plt.show()
fig.savefig('exp_bundle_rev.png', dpi=300, bbox_inches='tight')
```



1.0.6 PRODUCT RECOMMENDERS

[380]:
```
# general multiple product recommender that takes in
# either the expect revenue table or probability table
# to make 'num_recommendations' recommendations
def product_recommender(table, product_name, num_recommendations):
    series = table.loc[:, product_name]
    return list(series.nlargest(num_recommendations).index.values)
```

[381]:
```
# test multiple product recommender with expected revenue
# table as input and primary product 'USB-C Charging Cable'
primary_product = 'USB-C Charging Cable'
product_recommender(epbrt, primary_product, 3)
```

[381]:
```
['Google Phone', 'Vareebadd Phone', 'Macbook Pro Laptop']
```

[382]:
```
# test multiple product recommender with expected revenue
# table as input
product_recommender(cpbpt, primary_product, 3)
```

[382]: ['Google Phone', 'Vareebadd Phone', 'Lightning Charging Cable']

1.0.7 ALGOIRTHM SIMULATION FUNCTION

[383]:
```
# Simulates the purchasing or not purchasing of recommended
# products for the entire test dataset of customers
def product_recs_sim(table, num_recs):
    total_revenue = 0
    correct_count = 0
    for b in test_data_list:
        primary_product = b[0]
        rec_prod_list = product_recommender(table, primary_product, num_recs)
        for prod in rec_prod_list:
            if prod == b[1]:
                total_revenue += price_dict[primary_product] + price_dict[prod]
                correct_count += 1
                break
    return total_revenue, correct_count
```

1.0.8 ALGORITHM PERFORMANCE COMPARISON

[386]:
```
# function to build performance comparison table
def build_performance_table(n):
    num_recommendations = list(range(1, n+1))

    erm_rev_list = []
    pm_rev_list = []
    increase_rev = []
    percentage_increase_rev = []
    erm_rev_per_cust = []
    pm_rev_per_cust = []
    erm_num_orders = []
    pm_num_orders = []
    inc_rev_per_cust = []
    erm_cr = []
    pm_cr = []
    change_cr = []

    for i in range(1, n+1):

        erm_total_revenue, erm_correct_count = product_recs_sim(epbrt, i)
        erm_rev_list.append(erm_total_revenue)
        erm_rev_per_cust.append(erm_total_revenue/erm_correct_count)
        erm_cr.append(erm_correct_count/len(test_data_list))
        erm_num_orders.append(erm_correct_count)

        pm_total_revenue, pm_correct_count = product_recs_sim(cpbpt, i)
```

```
        pm_rev_list.append(pm_total_revenue)
        pm_rev_per_cust.append(pm_total_revenue/pm_correct_count)
        pm_cr.append(pm_correct_count/len(test_data_list))
        pm_num_orders.append(pm_correct_count)

        increase_rev.append(erm_total_revenue - pm_total_revenue)
        percentage_increase_rev.append(erm_total_revenue/pm_total_revenue - 1)
        inc_rev_per_cust.append(erm_rev_per_cust[-1]-pm_rev_per_cust[-1])
        change_cr.append(erm_cr[-1]-pm_cr[-1])

    col_names = ['# of Recommendations', 'ERM Revenue', 'PM Revenue',
                 'Increase in Revenue', '% Increase in Revenue', 'Avg. Order␣
↪Value (ERM)',
                 'Avg. Order Value (PM)', 'Increase Revenue Per Customer', 'ERM␣
↪Conv. Rate',
                 'PM Conv. Rate', 'Change Conv. Rate', 'ERM # Orders', 'PM #␣
↪Orders']

    cols = [num_recommendations, erm_rev_list, pm_rev_list, increase_rev,␣
↪percentage_increase_rev,
            erm_rev_per_cust, pm_rev_per_cust, inc_rev_per_cust, erm_cr, pm_cr,␣
↪change_cr, erm_num_orders,
            pm_num_orders]

    perf_dict = dict(zip(col_names, cols))
    return pd.DataFrame(perf_dict)
```

[387]:
```
# build performance comparison table
performance_table = build_performance_table(5)
```

1.0.9 COUNTS

[392]:
```
# number of single purchase test customers
len(test_data_list)
```

[392]: 171301

[393]:
```
# number of multi-purchase product bundles
len(product_bundles)
```

[393]: 6474