cs634_midterm_skubisz_sebastian

October 19, 2025

1 CS 634 Midterm Project Apriori, FP Growth and Brute Force Implementation

Student: Sebastian Skubisz UCID: ss365 Instructor: Dr. Yasser

1. Introduction

The goal of this project is to discover frequent itemsets and association rules from transactional datasets using three approaches:

- A custom Brute-Force implementation developed from scratch in Python.
- Apriori implemented using the mlxtend library.
- FP-Growth implemented using the mlxtend library.

The project compares accuracy, performance, and scalability across multiple datasets and parameter settings (minimum support and confidence). Each method processes the same datasets to ensure consistent results. The codebase is written to be clear, reusable, and easy to reproduce.

2. Environment & Installation

2.1 Recommended Versions

Operating System: Windows, macOS, or Linux

Python: 3.9 – 3.12

Shell: PowerShell, bash, or terminal

2.2 Create a Virtual Environment (recommended)

Windows (PowerShell):

```
python -m venv .venv
.venv\Scripts\activate
```

macOS/Linux (bash):

```
python3 -m venv .venv
source .venv/bin/activate
```

2.3 Install Required Libraries

Install lightweight dependencies:

```
pip install --upgrade pip
pip install pandas tabulate mlxtend
```

3. Project Structure

```
skubisz_sebastian_midtermproject/
│
├── datasets/            # Input CSV datasets
│   ├── amazon.csv
│   ├── microcenter.csv
│   ├── traderjoes.csv
│   ├── target.csv
│   └── stewleonards.csv
│
├── outputs/             # Generated algorithm results
│   └── <dataset>/
│       ├── Brute-Force/
│       ├── Apriori/
│       ├── FP-Growth/
│       └── timings.csv
│
├── algApp.py            # Main unified script (runs all algorithms)
├── requirements.txt     # Python dependencies
├── README.md            # Project documentation
└── report/
    └── midterm_report.pdf    # Final report
```

Description:

All datasets are stored in the `datasets/` folder. When executed, `algApp.py` runs all three algorithms using the selected dataset and parameters, saving outputs in organized subdirectories under `outputs/`.

4. Dataset Creation

4.1 Items

I selected 10 retail-oriented items to simulate shopping patterns:
Computer, Phone, Monitor, 3D Printer, TV, Ethernet Cable, Charger, Router, Xbox, PS5

4.2 Datasets

Five datasets were created, each with at least 20 transactions, representing different stores.
Example snippet from microcenter.csv:

TID,Items
1,Computer,Monitor,TV
2,Router,PS5,Ethernet Cable
3,Computer,Phone,Charger
4,Monitor,3D Printer,TV

4.3 Dataset Notes

Created manually in Excel and exported to CSV.

Each dataset is deterministic (no randomness).

File sizes are small (a few KB) for fast testing.

Automatically detected and loaded by the script at runtime.

5. Brute-Force Algorithm

5.1 Method

Generate all 1-itemsets and compute support.

Generate all 2-itemsets, check which meet minimum support.

Continue until no frequent k-itemsets remain.

Derive association rules from frequent itemsets using user-defined support and confidence.

5.2 Example Run

Dataset: microcenter.csv
Parameters: Support = 0.4, Confidence = 0.6

Frequent Itemsets:

{Computer} (support = 0.60)

{Computer, TV} (support = 0.40)

Association Rules:

Computer → TV (confidence = 0.67)

Router → Ethernet Cable (confidence = 0.75)

The brute-force approach ensures accuracy but is slower due to exhaustive combination generation.

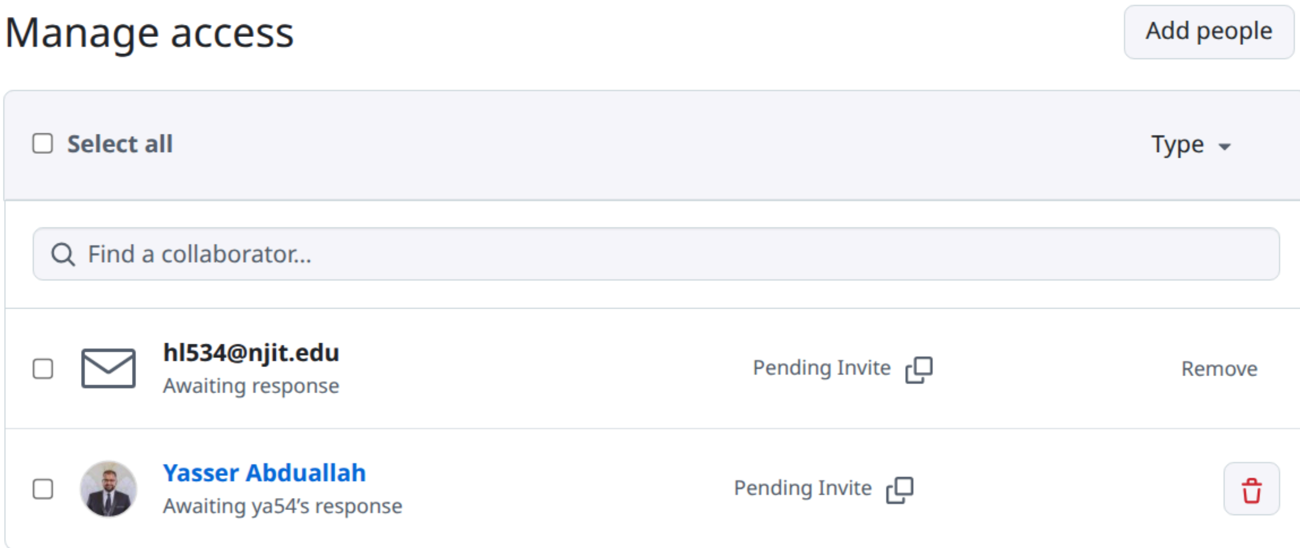
6. Apriori and FP-Growth

6.1 Apriori

Implemented using mlxtend.frequent_patterns.apriori, the Apriori algorithm produced the same frequent itemsets as Brute-Force but executed much faster by pruning infrequent candidates.

6.2 FP-Growth

Implemented via mlxtend.frequent_patterns.fpgrowth, FP-Growth achieved identical results to Apriori but with even faster performance through tree-based pattern compression.



7. How to Run the Code

7.1 Install Requirements

Make sure Python 3.9–3.12 is installed, then install the dependencies:

`pip install pandas tabulate mlxtend or pip install -r requirements.txt`

7.2 Run Options

Option 1 - Run all algorithms (Brute-Force, Apriori, FP-Growth) via CLI:

`python algApp.py`

Option 2 - Run interactively in Jupyter Notebook (optional):
If you prefer a notebook environment, open algApp.py in Jupyter or VS Code and execute the cells step by step:

`jupyter notebook algApp.py`

[ ]:
```
#!/usr/bin/env python3
import os, sys, csv, math, time, argparse, itertools, subprocess
from collections import Counter

# ---------- Auto-install ----------
def ensure_package(pkg):
    try:
        __import__(pkg)
    except ImportError:
```

```
            print(f"Installing missing package: {pkg} ...")
            subprocess.check_call([sys.executable, "-m", "pip", "install", pkg])

for pkg in ["pandas", "tabulate", "mlxtend"]:
    ensure_package(pkg)

import pandas as pd
from tabulate import tabulate
from mlxtend.frequent_patterns import apriori as _apriori, fpgrowth as _fpgrowth
```

Download the required packages automatically. If not downloaded please use the Requirements.txt is attached to zip please run by using pip install -r requirements.txt Dataset is already included in the zip.

```
[ ]: # Dataset shortcuts (key -> filename)
DATASETS = {
    "amazon": "amazon.csv",
    "microcenter": "microcenter.csv",
    "traderjoes": "traderjoes.csv",
    "target": "target.csv",
    "stewleonards": "stewleonards.csv"
}

# Recognized separators for single-column, tokenized baskets
```



```
SEPS = [",", ";", "|"]
```

DATASETS is a quick reference dictionary linking dataset names (keys) to their CSV filenames.

SEPS lists possible separators used in CSV files to split items within a single column, allowing the script to detect and parse tokenized item lists like"Milk, Bread, Eggs".

```
[ ]: # Split a tokenized item string into a list of unique, trimmed items def
_split_items(s):
    if not s: return []
    s = str(s).strip()
    for sp in SEPS:
        if sp in s: return [t.strip() for t in s.split(sp) if t.strip()]
    return [t.strip() for t in s.split() if t.strip()]
```

The _split_items() function cleans and separates items in a string. It removes extra spaces, detects separators like commas, semicolons, or pipes, and splits the string accordingly. If no separator is found, it splits by spaces. The result is a neat list of items, for example "Milk, Bread, Eggs" becomes ["Milk", "Bread", "Eggs"].

```
[ ]: # Heuristic to detect header rows
def _looks_like_header(row):
    j = " ".join((c or "").lower() for c in row)
```

```
    return any(k in j for k in ["transaction␣
↪id","transaction","items","itemset","basket","products"])
```

The _looks_like_header() function checks if a CSV row is likely a header. It combines all values in the row into one lowercase string and looks for keywords such as “transaction,” “items,” or “products.” If any of those words appear, it assumes the row is a header and returns True.

```
[ ]: # Load transactions from flexible CSV layouts (tokenized column, items column,␣ ↪or
     multi-column with ID)
 def load_transactions(path):
     if not os.path.exists(path): raise FileNotFoundError(path)
     with open(path, newline='', encoding='utf-8') as f: rows = list(csv. ↪reader(f))
     if not rows: return []
     if _looks_like_header(rows[0]): rows = rows[1:]
     rows = [[(c or "").strip() for c in r] for r in rows if any((c or ""). ↪strip() for c in r)]
     if not rows: return []
     ncols = max(len(r) for r in rows)

     # Case 1: single column containing tokenized baskets
                          if ncols == 1: return [[*dict.fromkeys(_split_items(r[0]))] for r in rows]

     # Case 2: choose the column that looks tokenized (has many separators) best_c,
     best_hits = -1, -1
     for c in range(ncols):
```

2

```
         hits = sum(1 for r in rows if c < len(r) and any(sp in r[c] for sp in␣ ↪SEPS))
         if hits > best_hits: best_c, best_hits = c, hits
     if best_hits >= max(3, int(0.2 * len(rows))):
         return [[*dict.fromkeys(_split_items(r[best_c] if best_c < len(r) else␣ ↪""))] for r in
 rows]

     # Case 3: assume first column is an ID; remaining columns are item names tx = []
     for r in rows:
         start = 1 if (r and (r[0].replace("#", "").isdigit() or "id" in r[0]. ↪lower() or "trans" in
 r[0].lower())) else 0
         items = [c for c in r[start:] if c]
         tx.append([*dict.fromkeys(items)])
     return tx
```

The load_transactions() function reads and cleans a CSV file of transactions. It removes headers and empty cells, then checks how the data is formatted. If there’s one column, it splits items in that column. If one column has many separators, it treats that as the item list. Otherwise, it assumes the first column is an ID and the rest are items, returning a clean list of transactions.

```
[ ]: # Convert list-of-lists transactions into a one-hot encoded DataFrame (bool␣ ↪dtype)
 def to_onehot(tx):
     items = sorted({i for t in tx for i in t})
```

```
    return pd.DataFrame([{i: (i in set(t)) for i in items} for t in tx],␣
↪dtype=bool)

# Count transactions containing a given itemset (helper for brute-force) def
_support_count(items, tx):
    st = set(items)
    return sum(st.issubset(t) for t in tx)
```

The to_onehot() function converts transactions into a one-hot encoded table where each item becomes a column marked True or False. The _support_count() function counts how many trans actions contain all items from a given itemset.

```
[ ]: # Generate association rules from frequent itemsets with unified semantics␣
↪across algorithms
def _rules_from_fi(fi_df, conf_pct):
    # fi_df must have columns: 'itemset' (iterable/frozenset) and 'support'␣
↪(fraction 0..1)
    if fi_df.empty:
        return pd.
↪DataFrame(columns=["antecedents","consequents","support","confidence"])
    min_conf = conf_pct / 100.0
    sup_map = {}
```



```
    for fs, s in zip(fi_df["itemset"], fi_df["support"]):
        k = fs if isinstance(fs, frozenset) else frozenset(fs)
        sup_map[k] = float(s)

    rows = []
    for L, supL in sup_map.items():
        if len(L) < 2: # need at least 2 items to form A|B
            continue
        for r in range(1, len(L)):
            for A in itertools.combinations(L, r):
                A = frozenset(A)
                B = L - A
                supA = sup_map.get(A)
                if not supA or supA <= 0:
                    continue
                conf = supL / supA
                if conf >= min_conf:
                    rows.append({
                        "antecedents": A,
                        "consequents": B,
                        "support": supL,
                        "confidence": conf
                    })
    return pd.DataFrame(rows,␣
```

```
↪columns=["antecedents","consequents","support","confidence"])
```

The _rules_from_fi() function creates association rules from frequent itemsets. It checks each itemset that meets the minimum confidence level and splits it into possible rule pairs (A → B). For each pair, it calculates support and confidence, keeping only those that meet the confidence threshold. The result is a table of rules showing which items are likely to be bought together.

[ ]:
```
# Brute-force miner (custom): enumerate all item combinations meeting minsup;␣
↪then build rules
def brute_force(tx, sup_pct, conf_pct):
    t0 = time.perf_counter()
    n = len(tx)
    if n == 0:
        return (pd.DataFrame(columns=["itemset","support"]),
                pd.
↪DataFrame(columns=["antecedents","consequents","support","confidence"]), 0.0)

    min_sup = max(1, math.ceil(sup_pct / 100 * n))
    min_conf = conf_pct / 100
    all_items = sorted({i for t in tx for i in t})

    freq = {} # map frozenset -> support count
```



```
    k = 1
    while True:
        found = False
        for comb in itertools.combinations(all_items, k):
            cnt = _support_count(comb, tx)
            if cnt >= min_sup:
                freq[frozenset(comb)] = cnt
                found = True
        if not found:
            break
        k += 1

    fi_rows = [{"itemset": fs, "support": c/n} for fs, c in freq.items()] fi_df =
    pd.DataFrame(fi_rows) if fi_rows else pd.
↪DataFrame(columns=["itemset","support"])

    # Rules with same logic as _rules_from_fi to ensure parity
    rules = []
    sup_map = {fs: c/n for fs, c in freq.items()}
    for L, supL in sup_map.items():
        if len(L) < 2:
            continue
        for r in range(1, len(L)):
```

```
                for A in itertools.combinations(L, r):
                    A = frozenset(A)
                    B = L - A
                    supA = sup_map.get(A, 0.0)
                    if supA <= 0:
                        continue
                    conf = supL / supA
                    if conf >= min_conf:
                        rules.append({
                            "antecedents": A,
                            "consequents": B,
                            "support": supL,
                            "confidence": conf
                        })
    rules_df = pd.DataFrame(rules,␣
 ↪columns=["antecedents","consequents","support","confidence"]) dt =
    time.perf_counter() - t0
    return fi_df, rules_df, dt
```

The brute_force() function finds frequent itemsets and rules by checking every possible item com bination. It counts how often each combination appears and keeps those that meet the minimum support. Then, it creates rules from these itemsets and keeps only the ones meeting the confidence level. Finally, it returns the frequent itemsets, rules, and the time it took to run.

[ ]:
```
# Apriori miner via mlxtend (find frequent itemsets; rules created by shared␣ ↪rule generator)
def apriori(df_onehot, sup_pct, conf_pct):
    t0 = time.perf_counter()
    sup = sup_pct / 100.0
    fi = _apriori(df_onehot, min_support=sup, use_colnames=True)
    if fi.empty:
        return (pd.DataFrame(columns=["itemset","support"]),
                pd.
  ↪DataFrame(columns=["antecedents","consequents","support","confidence"]),
                time.perf_counter() - t0)
    fi = fi.rename(columns={"itemsets":"itemset"})[["itemset","support"]] rules =
    _rules_from_fi(fi, conf_pct)
    dt = time.perf_counter() - t0
    return fi, rules, dt
```

The apriori() function uses the Apriori algorithm to find frequent itemsets from the one-hot encoded data. It filters itemsets based on the minimum support and then generates rules using the shared rule function. Finally, it returns the frequent itemsets, the generated rules, and the total execution time.

[ ]:
```
# FP-Growth miner via mlxtend (find frequent itemsets; rules created by shared␣ ↪rule
  generator)
```

```
def fpgrowth(df_onehot, sup_pct, conf_pct):
    t0 = time.perf_counter()
    sup = sup_pct / 100.0
    fi = _fpgrowth(df_onehot, min_support=sup, use_colnames=True)
    if fi.empty:
        return (pd.DataFrame(columns=["itemset","support"]),
                pd.
 ↪DataFrame(columns=["antecedents","consequents","support","confidence"]),
                time.perf_counter() - t0)
    fi = fi.rename(columns={"itemsets":"itemset"})[["itemset","support"]] rules =
    _rules_from_fi(fi, conf_pct)
    dt = time.perf_counter() - t0
    return fi, rules, dt
```

The fpgrowth() function uses the FP-Growth algorithm to quickly find frequent itemsets from the one-hot encoded data. It keeps only itemsets that meet the minimum support and then generates rules using the shared rule function. Finally, it returns the frequent itemsets, generated rules, and the time taken to run.

```
[ ]: # Convert itemsets to readable strings for printing/saving
def _format_itemset(x):
    if isinstance(x, (set, frozenset, list, tuple)):
        return ", ".join(sorted(map(str, x)))
    try:
        return ", ".join(sorted(map(str, list(x))))
```



```
    except Exception:
        return str(x)
```

The _format_itemset() function converts a collection of items (like a set or list) into a readable string. It sorts and joins all items with commas, such as turning {“Milk”, “Bread”} into “Bread, Milk”. If formatting fails, it simply returns the item as a string.

```
[ ]: # Pretty-print association rules sorted by confidence then support def
print_rules_table(df, title):
    print(f"\n{title}")
    if df.empty:
        print("(no rules)")
        return
    d = df.sort_values(["confidence","support"], ascending=False).copy()
    d["antecedents"] = d["antecedents"].apply(_format_itemset)
    d["consequents"] = d["consequents"].apply(_format_itemset)
    d["support"] = d["support"].map(lambda v: f"{v:.2f}")
    d["confidence"] = d["confidence"].map(lambda v: f"{v:.2f}")
    print(tabulate(d[["antecedents","consequents","support","confidence"]], ␣
 ↪headers=["Antecedent(s)","Consequent(s)","Support","Confidence"],
                   tablefmt="github", showindex=False))
```

The print_rules_table() function neatly displays all association rules in a table format. It sorts the rules by confidence and support, formats itemsets for readability, and rounds the values. If no rules are found, it simply prints “(no rules)”.

```
[ ]: # Pretty-print frequent itemsets sorted by support
     def print_itemsets_table(df, title):
         print(f"\n{title}")
         if df.empty:
             print("(no frequent itemsets)")
             return
         d = df.copy().sort_values(["support"], ascending=[False])
         d["itemset"] = d["itemset"].apply(_format_itemset)
         d["support"] = d["support"].map(lambda v: f"{v:.2f}")
                 print(tabulate(d[["itemset","support"]], headers=["Itemset","Support"],
                                tablefmt="github", showindex=False))
```

The print_itemsets_table() function displays all frequent itemsets in a clear table. It sorts them by support in descending order, formats the item names for readability, and rounds the support values. If no itemsets are found, it prints “(no frequent itemsets)”.

```
[ ]: # Save frequent itemsets to CSV (formats itemset lists into strings) def
     save_csv_itemsets(df, path):
         os.makedirs(os.path.dirname(path), exist_ok=True)
         out = df.copy()
         out["itemset"] = out["itemset"].apply(_format_itemset)
```



```
         out = out[["itemset","support"]] if "support" in out.columns else
     ↪out[["itemset"]]
         out.to_csv(path, index=False)

     # Save association rules to CSV (antecedents/consequents as strings) def
     save_csv_rules(df, path):
         os.makedirs(os.path.dirname(path), exist_ok=True)
         out = df.copy()
         out["antecedents"] = out["antecedents"].apply(_format_itemset)
         out["consequents"] = out["consequents"].apply(_format_itemset)
         out = out[["antecedents","consequents","support","confidence"]]
         out.to_csv(path, index=False)
```

The save_csv_itemsets() function saves frequent itemsets to a CSV file, converting each itemset into a readable string before writing. The save_csv_rules() function does the same for association rules, formatting the antecedents and consequents, then saving them along with their support and confidence values.

```
[ ]: # Normalize user-provided percentage/fraction to percentage
     def normalize_pct(value, default_if_none):
```

```
        if value is None:
            return float(default_if_none)
        v = float(value)
        if 0.0 <= v <= 1.0:
            return v * 100.0
        return v

    def _pct_arg(s: str) -> float:
        """Accept 0..1 (fraction) or 1..100 (percent), but disallow 0.""" try:
            v = float(s)
        except ValueError:
            raise argparse.ArgumentTypeError("Must be a number.")
        if (0 < v <= 1) or (1 < v <= 100) or v == 1.0:
            return v
        raise argparse.ArgumentTypeError("Use (0,1] for fraction or (0,100] for␣ ↪percent; 0
      is not allowed.")

    # Prompt for minsup/minconf if omitted
    def prompt_pct(label: str, default: float) -> float:
        while True:
            s = input(f"{label} (1-100 or 0..1) [{default}]: ").strip()
            if s == "":
                return float(default)
            try:
                v = float(s)
                # disallow 0; allow (0,1] or (0,100]
```



```
                if (0 < v <= 100) or (0 < v <= 1):
                    return v
            except Exception:
                pass
                        print("Please enter a number in (0,100] or (0,1] (e.g., 0.4 or 40).")
```

normalize_pct() converts user input into a percentage. If the value is between 0 and 1, it multiplies by 100; otherwise, it returns the number as-is.

_pct_arg() checks that the input is a valid number greater than 0 and within 1–100 (percent) or 0–1 (fraction). It prevents users from entering 0 or invalid values.

prompt_pct() asks the user to enter a support or confidence value. It repeats the prompt until the user gives a valid number greater than 0 within the allowed range.

```
[ ]: # Interactive dataset chooser for when --dataset is omitted
    def choose_dataset():
        keys = list(DATASETS.keys())
        while True:
            print("Choose a dataset:")
```

```
        for i,k in enumerate(keys,1):
            print(f" {i}. {k.title()} ({DATASETS[k]})")
        sel = input(f"Enter number (1-{len(keys)}): ").strip()
        if sel.isdigit() and 1 <= int(sel) <= len(keys):
            key = keys[int(sel)-1]; path = DATASETS[key]
            if os.path.exists(path): return key, path
            print(f"File not found: {path}")
        else:
            print(f"Invalid choice. Enter 1–{len(keys)}.")

# Merge frequent itemsets from multiple methods, keeping max support per unique itemset
def consolidate_itemsets(*dfs):
    parts = [df[["itemset","support"]].copy() for df in dfs if df is not None and not df.empty]
    if not parts:
        return pd.DataFrame(columns=["itemset","support","len"])
    cat = pd.concat(parts, ignore_index=True)
    cat["key"] = cat["itemset"].apply(lambda x: frozenset(x) if not isinstance(x, frozenset) else x)
    agg = (cat.groupby("key", as_index=False).agg({"support":"max"}))
    agg["itemset"] = agg["key"].apply(lambda k: k)
    agg["len"] = agg["itemset"].apply(len)
    return agg[["itemset","support","len"]]
```

The choose_dataset() function asks the user to pick which dataset to use when none is given through the command line. It displays all available dataset options, waits for the user to enter a number, and returns the selected dataset's name and file path once confirmed.

The consolidate_itemsets() function combines frequent itemsets from all algorithms. It merges duplicates, keeps the highest support value for each unique itemset, and adds the itemset length for easy comparison.

[ ]:
```
# Orchestrate the full pipeline: load, mine (3 methods), print, and save outputs
def run_all(ds_key, path, sup_in, conf_in):
    sup_pct = normalize_pct(sup_in, 20)
    conf_pct = normalize_pct(conf_in, 50)

    tx = load_transactions(path)
    n, uniq = len(tx), Counter(i for t in tx for i in t)
    need = math.ceil(sup_pct/100 * n) if n else 0

    print(f"\n===== {ds_key.title()} =====")
    print(f"Loaded {n} transactions, {len(uniq)} unique items. "
          f"Min Support {sup_pct:.0f}% (>= {need}/{n}), Min Confidence {conf_pct:.0f}%")
    if uniq:
```

```
        freqs = pd.DataFrame(sorted(uniq.items(), key=lambda x: (-x[1], x[0])),␣
↪columns=["Item","Count"])
        print(tabulate(freqs, headers="keys", tablefmt="github",␣
↪showindex=False))

    df_onehot = to_onehot(tx)

    # Prepare output directories per algorithm
    root_out = os.path.join("outputs", ds_key)
    out_bf = os.path.join(root_out, "Brute-Force")
    out_ap = os.path.join(root_out, "Apriori")
    out_fp = os.path.join(root_out, "FP-Growth")
    os.makedirs(root_out, exist_ok=True)

    timings = []

    # Brute-Force mining + CSV export
    bf_itemsets, bf_rules, t = brute_force(tx, sup_pct, conf_pct)
    timings.append(("Brute-Force", int(len(bf_itemsets)), int(len(bf_rules)),␣ ↪round(t,4)))
    save_csv_itemsets(bf_itemsets, os.path.join(out_bf, "frequent_itemsets. ↪csv"))
                    save_csv_rules(bf_rules, os.path.join(out_bf, "association_rules.csv"))

    # Apriori mining + CSV export
    ap_itemsets, ap_rules, t = apriori(df_onehot, sup_pct, conf_pct)
    timings.append(("Apriori", int(len(ap_itemsets)), int(len(ap_rules)),␣ ↪round(t,4)))
```



```
    save_csv_itemsets(ap_itemsets, os.path.join(out_ap, "frequent_itemsets. ↪csv"))
                    save_csv_rules(ap_rules, os.path.join(out_ap, "association_rules.csv"))

    # FP-Growth mining + CSV export
    fp_itemsets, fp_rules, t = fpgrowth(df_onehot, sup_pct, conf_pct)
timings.append(("FP-Growth", int(len(fp_itemsets)), int(len(fp_rules)),␣ ↪round(t,4)))
    save_csv_itemsets(fp_itemsets, os.path.join(out_fp, "frequent_itemsets. ↪csv"))
                    save_csv_rules(fp_rules, os.path.join(out_fp, "association_rules.csv"))

    # Print consolidated itemsets and per-method rule tables
    consolidated = consolidate_itemsets(bf_itemsets, ap_itemsets, fp_itemsets)
    print_itemsets_table(consolidated, "Frequent Itemsets (Consolidated - all)")
    print_rules_table(bf_rules, "Brute-Force: Association Rules (all)")
    print_rules_table(ap_rules, "Apriori: Association Rules (all)")
    print_rules_table(fp_rules, "FP-Growth: Association Rules (all)")

    # Print and save timing summary CSV
    tdf = pd.DataFrame(timings, columns=["Algorithm","Frequent␣
↪Itemsets","Rules","Time (s)"])
```

```
    print("\nTiming Summary:")
    print(tabulate(tdf, headers="keys", tablefmt="github", showindex=False))
    tdf.to_csv(os.path.join(root_out, "timings.csv"), index=False)
```

The run_all() function executes the whole workflow for one dataset. It normalizes thresholds, loads and summarizes the data, and builds a one-hot table. Then it runs Brute-Force, Apriori, and FP-Growth, saving itemsets and rules for each. It prints consolidated itemsets and per-method rule tables. Finally, it outputs a timing summary and saves it to ./outputs//timings.csv.

[ ]:
```
def main(argv=None):
    p = argparse.ArgumentParser(
        description="Run Brute-Force, Apriori, FP-Growth on one dataset; prints␣
↪consolidated itemsets and per-alg rules."
    )
    p.add_argument("--dataset", choices=list(DATASETS.keys()))
    p.add_argument("--minsup", type=float)
    p.add_argument("--minconf", type=float)

    if argv is None:
        argv = [] if 'ipykernel' in sys.modules else sys.argv[1:]

    a, _ = p.parse_known_args(argv) # <-- ignore Jupyter's extra args

    if a.dataset:
        key, path = a.dataset, DATASETS[a.dataset]
```



```
    else:
        key, path = choose_dataset()

    sup_in = a.minsup if a.minsup is not None else prompt_pct("Minimum␣
↪support", 20)
    conf_in = a.minconf if a.minconf is not None else prompt_pct("Minimum␣
↪confidence", 50)
    run_all(key, path, sup_in, conf_in)

if __name__ == "__main__":
    main()
```

[ ]:

It lets the program accept command-line inputs for dataset, minimum support, and confidence. If no arguments are given, it asks the user interactively. It chooses the dataset, gets support and confidence values, and then runs all algorithms. The if **name** == “**main**”: main() line runs the function when the script is executed directly.