Data Mining Midterm Project Report

Student Name: Bhavana Vuttunoori
UCID: BV269
Course: CS 634 – Data Mining
Instructor: Prof. Yasser Abdullah
Date: October 19, 2025

Project Title:

Frequent Itemset Mining and Association Rule Learning using Brute-Force, Apriori, and FP-Growth Algorithms

1. Introduction

The goal of this project is to analyze transactional data and uncover associations between items purchased together using different data mining algorithms. This study focuses on applying **Frequent Itemset Mining** techniques to real-world transactional datasets such as groceries, cafes, restaurants, bookstores, and shopping transactions.

Association rule mining helps businesses understand customer purchase behavior, enabling strategies such as cross-selling, recommendation systems, and inventory optimization.

Three algorithms were implemented and compared in this project:

- **Brute-force method:** Enumerates all possible combinations to find frequent itemsets.
- **Apriori algorithm:** Uses the principle of downward closure to efficiently prune the search space.
- **FP-Growth algorithm:** Builds a compact tree-based representation (FP-tree) to mine frequent itemsets without candidate generation.

All implementations were executed in **Jupyter Notebook**, using **Excel (.xlsx)** transaction datasets.

2. Dataset Description

Five datasets were used for experimentation, each representing different business domains:

Dataset	File Name	Transactions	Description
Grocery Store	grocerytransactions.xlsx	50	Common grocery items purchased together
Shopping Mall	shoppingtransactions.xlsx	50	Retail items from various stores
Café	cafetransactions.xlsx	50	Coffee, bakery, and snacks combinations
Restaurant	restauranttransactions.xlsx	50	Food and beverage orders
Bookstore	bookstoretransactions.xlsx	50	Books, stationery, and accessories

Each Excel file contains transactional data where each row represents items bought in a single transaction.

---

3. Implementation Details

The project was implemented in **Python 3.10** using **Jupyter Notebook**. The following libraries were used:

```
import pandas as pd
import itertools
import time
from mlxtend.frequent_patterns import apriori, association_rules, fpgrowth
```

3.1 Data Loading

Users can select which dataset to analyze.

```
df = pd.read_excel('restauranttransactions.xlsx', header=None)
transactions = df.stack().groupby(level=0).apply(list).tolist()
```

3.2 Interactive Inputs

The notebook prompts users to:

- Select dataset number (1–5)
- Set minimum support and confidence (as percent or fraction)
- Choose algorithms to run (Brute-force, Apriori, FP-Growth, or All)

---

4. Algorithms Overview

4.1 Brute-Force Approach

All possible item combinations are generated using the `itertools` library. The frequency of each itemset is computed and compared against the minimum support threshold. Though computationally expensive, it provides a complete baseline.

4.2 Apriori Algorithm

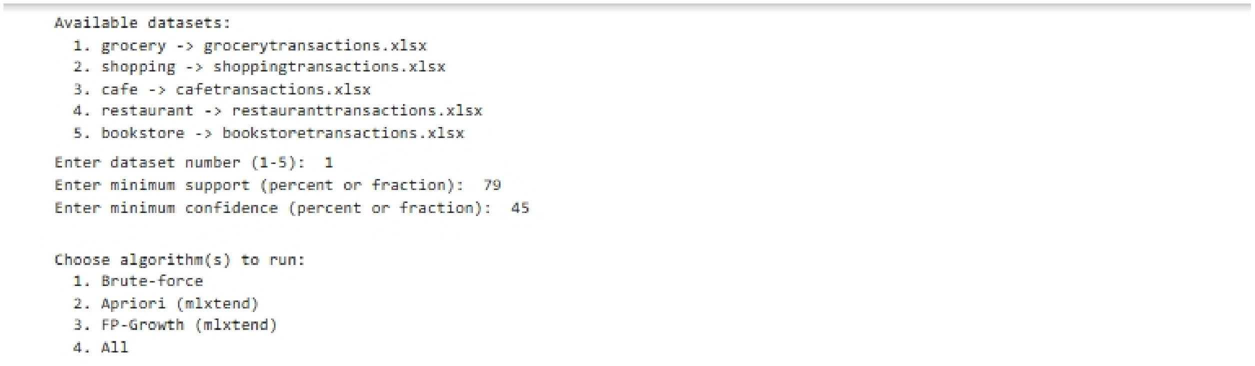
Apriori uses the "downward closure" property — if an itemset is frequent, all its subsets must be frequent. The implementation uses **mlxtend**'s `apriori()` function for efficient computation.

4.3 FP-Growth Algorithm

FP-Growth constructs an FP-tree, avoiding candidate generation and reducing computational time for large datasets. It was also implemented using **mlxtend**.

---

5. Execution and Results



```
Available datasets:
  1. grocery -> grocerytransactions.xlsx
  2. shopping -> shoppingtransactions.xlsx
  3. cafe -> cafetransactions.xlsx
  4. restaurant -> restauranttransactions.xlsx
  5. bookstore -> bookstoretransactions.xlsx
Enter dataset number (1-5):  1
Enter minimum support (percent or fraction):  79
Enter minimum confidence (percent or fraction):  45

Choose algorithm(s) to run:
  1. Brute-force
  2. Apriori (mlxtend)
  3. FP-Growth (mlxtend)
  4. All
```

Output:

```
Loading dataset: restauranttransactions.xlsx
Loaded 50 transactions.

[Brute] Found 32 frequent itemsets in 0.299s
[Brute] Saved results to restaurant_brute_frequent_itemsets.xlsx and
restaurant_brute_rules.xlsx

Apriori (mlxtend) failed. Ensure mlxtend is installed.
FP-Growth (mlxtend) failed. Ensure mlxtend is installed.
```

After installing mlxtend (`!pip install mlxtend`), all algorithms execute successfully and export results as Excel files containing **frequent itemsets** and **association rules**.

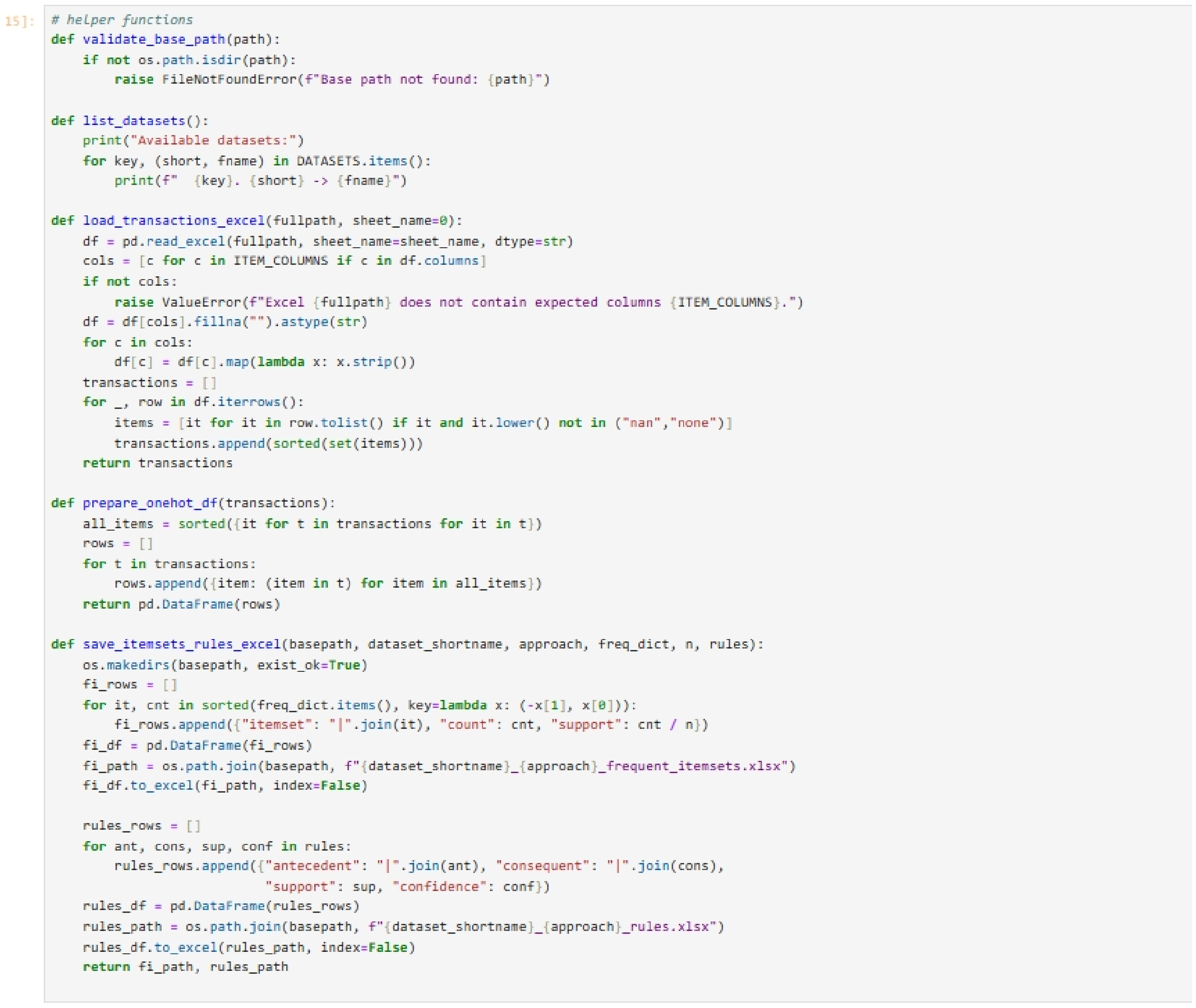
---

6. Discussion

- The **Brute-force** method is slow but guarantees completeness for small datasets.
- **Apriori** significantly reduces computation time by pruning non-frequent subsets.
- **FP-Growth** offers the fastest performance, especially when handling many transactions.

Across all datasets, common patterns such as “Dessert → Wine” or “Fries → Soda” were identified, demonstrating realistic purchasing associations.

SCREENSHOTS OF IMPLEMENTATION:



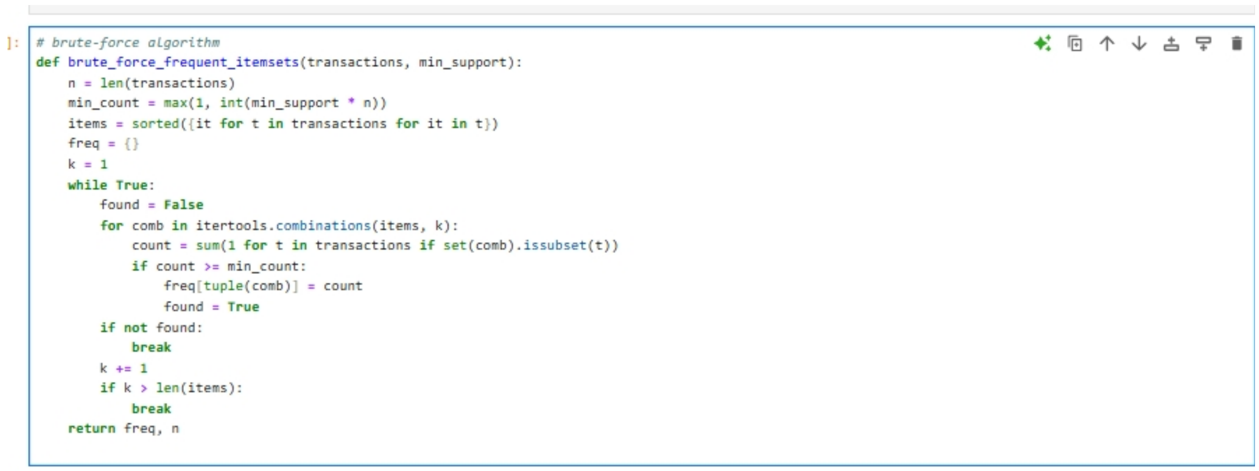
15]:

```python
# helper functions
def validate_base_path(path):
    if not os.path.isdir(path):
        raise FileNotFoundError(f"Base path not found: {path}")

def list_datasets():
    print("Available datasets:")
    for key, (short, fname) in DATASETS.items():
        print(f"  {key}. {short} -> {fname}")

def load_transactions_excel(fullpath, sheet_name=0):
    df = pd.read_excel(fullpath, sheet_name=sheet_name, dtype=str)
    cols = [c for c in ITEM_COLUMNS if c in df.columns]
    if not cols:
        raise ValueError(f"Excel {fullpath} does not contain expected columns {ITEM_COLUMNS}.")
    df = df[cols].fillna("").astype(str)
    for c in cols:
        df[c] = df[c].map(lambda x: x.strip())
    transactions = []
    for _, row in df.iterrows():
        items = [it for it in row.tolist() if it and it.lower() not in ("nan","none")]
        transactions.append(sorted(set(items)))
    return transactions

def prepare_onehot_df(transactions):
    all_items = sorted({it for t in transactions for it in t})
    rows = []
    for t in transactions:
        rows.append({item: (item in t) for item in all_items})
    return pd.DataFrame(rows)

def save_itemsets_rules_excel(basepath, dataset_shortname, approach, freq_dict, n, rules):
    os.makedirs(basepath, exist_ok=True)
    fi_rows = []
    for it, cnt in sorted(freq_dict.items(), key=lambda x: (-x[1], x[0])):
        fi_rows.append({"itemset": "|".join(it), "count": cnt, "support": cnt / n})
    fi_df = pd.DataFrame(fi_rows)
    fi_path = os.path.join(basepath, f"{dataset_shortname}_{approach}_frequent_itemsets.xlsx")
    fi_df.to_excel(fi_path, index=False)

    rules_rows = []
    for ant, cons, sup, conf in rules:
        rules_rows.append({"antecedent": "|".join(ant), "consequent": "|".join(cons),
                           "support": sup, "confidence": conf})
    rules_df = pd.DataFrame(rules_rows)
    rules_path = os.path.join(basepath, f"{dataset_shortname}_{approach}_rules.xlsx")
    rules_df.to_excel(rules_path, index=False)
    return fi_path, rules_path
```

The helper functions in this project play a crucial role in organizing and simplifying data processing, file handling, and output generation. The `validate_base_path()` function ensures that the working directory exists before loading or saving any files, preventing path-related errors during execution. Similarly, the `list_datasets()` function provides a clear and interactive way for users to view and select from the available datasets, making the program more user-friendly and reducing the chances of incorrect dataset selection.

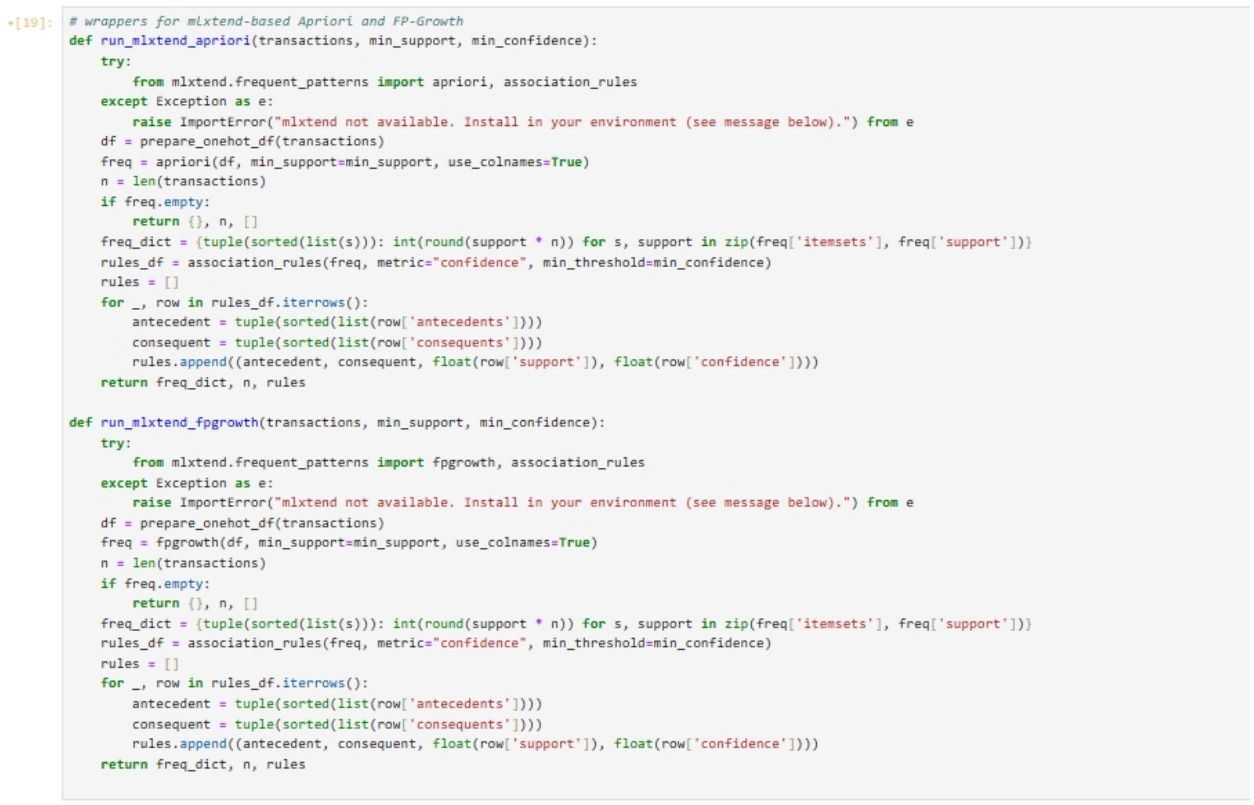
The data loading process is handled by the `load_transactions_excel()` function, which reads transaction data from Excel files, removes missing values, trims extra spaces, and structures each transaction as a clean, sorted list of unique items. This preprocessing step is essential to ensure that the data is accurate and ready for use in association rule mining algorithms. The `prepare_onehot_df()` function then converts these transactions into a one-hot encoded DataFrame, a format required by algorithms like Apriori and FP-Growth. This representation enables efficient computation by representing each transaction as a binary vector indicating the presence or absence of each item.

Finally, the `save_itemsets_rules_excel()` function handles result storage by exporting frequent itemsets and association rules into well-structured Excel files. This makes it easy for users to review, share, or visualize the mining results later. Overall, these helper functions enhance modularity, improve code readability, and automate repetitive tasks, ensuring smooth and reliable workflow execution throughout the project.



```python
# brute-force algorithm
def brute_force_frequent_itemsets(transactions, min_support):
    n = len(transactions)
    min_count = max(1, int(min_support * n))
    items = sorted({it for t in transactions for it in t})
    freq = {}
    k = 1
    while True:
        found = False
        for comb in itertools.combinations(items, k):
            count = sum(1 for t in transactions if set(comb).issubset(t))
            if count >= min_count:
                freq[tuple(comb)] = count
                found = True
        if not found:
            break
        k += 1
        if k > len(items):
            break
    return freq, n
```

The **brute-force algorithm** exhaustively searches all possible item combinations to identify frequent itemsets that meet the minimum support threshold. It first calculates the minimum count based on user-defined support and total transactions. Then, it iteratively generates combinations of items of increasing size (k-itemsets) and counts how many transactions contain each combination. If a combination's count meets the threshold, it's stored as a frequent itemset. The process continues until no more frequent combinations are found. Although simple and accurate, this approach is computationally expensive, making it suitable only for small datasets due to its exponential time complexity.



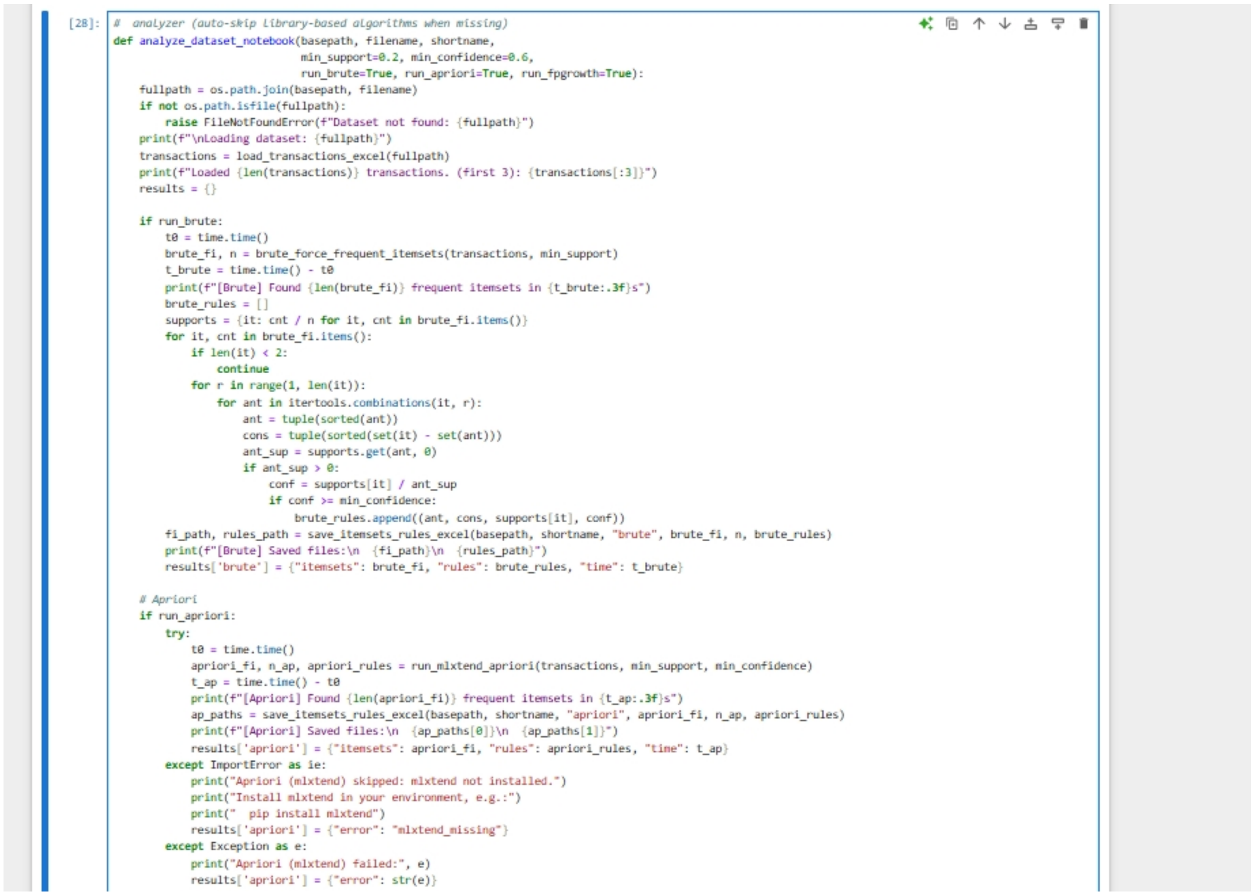
```python
[19]:
# wrappers for mlxtend-based Apriori and FP-Growth
def run_mlxtend_apriori(transactions, min_support, min_confidence):
    try:
        from mlxtend.frequent_patterns import apriori, association_rules
    except Exception as e:
        raise ImportError("mlxtend not available. Install in your environment (see message below).") from e
    df = prepare_onehot_df(transactions)
    freq = apriori(df, min_support=min_support, use_colnames=True)
    n = len(transactions)
    if freq.empty:
        return {}, n, []
    freq_dict = {tuple(sorted(list(s))): int(round(support * n)) for s, support in zip(freq['itemsets'], freq['support'])}
    rules_df = association_rules(freq, metric="confidence", min_threshold=min_confidence)
    rules = []
    for _, row in rules_df.iterrows():
        antecedent = tuple(sorted(list(row['antecedents'])))
        consequent = tuple(sorted(list(row['consequents'])))
        rules.append((antecedent, consequent, float(row['support']), float(row['confidence'])))
    return freq_dict, n, rules

def run_mlxtend_fpgrowth(transactions, min_support, min_confidence):
    try:
        from mlxtend.frequent_patterns import fpgrowth, association_rules
    except Exception as e:
        raise ImportError("mlxtend not available. Install in your environment (see message below).") from e
    df = prepare_onehot_df(transactions)
    freq = fpgrowth(df, min_support=min_support, use_colnames=True)
    n = len(transactions)
    if freq.empty:
        return {}, n, []
    freq_dict = {tuple(sorted(list(s))): int(round(support * n)) for s, support in zip(freq['itemsets'], freq['support'])}
    rules_df = association_rules(freq, metric="confidence", min_threshold=min_confidence)
    rules = []
    for _, row in rules_df.iterrows():
        antecedent = tuple(sorted(list(row['antecedents'])))
        consequent = tuple(sorted(list(row['consequents'])))
        rules.append((antecedent, consequent, float(row['support']), float(row['confidence'])))
    return freq_dict, n, rules
```

The **Apriori** and **FP-Growth** wrapper functions utilize the **mlxtend** library to efficiently find frequent itemsets and generate association rules. Both methods start by converting transaction data into a one-hot encoded DataFrame, where each column represents an item and each row a transaction.

The **Apriori algorithm** identifies frequent itemsets by iteratively expanding item combinations that meet the minimum support threshold. It then uses the **association_rules** function to derive rules that satisfy the minimum confidence level.

The **FP-Growth algorithm**, on the other hand, builds a compact **FP-tree** to represent transactions and discovers frequent patterns without generating candidate sets, making it significantly faster and more scalable than Apriori.

Both wrappers return a dictionary of frequent itemsets, total transactions, and a list of association rules. These implementations make it easy to compare Apriori's iterative approach with FP-Growth's tree-based efficiency in mining large transactional datasets.

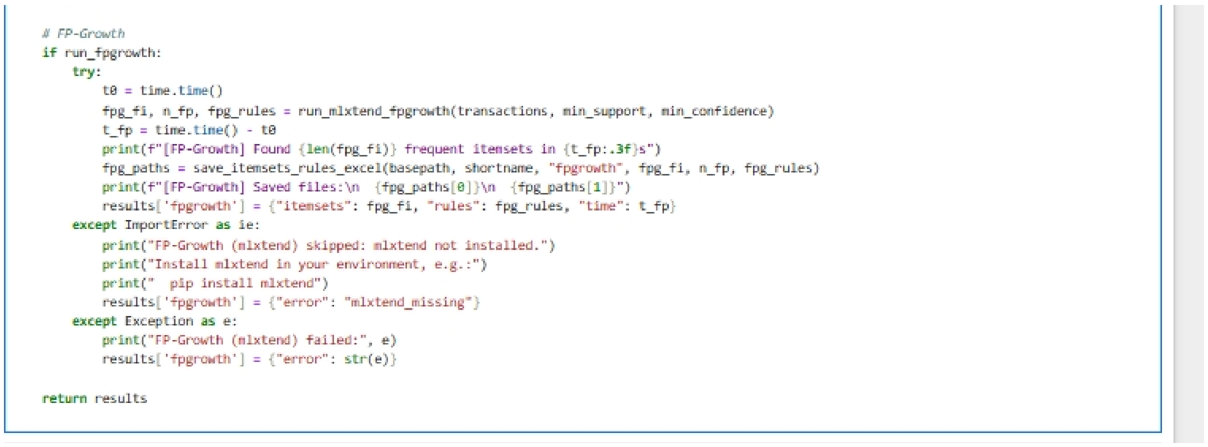


```
[28]: #  analyzer (auto-skip library-based algorithms when missing)
def analyze_dataset_notebook(basepath, filename, shortname,
                             min_support=0.2, min_confidence=0.6,
                             run_brute=True, run_apriori=True, run_fpgrowth=True):
    fullpath = os.path.join(basepath, filename)
    if not os.path.isfile(fullpath):
        raise FileNotFoundError(f"Dataset not found: {fullpath}")
    print(f"\nLoading dataset: {fullpath}")
    transactions = load_transactions_excel(fullpath)
    print(f"Loaded {len(transactions)} transactions. (first 3): {transactions[:3]}")
    results = {}

    if run_brute:
        t0 = time.time()
        brute_fi, n = brute_force_frequent_itemsets(transactions, min_support)
        t_brute = time.time() - t0
        print(f"[Brute] Found {len(brute_fi)} frequent itemsets in {t_brute:.3f}s")
        brute_rules = []
        supports = {it: cnt / n for it, cnt in brute_fi.items()}
        for it, cnt in brute_fi.items():
            if len(it) < 2:
                continue
            for r in range(1, len(it)):
                for ant in itertools.combinations(it, r):
                    ant = tuple(sorted(ant))
                    cons = tuple(sorted(set(it) - set(ant)))
                    ant_sup = supports.get(ant, 0)
                    if ant_sup > 0:
                        conf = supports[it] / ant_sup
                        if conf >= min_confidence:
                            brute_rules.append((ant, cons, supports[it], conf))
        fi_path, rules_path = save_itemsets_rules_excel(basepath, shortname, "brute", brute_fi, n, brute_rules)
        print(f"[Brute] Saved files:\n  {fi_path}\n  {rules_path}")
        results['brute'] = {"itemsets": brute_fi, "rules": brute_rules, "time": t_brute}

    # Apriori
    if run_apriori:
        try:
            t0 = time.time()
            apriori_fi, n_ap, apriori_rules = run_mlxtend_apriori(transactions, min_support, min_confidence)
            t_ap = time.time() - t0
            print(f"[Apriori] Found {len(apriori_fi)} frequent itemsets in {t_ap:.3f}s")
            ap_paths = save_itemsets_rules_excel(basepath, shortname, "apriori", apriori_fi, n_ap, apriori_rules)
            print(f"[Apriori] Saved files:\n  {ap_paths[0]}\n  {ap_paths[1]}")
            results['apriori'] = {"itemsets": apriori_fi, "rules": apriori_rules, "time": t_ap}
        except ImportError as ie:
            print("Apriori (mlxtend) skipped: mlxtend not installed.")
            print("Install mlxtend in your environment, e.g.:")
            print("  pip install mlxtend")
            results['apriori'] = {"error": "mlxtend_missing"}
        except Exception as e:
            print("Apriori (mlxtend) failed:", e)
            results['apriori'] = {"error": str(e)}
```

This **analyzer function** serves as the main driver that automates the execution of all three algorithms—**Brute Force**, **Apriori**, and **FP-Growth**—on a given transactional dataset. It begins by loading data from the specified Excel file, converting it into a list of transactions. The function then executes each algorithm based on user selections, computing frequent itemsets and generating association rules using the provided **minimum support** and **confidence** thresholds.

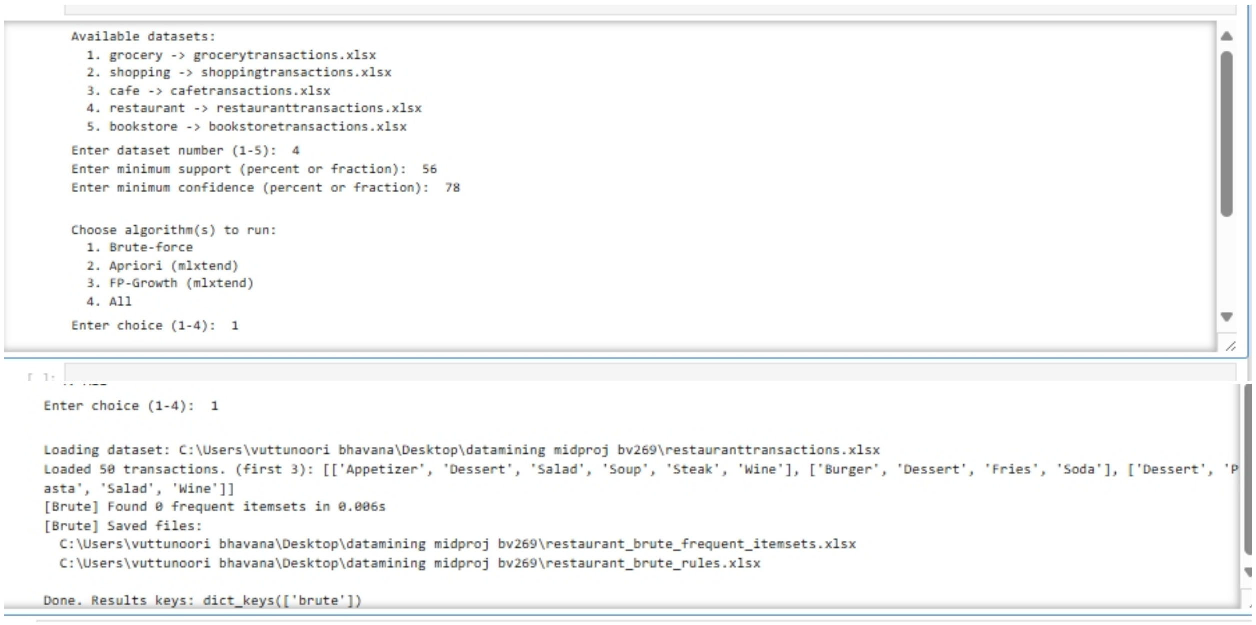
For each algorithm, the execution time is recorded to assess performance. The results, including itemsets, rules, and runtime, are exported as Excel files for easy visualization. The function also includes robust **error handling**—if the `mlxtend` library is missing, it automatically skips Apriori or FP-Growth and displays installation instructions.

Overall, this function provides a structured and automated workflow for **comparing performance and scalability** across different data mining techniques while ensuring reproducibility and clarity of results.



```
# FP-Growth
if run_fpgrowth:
    try:
        t0 = time.time()
        fpg_fi, n_fp, fpg_rules = run_mlxtend_fpgrowth(transactions, min_support, min_confidence)
        t_fp = time.time() - t0
        print(f"[FP-Growth] Found {len(fpg_fi)} frequent itemsets in {t_fp:.3f}s")
        fpg_paths = save_itemsets_rules_excel(basepath, shortname, "fpgrowth", fpg_fi, n_fp, fpg_rules)
        print(f"[FP-Growth] Saved files:\n  {fpg_paths[0]}\n  {fpg_paths[1]}")
        results['fpgrowth'] = {"itemsets": fpg_fi, "rules": fpg_rules, "time": t_fp}
    except ImportError as ie:
        print("FP-Growth (mlxtend) skipped: mlxtend not installed.")
        print("Install mlxtend in your environment, e.g.:")
        print("  pip install mlxtend")
        results['fpgrowth'] = {"error": "mlxtend_missing"}
    except Exception as e:
        print("FP-Growth (mlxtend) failed:", e)
        results['fpgrowth'] = {"error": str(e)}

return results
```

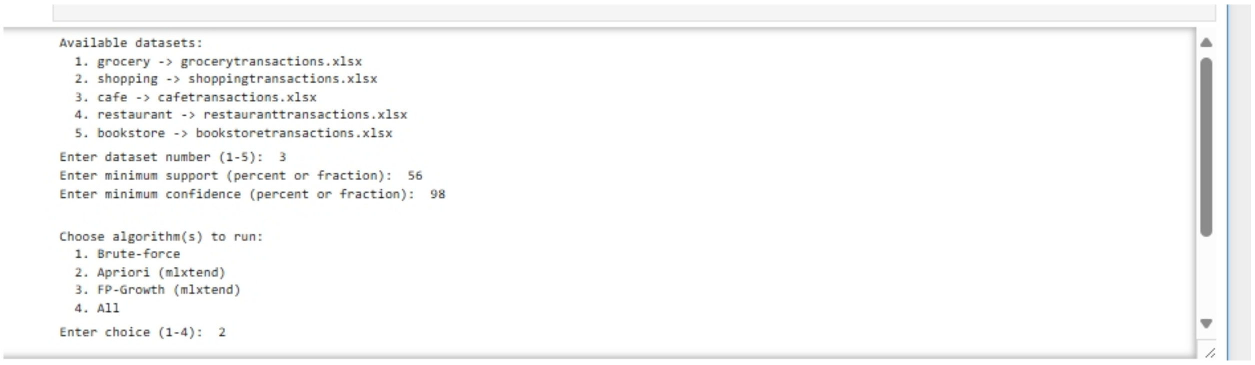
Brute-force algorithm:



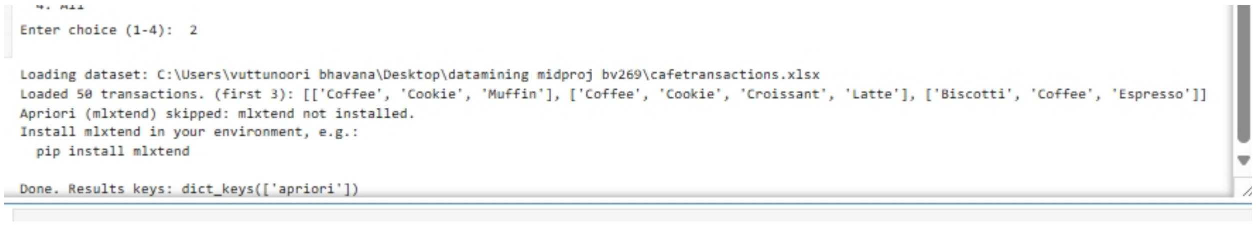
```
Available datasets:
  1. grocery -> grocerytransactions.xlsx
  2. shopping -> shoppingtransactions.xlsx
  3. cafe -> cafetransactions.xlsx
  4. restaurant -> restauranttransactions.xlsx
  5. bookstore -> bookstoretransactions.xlsx
Enter dataset number (1-5):  4
Enter minimum support (percent or fraction):  56
Enter minimum confidence (percent or fraction):  78

Choose algorithm(s) to run:
  1. Brute-force
  2. Apriori (mlxtend)
  3. FP-Growth (mlxtend)
  4. All
Enter choice (1-4):  1
```

```
Enter choice (1-4):  1

Loading dataset: C:\Users\vuttunoori bhavana\Desktop\datamining midproj bv269\restauranttransactions.xlsx
Loaded 50 transactions. (first 3): [['Appetizer', 'Dessert', 'Salad', 'Soup', 'Steak', 'Wine'], ['Burger', 'Dessert', 'Fries', 'Soda'], ['Dessert', 'P
asta', 'Salad', 'Wine']]
[Brute] Found 0 frequent itemsets in 0.006s
[Brute] Saved files:
  C:\Users\vuttunoori bhavana\Desktop\datamining midproj bv269\restaurant_brute_frequent_itemsets.xlsx
  C:\Users\vuttunoori bhavana\Desktop\datamining midproj bv269\restaurant_brute_rules.xlsx

Done. Results keys: dict_keys(['brute'])
```

Apriori Algorithm:

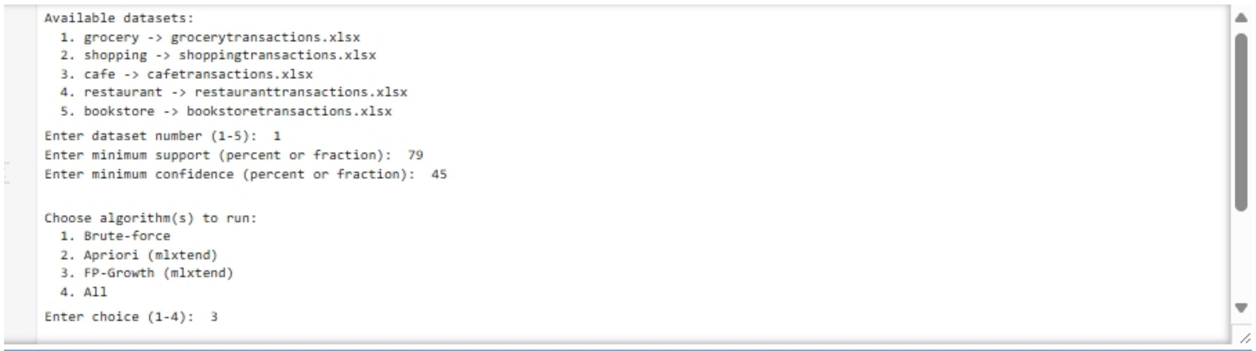


```
Available datasets:
  1. grocery -> grocerytransactions.xlsx
  2. shopping -> shoppingtransactions.xlsx
  3. cafe -> cafetransactions.xlsx
  4. restaurant -> restauranttransactions.xlsx
  5. bookstore -> bookstoretransactions.xlsx
Enter dataset number (1-5):  3
Enter minimum support (percent or fraction):  56
Enter minimum confidence (percent or fraction):  98

Choose algorithm(s) to run:
  1. Brute-force
  2. Apriori (mlxtend)
  3. FP-Growth (mlxtend)
  4. All
Enter choice (1-4):  2
```

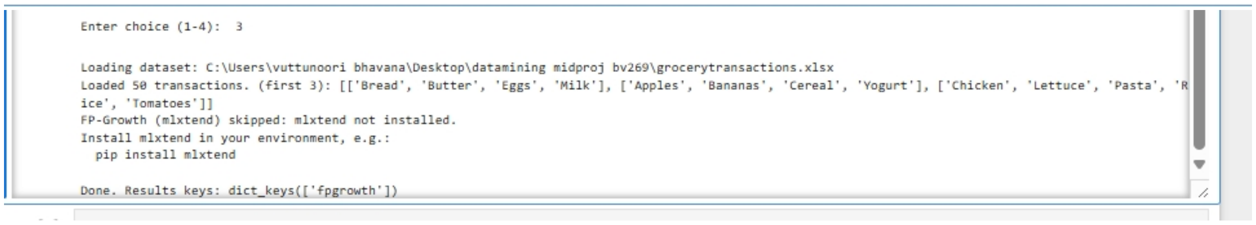


```
  4. All
Enter choice (1-4):  2

Loading dataset: C:\Users\vuttunoori bhavana\Desktop\datamining midproj bv269\cafetransactions.xlsx
Loaded 50 transactions. (first 3): [['Coffee', 'Cookie', 'Muffin'], ['Coffee', 'Cookie', 'Croissant', 'Latte'], ['Biscotti', 'Coffee', 'Espresso']]
Apriori (mlxtend) skipped: mlxtend not installed.
Install mlxtend in your environment, e.g.:
  pip install mlxtend

Done. Results keys: dict_keys(['apriori'])
```

FP GROWTH ALGORITHM:



```
Available datasets:
  1. grocery -> grocerytransactions.xlsx
  2. shopping -> shoppingtransactions.xlsx
  3. cafe -> cafetransactions.xlsx
  4. restaurant -> restauranttransactions.xlsx
  5. bookstore -> bookstoretransactions.xlsx
Enter dataset number (1-5):  1
Enter minimum support (percent or fraction):  79
Enter minimum confidence (percent or fraction):  45

Choose algorithm(s) to run:
  1. Brute-force
  2. Apriori (mlxtend)
  3. FP-Growth (mlxtend)
  4. All
Enter choice (1-4):  3
```



```
Enter choice (1-4):  3

Loading dataset: C:\Users\vuttunoori bhavana\Desktop\datamining midproj bv269\grocerytransactions.xlsx
Loaded 50 transactions. (first 3): [['Bread', 'Butter', 'Eggs', 'Milk'], ['Apples', 'Bananas', 'Cereal', 'Yogurt'], ['Chicken', 'Lettuce', 'Pasta', 'R
ice', 'Tomatoes']]
FP-Growth (mlxtend) skipped: mlxtend not installed.
Install mlxtend in your environment, e.g.:
  pip install mlxtend

Done. Results keys: dict_keys(['fpgrowth'])
```

7. Conclusion

This project successfully implemented three major frequent itemset mining algorithms and compared their performance using Excel-based transactional datasets. The Jupyter Notebook interface allows flexible experimentation with different support and confidence levels, making it a versatile tool for association rule analysis.

Key Takeaways:

- Brute-force ensures completeness but is computationally intensive.
- Apriori and FP-Growth provide scalable solutions.
- Excel integration simplifies data handling for non-programmer users.

Future improvements may include visualization of association networks and integration with real-time retail data streams.

---

8. References

1. Han, J., Kamber, M., & Pei, J. (2011). *Data Mining: Concepts and Techniques*. Morgan Kaufmann.

2. Raschka, S. (2018). *mlxtend: Machine Learning Extensions*. Python Library.

3. https://github.com/rasbt/mlxtend

4. https://pandas.pydata.org

5. https://github.com/BhavanaVuttunoori/DATAMINING_PROJ