CSC 240 – Project 3: Market Basket Analysis Using Apriori Algorithm

March 3, 2023

Sammy Potter (spott14@u.rochester.edu)
Justin Lee (jlee363@u.rochester.edu)
CSC 240

1. Initial Approach

Our goal for this project was to mine frequent itemsets from web data through Market Basket Analysis using the Apriori algorithm. We can use these rules to provide data-driven recommendations for a user based on their search query.

We began the project by finding libraries we thought would be helpful. First, we decided to import the python library "pandas," which gave us the ability to read in CSVs and format them into DataFrames, which are 2-dimensional data structures that are easily mutable.

In addition, we found the python library "mlxtend," which includes tools that helped us extract transaction data from a DataFrame, calculate association rules from a dataset, and run the Apriori algorithm on the extracted data.

We imported the mentioned libraries as such:

SEGMENT I: IMPORTING LIBRARIES

```
import pandas as pd
from mlxtend.frequent_patterns import apriori
from mlxtend.frequent_patterns import association_rules
from mlxtend.preprocessing import TransactionEncoder
```

---

2. Data Pre-processing

To begin taking the given data and converting it into a minable format, we first read in the CSV and formatted it into a pandas DataFrame:

SEGMENT II: FORMATTING AS PANDAS DATAFRAME

```
df = pd.read_csv("anonymous-msweb.data", skiprows = 301, header = None)
```

The parameter "skiprows=301" makes the read_csv function ignore the first 301 rows. This is because the first 301 rows of the given data are attribute lines that only provide information about a website's relative URL and its attribute ID number, none of which include information about transactions.

Next, we looped through the DataFrame, which only included "C" (case) or "V" (vote) lines that told us which user visited which websites. In the context of Market Basket Analysis, we interpreted each user and the websites they visited as unique transactions.

SEGMENT III: LOOPING THROUGH DATAFRAME

```
current_id = None
bar_length = 30
progress_interval = int(len(df) / bar_length)
bar_count = 0
print("Processing...")
for index, row in df.iterrows():
  if index % progress_interval == 0:
    print(f"\r{'█' * bar_count}{' ' * (bar_length - bar_count)}| {round(bar_count /
bar_length * 100, 1)}%", end="", flush=True)
    bar_count += 1
```

```
    if row[0] == 'C':
        current_id = row[2]
        df.drop(index, inplace = True)
    else:
        df.at[index, 2] = current_id
print("\n")
```

This loop does multiple things:

1. Populate the third column of each vote line with the relevant case ID.
2. Drop each case line from the table, leaving only vote lines.
3. Update and display a process bar.

After the loop, we drop the column containing the "C" and "V" labels:

SEGMENT IV: DROPPING COLUMN FROM DATAFRAME

```
df.rename(columns = {0:'type', 1:'item', 2:'user'}, inplace = True)
df.drop('type', axis = 1, inplace = True)
```

Next, we utilized the TransactionEncoder in order to extract the transaction data from our current DataFrame and format it for the Apriori function:

SEGMENT V: UTILIZING TRANSACTION ENCODER

```
encoder = TransactionEncoder()
transactions = pd.DataFrame(encoder.fit(df).transform(df), columns = encoder.columns_)
```

---

3. Problems We Encountered

One issue we encountered was time complexity. Pre-processing close to 150,000 lines of data each time we wanted to test our code was time consuming and tedious. We decided to switch to a python notebook on Google Colab in order to avoid re-processing the data every time. This allowed us to see how the results of the Apriori algorithm changed based on our changes to its parameters.

Another issue we encountered was choosing good values for the Apriori function's parameters, which included the minimum support threshold and confidence. Our settings yielded 568 rules. We ultimately decided on min_support = 0.01 and min_threshold = 1 because we wanted our REPL system to have a better chance of having a recommendation to make for the majority of the pages. Even if the lift value of the recommendation was low, we still wanted to be able to recommend something.

4. Results

To run the Apriori algorithm on our pre-processed data, we run the following:

SEGMENT VI: RUNNING APRIORI

```
frequent_itemsets = apriori(transactions, min_support = 0.01, use_colnames = True)
rules = association_rules(frequent_itemsets, metric = "lift",  min_threshold = 1)
```

With our values of min_support and min_threshold (lift), we generated a total of 568 rules. Given our REPL system that takes in a user's query and returns the most significant rule (code shown below), we wanted a majority of the websites to be associated with a rule, as having a weaker recommendation is still better than having no recommendation at all.

SEGMENT VII: REPL

```
def nameFromId(d, qi):
```

```
    return d.loc[d['id'] == int(qi)].iloc[0]['path']

query = ""
while query != "quit":
    query = input("Enter site name: ")
    if query == "quit": continue
    try:
        query_id = names.loc[names['path'] == query].iloc[0]['id']
        results = rules.loc[rules['antecedents'].apply(lambda x: query_id in x)]
        if results.empty:
            print("No recommendation")
        else:
            highest = results.loc[results['lift'].idxmax()]
            res = []
            for x in highest['consequents']:
                res.append(nameFromId(names, x))
            print(f"Recommendation: {', '.join(res)} (lift: {round(highest['lift'],3)})")
    except IndexError:
        print(f"Invalid site name: {query}")
```

This code runs a REPL that, given an input N from the user, returns a relevant set of pages as a recommendation. Using the rules that we generated, we extract all rules that contain N in their antecedent set. We then choose the rule with the highest lift, and output its consequent set. Below is an example of the program output:

SEGMENT VII: EXAMPLE PROGRAM OUTPUT

```
Enter site name: /games
Recommendation: /products (lift: 1.559)
Enter site name: /windows
Recommendation: /msdownload, /windowssupport (lift: 16.629)
Enter site name: /msdownload
Recommendation: /windows, /isapi (lift: 16.629)
```

Enter site name: quit

The recommendations generated based on our queries are related. Through our testing of other inputs, we have found that for most inputs, our rules produce interesting results.

3. Appex

We include both the project's code in its entirety below, as well as a link to the project's Google Colab here:

SEGMENT VII: FULL CODE

```
import pandas as pd
from mlxtend.frequent_patterns import apriori
from mlxtend.frequent_patterns import association_rules
from mlxtend.preprocessing import TransactionEncoder

df = pd.read_csv("anonymous-msweb.data", skiprows = 301, header = None)

current_id = None
bar_length = 30
progress_interval = int(len(df) / bar_length)
bar_count = 0
print("Processing...")
for index, row in df.iterrows():
  if index % progress_interval == 0:
    print(f"\r{'█' * bar_count}{' ' * (bar_length - bar_count)}| {round(bar_count /
bar_length * 100, 1)}%", end="", flush=True)
    bar_count += 1
  if row[0] == 'C':
    current_id = row[2]
    df.drop(index, inplace = True)
  else:
    df.at[index, 2] = current_id
```

```
print("\n")

df.rename(columns = {0:'type', 1:'item', 2:'user'}, inplace = True)
df.drop('type', axis = 1, inplace = True)

df.item = df.item.transform(lambda x: [x])
df = df.groupby(['user']).sum()['item'].reset_index(drop = True)

encoder = TransactionEncoder()
transactions = pd.DataFrame(encoder.fit(df).transform(df), columns = encoder.columns_)

frequent_itemsets = apriori(transactions, min_support = 0.01, use_colnames = True)
rules = association_rules(frequent_itemsets, metric = "lift",  min_threshold = 1)
print(f"Total rules: {len(rules)}")

def nameFromId(d, qi):
    return d.loc[d['id'] == int(qi)].iloc[0]['path']

names = pd.read_csv("anonymous-msweb.data", skiprows=7, nrows=293, header = None)

names.rename(columns = {0:'type', 1:'id', 2:'N', 3:'name', 4:'path'}, inplace = True)
names.drop(['type', 'N'], axis = 1, inplace = True)

query = ""
while query != "quit":
    query = input("Enter site name: ")
    if query == "quit": continue
    try:
        query_id = names.loc[names['path'] == query].iloc[0]['id']
        results = rules.loc[rules['antecedents'].apply(lambda x: query_id in x)]
        if results.empty:
            print("No recommendation")
        else:
```

```
        highest = results.loc[results['lift'].idxmax()]
        res = []
        for x in highest['consequents']:
            res.append(nameFromId(names, x))
        print(f"Recommendation: {', '.join(res)} (lift: {round(highest['lift'],3)})")
    except IndexError:
        print(f"Invalid site name: {query}")
```
