CS 634 Data Mining Midterm Term Project -- Tap47

1. Introduction

This notebook provides a full overview of the Apriori algorithm implementation and application in association rule mining. The source.py file provides a custom implementation of the Apriori algorithm, and the Jupyter notebook shows how to apply this implementation to a dataset.

2. Abstract

This report presents a comprehensive analysis of the Apriori algorithm, a foundational method for association rule mining used to identify relationships among items in transactional data. The primary objective was to implement the Apriori algorithm from scratch and apply it to a dataset. The analysis involved the development of a custom apriori class that calculates support, generates frequent itemsets, and derives association rules.

3. Apriori Algorithm Implementation (source.py)

The apriori class is defined in the source.py file, and it has methods for estimating support, creating frequent itemsets, and finding association rules. The class and its functions are described in detail below.

3.1

```
__init__(self, series)
  Purpose: Initializes the class with a series of transactions.
  Parameter: series – A pandas Series containing transaction data (each entry is a list of items).
```

3.2 calculate_support(self, items)

This function computes the support of an itemset, which is the percentage of transactions containing all the items in that set.

```
def calculate_support(self, items):
    transactions = self.series
    counts = np.array([0] * len(self.series))
    for item in items:
        counts += transactions.str.count(item)
    return (np.where(counts >= len(items), 1, 0).sum() / len(transactions)) * 100
```

3.3 get_unique_items(self, transactions=None)

This method extracts all unique items from the transactions.

```
def get_unique_items(self, transactions=None):
    if not transactions:
        transactions = self.series
    items = np.array([])
    for transaction in transactions:
        items = np.append(items, transaction.replace(" ", "").split(","))
    return set(items)
```

3.4 generate_item_sets(self, items, k=0)

Generates combinations of items of size k.

```
def generate_item_sets(self, items, k=0):
    return list(itertools.combinations(items, r=k))
```

3.5 get_freq_items(self, min_support, items=None)

Finds all frequent itemsets that have support greater than or equal to min_support.

```
def get_freq_items(self, min_support, items=None):
    if not items:
```

```
            items = self.get_unique_items()
        count = 1
        freq_items = []
        while True:
            temp = []
            for i in self.generate_item_sets(items, count):
                support = self.calculate_support(i)
                if support >= min_support:
                    print(f"Support for item-set {i}: {support}")
                    temp.append(i)
            if len(temp) > 0:
                freq_items += temp
                count += 1
            else:
                break
        return freq_items
```

3.6 get_associations(self, freq_items, min_conf)

Generates association rules based on the frequent itemsets and a minimum confidence threshold.

```
    def get_associations(self, freq_items, min_conf):
        temp = []
        for _ in freq_items:
            if len(_) == 1:
                continue
            for items in itertools.permutations(_):
                try:
                    conf = (self.calculate_support(items) / self.calculate_support(items[:-1])) * 100
                except:
                    continue
                if conf >= min_conf and conf <= 100:
                    temp.append([items, conf])
        print("Associations:", temp)
        return temp
```

4. Applying and comparing the Apriori Algorithm to existing algorithm in the Jupyter Notebook

The notebook (tap47.ipynb) explains how the Apriori algorithm is used on a dataset. This is how the process works:

4.1 Dataset loading The dataset for analysis is imported from a CSV file, and transactions are renamed for clarity.

```
import pandas as pd
import numpy as np
import itertools
from source import apriori
df = pd.read_csv("data/assignment1.csv")
df = df.rename(columns={"Transaction": "Items"})
```

4.2 Preprocessing with One-Hot Encoding

One-hot encoding is implemented using the MultiLabelBinarizer from sklearn.preprocessing. Each transaction is divided into separate elements, and a binary representation is generated.

```
from sklearn.preprocessing import MultiLabelBinarizer

# Preprocessing: splitting items and applying one-hot encoding
test = [i.replace(" ", "").split(",") for i in np.array(df["Items"])]
mlb = MultiLabelBinarizer()
res = pd.DataFrame(mlb.fit_transform(test),
                   columns=mlb.classes_)
```

4.3 Compared my algorithm using external packages (efficient-apriori, mlxtend).

4.3.1 Efficient-algorithm

```
from efficient_apriori import apriori as appp
itemsets, rules = appp([i.replace(" ", "").split(",") for i in np.array(df["Items"])] , min_support=0
rules
```

4.3.2 mlxtend apriori

```
from mlxtend.frequent_patterns import apriori as fpgrowth, association_rules, apriori as appriori
frequent_itemsets = appriori(res, min_support=0.5, use_colnames= True)
rules = association_rules(frequent_itemsets, metric="confidence", min_threshold=0.5)
rules
```

4.3.3 mlxtend fp growth tree

```
frequent_itemsets = fpgrowth(res, min_support=0.5, use_colnames= True)
rules = association_rules(frequent_itemsets, metric="confidence", min_threshold=0.5)
rules
```

5. Conclusion

In this study, I successfully developed the Apriori algorithm for association rule mining, demonstrating its usefulness in identifying associations in transactional data. The bespoke implementation, wrapped in the apriori class in source.py, offered a thorough grasp of the algorithm's mechanics, including support and confidence calculations.