songjiang_liu_midtermproj

October 13, 2024

1 FA24-CS634101 Data Mining

1.1 Midterm Project Report

Professor: Yasser Abduallah

Student: Songjiang Liu

UCID: sl947

GitHub Repository: https://github.com/youjustlook/CS634_songjiang_liu_midtermproj

1.1.1 Introduction

This report shows the step by step implementation of algorithms used for generating request items sets and association rules for transactions from 5 databases. The databases are modified from given sample records or generated by ChatGPT as demonstrative data. These algorithms are useful for generating shopping recommendations, finding concurrent patterns, etc. for many industry applications.

About Support and Confidence Support is the proportion of transactions in a dataset that contain a particular itemset, calculated as as the ratio of the number of transactions containing the itemset to the total number of transactions: **Support**:

$$\text{Support}(A) = \frac{\text{Transactions containing } A}{\text{Total transactions}}$$

Confidence is Confidence is the likelihood that an item appears in a transaction given that another item or itemset is already present, calculated as the ratio of the number of transactions containing both the antecedent and the consequent to the number of transactions containing only the antecedent: **Confidence**:

$$\text{Confidence}(A \to B) = \frac{\text{Transactions containing both } A \text{ and } B}{\text{Transactions containing } A}$$

About Algorithms Implemented/Used ##### self-implemented brute force algorithm The first algorithm in this project is a self-implemented brute force algorithm to find frequent item set and generate association rules from transactions at given support and confidence with enumeration for all possible combinations from item set of 1 to k items in the set. ##### built-in Apriori algorithm The second algorithm is the built-in Apriori algorithm in apriori_python package. It does the same thing as the self-implemented brute force algorithm but is optimized for efficiency

too. ##### built-in FP-Growth algorithm The third algorithm is the built-in FP-Growth algorithm in mlxtend package. It first constructs an FP-tree to represent the transaction database, and then recursively mining the tree to extract frequent itemsets by identifying conditional patterns and combining them to generate larger frequent itemsets without the need for multiple database scans.

1.1.2 Code Implementation

It takes selection of the databases, self-determined support level and confidence level, and finds the frequent items sets and association rules under the aforementioned requirement. In the end it present the execution time of three algorithms: student-built brute force algorithm, built-in Apriori algorithm, and built-in FP-Growth algorithm

Run the following code (delete # in the front first for the codes to be effective) to install necessary package if not found in environment.

[1]:
```
# !pip install pandas
# !pip install numpy
# !pip install apriori_python
# !pip install mlxtend
```

Step 1: Import and Load Dataset

[2]:
```
# import libraries
import pandas as pd
import numpy as np
from apriori_python.apriori import apriori
from mlxtend.frequent_patterns import fpgrowth
from mlxtend.frequent_patterns import association_rules
from mlxtend.preprocessing import TransactionEncoder
import itertools
import time

print("Hello, welcome to the Apriori Algorithm. Version 1.0")

def load_database(choice):
    databases = {
        1: "Amazon.csv",
        2: "BestBuy.csv",
        3: "HomeDepot.csv",
        4: "K-mart.csv",
        5: "Nike.csv"
    }
    try:
        return pd.read_csv(databases[choice])
    except KeyError:
        print("Invalid choice. Please enter a number between 1 and 5.")
        return None
    except FileNotFoundError:
        print(f"File {databases[choice]} not found.")
        return None

def get_item_set(transactions_data):
    item_set = set()
    for transaction in transactions_data.iloc[:, 1]:
        items = transaction.split(', ')
        item_set.update(items)
    return item_set

# Load the database at choice
databases = ["Amazon", "BestBuy", "HomeDepot", "K-mart", "Nike"]

while True:
    print("Please select a database to load:")
    for i, db in enumerate(databases, 1):
        print(f"{i}. {db}")

    try:
        choice = int(input("Enter the number of your choice: "))
        if choice < 1 or choice > len(databases):
            raise ValueError("Invalid input. Please enter a number between 1␣
↪and 5.")

        transactions_data = load_database(choice)

        if transactions_data is None:
            continue  # if loading the database fails, ask for input again

        transactions_data_each = transactions_data.iloc[:, 1]
        print(f"Loaded {databases[choice-1]} database successfully.")

        # Display number of transactions
        num_transactions = len(transactions_data)
        print(f"Number of transactions: {num_transactions}")

        # Display item set information
        item_set = get_item_set(transactions_data)
        num_items = len(item_set)
        print(f"Number of items in the item set: {num_items}")
        print("Items in the item set:")
        for item in item_set:
            print(item)

        break  # exit the loop if everything is successful

    except ValueError as e:
        print(e)
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
```

```
Hello, welcome to the Apriori Algorithm. Version 1.0
Please select a database to load:
1. Amazon
2. BestBuy
3. HomeDepot
4. K-mart
5. Nike

Enter the number of your choice:  5

Loaded Nike database successfully.
Number of transactions: 20
Number of items in the item set: 10
Items in the item set:
Soccer Shoe
Socks
Modern Pants
Dry Fit V-Nick
Tech Pants
Hoodies
Running Shoe
Sweatshirts
Rash Guard
Swimming Shirt
```

Step 2: User Input for Support and Confidence

[3]:
```
# Get user input for minimum support and minimum confidence
while True:
    try:
        min_support = float(input("Enter the minimum support level in % (larger␣
 ↪than 0 and lower than 100%): "))
        if 0 < min_support < 100:
            break
        else:
            print("Invalid input. Please enter a value between 0 and 100.")
    except ValueError:
        print("Invalid input. Please enter a numeric value.")

while True:
    try:
        min_confidence = float(input("Enter the minimum confidence level in %␣
 ↪(larger than 0 and lower than 100%): "))
        if 0 < min_confidence < 100:
            break
        else:
            print("Invalid input. Please enter a value between 0 and 100.")
    except ValueError:
        print("Invalid input. Please enter a numeric value.")
```

```
Enter the minimum support level in % (larger than 0 and lower than 100%):  50
Enter the minimum confidence level in % (larger than 0 and lower than 100%):  40
```

Step 3: Brute Force Method

[4]:
```
# Brute force method
def generate_candidates(item_set, length):
    return [set(item) for item in itertools.combinations(item_set, length)]

def is_frequent(candidate, transactions, min_support_count):
    count = sum(1 for transaction in transactions if candidate.
 ↪issubset(transaction))
    return count >= min_support_count

def count_frequent(candidate, transactions):
    count = sum(1 for transaction in transactions if candidate.
 ↪issubset(transaction))
    return count

def brute_force_apriori(transactions, item_set, min_support):
    min_support_count = len(transactions) * min_support
    frequent_itemsets = []
    k = 1
    current_itemsets = [set([item]) for item in item_set]

    while current_itemsets:
        next_itemsets = []
        for itemset in current_itemsets:
            if is_frequent(itemset, transactions, min_support_count):
                frequent_itemsets.append(itemset)
        k += 1
        current_itemsets = generate_candidates(set(itertools.chain.
 ↪from_iterable(frequent_itemsets)), k)

    return frequent_itemsets

def find_association_rules(frequent_collections, data_samples,␣
 ↪threshold_confidence):
    association_rules = []
    for collection in frequent_collections:
        for num_elements in range(1, len(collection)):
            for precursor in itertools.combinations(collection, num_elements):
                precursor = set(precursor)
                outcome = collection - precursor
                if outcome:
                    support = sum(1 for sample in data_samples if collection.
 ↪issubset(sample)) / len(data_samples)
                    confidence = support / (sum(1 for sample in data_samples if␣
 ↪precursor.issubset(sample)) / len(data_samples))
                    if confidence >= threshold_confidence:
                        association_rules.append((precursor, outcome, support,␣
 ↪confidence))
    return association_rules

# Running the brute force method
transactions_data_processed = []
order = sorted(item_set)
for lines in transactions_data_each:
    trans = list(lines.strip().split(', '))
    trans_l = list(np.unique(trans))
    trans_l.sort(key=lambda x: order.index(x))
    transactions_data_processed.append(sorted(trans_l))

start_time_brute_force_freq_itemset = time.time_ns()
frequent_itemsets = brute_force_apriori(transactions_data_processed, item_set,␣
 ↪min_support / 100)
brute_force_time_freq_itemset = time.time_ns() -␣
 ↪start_time_brute_force_freq_itemset

print("--------------------\nFrequent item sets using brute force method: item␣
 ↪| support")
for itemset in frequent_itemsets:
    print(str(itemset) + " | " + str(count_frequent(itemset,␣
 ↪transactions_data_processed)/len(transactions_data_processed)))


print("--------------------\nAssociation rules using brute force method:")
start_time_self_association_rules = time.time_ns()
rules = find_association_rules(frequent_itemsets, transactions_data_processed,␣
 ↪min_confidence / 100)
find_rules_time = time.time_ns() - start_time_self_association_rules
for i, rule in enumerate(rules):
    antecedent, consequent, support, confidence = rule
    print(f"Rule {i+1}: {antecedent} -> {consequent} (support: {support},␣
 ↪confidence: {confidence})")
```

```
--------------------
Frequent item sets using brute force method: item | support
{'Socks'} | 0.65
{'Modern Pants'} | 0.5
```

```
{'Dry Fit V-Nick'} | 0.5
{'Running Shoe'} | 0.7
{'Sweatshirts'} | 0.65
{'Rash Guard'} | 0.6
{'Swimming Shirt'} | 0.55
{'Rash Guard', 'Dry Fit V-Nick'} | 0.5
{'Running Shoe', 'Socks'} | 0.55
{'Sweatshirts', 'Socks'} | 0.6
{'Modern Pants', 'Sweatshirts'} | 0.5
{'Running Shoe', 'Sweatshirts'} | 0.55
{'Rash Guard', 'Swimming Shirt'} | 0.5
{'Running Shoe', 'Sweatshirts', 'Socks'} | 0.5
--------------------
Association rules using brute force method:
Rule 1: {'Rash Guard'} -> {'Dry Fit V-Nick'} (support: 0.5, confidence:
0.8333333333333334)
Rule 2: {'Dry Fit V-Nick'} -> {'Rash Guard'} (support: 0.5, confidence: 1.0)
Rule 3: {'Running Shoe'} -> {'Socks'} (support: 0.55, confidence:
0.7857142857142858)
Rule 4: {'Socks'} -> {'Running Shoe'} (support: 0.55, confidence:
0.8461538461538461)
Rule 5: {'Sweatshirts'} -> {'Socks'} (support: 0.6, confidence:
0.923076923076923)
Rule 6: {'Socks'} -> {'Sweatshirts'} (support: 0.6, confidence:
0.923076923076923)
Rule 7: {'Modern Pants'} -> {'Sweatshirts'} (support: 0.5, confidence: 1.0)
Rule 8: {'Sweatshirts'} -> {'Modern Pants'} (support: 0.5, confidence:
0.7692307692307692)
Rule 9: {'Running Shoe'} -> {'Sweatshirts'} (support: 0.55, confidence:
0.7857142857142858)
Rule 10: {'Sweatshirts'} -> {'Running Shoe'} (support: 0.55, confidence:
0.8461538461538461)
Rule 11: {'Rash Guard'} -> {'Swimming Shirt'} (support: 0.5, confidence:
0.8333333333333334)
Rule 12: {'Swimming Shirt'} -> {'Rash Guard'} (support: 0.5, confidence:
0.9090909090909091)
Rule 13: {'Running Shoe'} -> {'Sweatshirts', 'Socks'} (support: 0.5, confidence:
0.7142857142857143)
Rule 14: {'Sweatshirts'} -> {'Running Shoe', 'Socks'} (support: 0.5, confidence:
0.7692307692307692)
Rule 15: {'Socks'} -> {'Running Shoe', 'Sweatshirts'} (support: 0.5, confidence:
0.7692307692307692)
Rule 16: {'Running Shoe', 'Sweatshirts'} -> {'Socks'} (support: 0.5, confidence:
0.9090909090909091)
Rule 17: {'Running Shoe', 'Socks'} -> {'Sweatshirts'} (support: 0.5, confidence:
0.9090909090909091)
Rule 18: {'Sweatshirts', 'Socks'} -> {'Running Shoe'} (support: 0.5, confidence:
0.8333333333333334)
```

Step 4: Built-in Apriori Algorithm

```
[5]: # Check with built-in Apriori algorithm
print("--------------------\nCheck with Built-in Apriori Algorithm:")
start_apriori_time_freq_itemset_and_rules = time.time_ns()
freq_item_set, rules = apriori(transactions_data_processed, min_support/100,␣
 ↪min_confidence/100)
apriori_time_freq_itemset_and_rules = time.time_ns() -␣
 ↪start_apriori_time_freq_itemset_and_rules

print("--------------------\nFrequent item sets with Built-in Apriori algorithm:
 ↪")
print(freq_item_set)

print("--------------------\nAssociation rules with Built-in Apriori algorithm:
 ↪")
for i, rule in enumerate(rules):
    print(f"Rule {i+1}: {rule}")
```

```
--------------------
Check with Built-in Apriori Algorithm:
--------------------
Frequent item sets with Built-in Apriori algorithm:
{1: {frozenset({'Running Shoe'}), frozenset({'Modern Pants'}),
frozenset({'Swimming Shirt'}), frozenset({'Dry Fit V-Nick'}),
frozenset({'Socks'}), frozenset({'Rash Guard'}), frozenset({'Sweatshirts'})}, 2:
{frozenset({'Sweatshirts', 'Socks'}), frozenset({'Running Shoe', 'Socks'}),
frozenset({'Running Shoe', 'Sweatshirts'}), frozenset({'Rash Guard', 'Swimming
Shirt'}), frozenset({'Rash Guard', 'Dry Fit V-Nick'}), frozenset({'Modern
Pants', 'Sweatshirts'})}, 3: {frozenset({'Running Shoe', 'Sweatshirts',
'Socks'})}}
--------------------
Association rules with Built-in Apriori algorithm:
Rule 1: [{'Running Shoe'}, {'Sweatshirts', 'Socks'}, 0.7142857142857143]
Rule 2: [{'Sweatshirts'}, {'Modern Pants'}, 0.7692307692307693]
Rule 3: [{'Sweatshirts'}, {'Running Shoe', 'Socks'}, 0.7692307692307693]
Rule 4: [{'Socks'}, {'Running Shoe', 'Sweatshirts'}, 0.7692307692307693]
Rule 5: [{'Running Shoe'}, {'Socks'}, 0.7857142857142857]
Rule 6: [{'Running Shoe'}, {'Sweatshirts'}, 0.7857142857142857]
Rule 7: [{'Rash Guard'}, {'Swimming Shirt'}, 0.8333333333333334]
Rule 8: [{'Rash Guard'}, {'Dry Fit V-Nick'}, 0.8333333333333334]
Rule 9: [{'Sweatshirts', 'Socks'}, {'Running Shoe'}, 0.8333333333333334]
Rule 10: [{'Socks'}, {'Running Shoe'}, 0.8461538461538461]
Rule 11: [{'Sweatshirts'}, {'Running Shoe'}, 0.8461538461538461]
Rule 12: [{'Swimming Shirt'}, {'Rash Guard'}, 0.9090909090909091]
Rule 13: [{'Running Shoe', 'Sweatshirts'}, {'Socks'}, 0.9090909090909091]
Rule 14: [{'Running Shoe', 'Socks'}, {'Sweatshirts'}, 0.9090909090909091]
Rule 15: [{'Sweatshirts'}, {'Socks'}, 0.9230769230769231]
Rule 16: [{'Socks'}, {'Sweatshirts'}, 0.9230769230769231]
Rule 17: [{'Dry Fit V-Nick'}, {'Rash Guard'}, 1.0]
Rule 18: [{'Modern Pants'}, {'Sweatshirts'}, 1.0]
```

The self-built brute force algorithm generates same frequent item sets andd association rules as the built-in Apriori algotithm.

Step 5: FP-Growth Algorithm

[6]:
```python
# Check with fp-growth algorithm
te = TransactionEncoder()
te_ary = te.fit(transactions_data_processed).
  ↪transform(transactions_data_processed)
df_encoded = pd.DataFrame(te_ary, columns=te.columns_)

start_time_fpgrowth = time.time_ns()
freq_item_set_fp = fpgrowth(df_encoded, min_support/100, use_colnames=True)
fpgrowth_time = time.time_ns() - start_time_fpgrowth

print("--------------------\nCheck with Built-in FP-Growth algorithm:")
print(freq_item_set_fp)

start_time_association_rules = time.time_ns()
rules_fp = association_rules(freq_item_set_fp, metric="confidence",␣
  ↪min_threshold=min_confidence / 100)
association_rules_time = time.time_ns() - start_time_association_rules

print("--------------------\nAssociation rules with FP-Growth algorithm: rule,␣
  ↪support, confidence")
for idx, row in rules_fp.iterrows():
    antecedent = set(row['antecedents'])
    consequent = set(row['consequents'])
    support = row['support']
    confidence = row['confidence']
    print(f"Rule {idx + 1}: {antecedent} -> {consequent} (support: {support:.
  ↪2f}, confidence: {confidence:.2f})")
```

```
--------------------
Check with Built-in FP-Growth algorithm:
    support                      itemsets
0      0.70                (Running Shoe)
1      0.65                 (Sweatshirts)
2      0.65                       (Socks)
3      0.50                (Modern Pants)
4      0.60                  (Rash Guard)
5      0.55              (Swimming Shirt)
6      0.50              (Dry Fit V-Nick)
7      0.55   (Running Shoe, Sweatshirts)
8      0.60          (Sweatshirts, Socks)
```

```
9      0.55               (Running Shoe, Socks)
10     0.50  (Running Shoe, Sweatshirts, Socks)
11     0.50         (Modern Pants, Sweatshirts)
12     0.50        (Rash Guard, Swimming Shirt)
13     0.50        (Rash Guard, Dry Fit V-Nick)
--------------------
Association rules with FP-Growth algorithm: rule, support, confidence
Rule 1: {'Running Shoe'} -> {'Sweatshirts'} (support: 0.55, confidence: 0.79)
Rule 2: {'Sweatshirts'} -> {'Running Shoe'} (support: 0.55, confidence: 0.85)
Rule 3: {'Sweatshirts'} -> {'Socks'} (support: 0.60, confidence: 0.92)
Rule 4: {'Socks'} -> {'Sweatshirts'} (support: 0.60, confidence: 0.92)
Rule 5: {'Running Shoe'} -> {'Socks'} (support: 0.55, confidence: 0.79)
Rule 6: {'Socks'} -> {'Running Shoe'} (support: 0.55, confidence: 0.85)
Rule 7: {'Running Shoe', 'Sweatshirts'} -> {'Socks'} (support: 0.50, confidence:
0.91)
Rule 8: {'Running Shoe', 'Socks'} -> {'Sweatshirts'} (support: 0.50, confidence:
0.91)
Rule 9: {'Sweatshirts', 'Socks'} -> {'Running Shoe'} (support: 0.50, confidence:
0.83)
Rule 10: {'Running Shoe'} -> {'Sweatshirts', 'Socks'} (support: 0.50,
confidence: 0.71)
Rule 11: {'Sweatshirts'} -> {'Running Shoe', 'Socks'} (support: 0.50,
confidence: 0.77)
Rule 12: {'Socks'} -> {'Running Shoe', 'Sweatshirts'} (support: 0.50,
confidence: 0.77)
Rule 13: {'Modern Pants'} -> {'Sweatshirts'} (support: 0.50, confidence: 1.00)
Rule 14: {'Sweatshirts'} -> {'Modern Pants'} (support: 0.50, confidence: 0.77)
Rule 15: {'Rash Guard'} -> {'Swimming Shirt'} (support: 0.50, confidence: 0.83)
Rule 16: {'Swimming Shirt'} -> {'Rash Guard'} (support: 0.50, confidence: 0.91)
Rule 17: {'Rash Guard'} -> {'Dry Fit V-Nick'} (support: 0.50, confidence: 0.83)
Rule 18: {'Dry Fit V-Nick'} -> {'Rash Guard'} (support: 0.50, confidence: 1.00)
```

The the results from brute force, apriori, and fp-growth are the same.

[7]:
```python
# Print combined execution times
print("\nExecution Times (in nanoseconds):")
print(f"Self-Built Brute Force: {brute_force_time_freq_itemset +␣
 ↪find_rules_time} ns")
print(f"Built-in Apriori: {apriori_time_freq_itemset_and_rules} ns")
print(f"Built-In FP-Growth: {fpgrowth_time + association_rules_time} ns")
print("--------------------\nFootnote:\n Built-in Apriori returns frequent␣
 ↪item sets and rules in one go, while the other two generate it separately.␣
 ↪\n And the databases only have 20 transactions. \n Therefore, sometimes␣
 ↪Built-in Apriori took the least time, and sometimes FP-Growth is even slower␣
 ↪than self-built brute force. ")
```

```
Execution Times (in nanoseconds):
```

```
Self-Built Brute Force: 2002600 ns
Built-in Apriori: 1003500 ns
Built-In FP-Growth: 5005500 ns
---------------------
Footnote:
 Built-in Apriori returns frequent item sets and rules in one go, while the
other two generate it separately.
 And the databases only have 20 transactions.
 Therefore, sometimes Built-in Apriori took the least time, and sometimes FP-
Growth is even slower than self-built brute force.
```

1.1.3 Conclusion

This project guides you to understand the use and functioning steps of typical association rule algorithms: one self-built brute force algorithm, which is a de facto Apriori algorithm, one optimized-efficient built-in Apriori algorithm, and one more scale-efficient FP-Growth algorim. The frequent items sets and association rules generated are matched with each other; however, their running time are diffent and can vary depending on scale of dataset and their own methodology behind.