Advanced Human–Machine Interaction
Interaction Data Analysis
TDM03: PrefixSpan – Correction

INSA Rouen Normandie – Normandie University

The complete executable correction is provided in `prefixspan_solution.py`. This document explains the important modelling and algorithmic choices.

1. Formalisation

- **Phenomenon:** recurrent navigation and help-seeking behaviour during an exercise session.
- **Observable traces:** timestamped interface actions. One trace corresponds to one `session_id`.
- **Representation:** a sequence over the action alphabet Σ. Each action is a singleton itemset.
- **Target output:** ordered subsequences whose trace support exceeds a threshold.
- **Problem class:** frequent sequential pattern mining.
- **Evaluation unit:** initially the session; later comparisons use the user.
- **Generalisation claim:** a user-disjoint split is required to evaluate new users.

The output is descriptive. A frequent pattern does not establish a causal relation between its actions.

2. Reconstructing sessions

```
def build_session_sequences(rows):
    events = defaultdict(list)
    session_user = {}
    for row in rows:
        sid = row["session_id"]
        session_user[sid] = row["user_id"]
        events[sid].append((int(row["timestamp_s"]),
                            int(row["event_id"]),
                            row["action"]))
    sequences = {
        sid: [action for _, _, action in sorted(values)]
        for sid, values in events.items()
    }
    return sequences, session_user
```

The second sorting key gives a deterministic order when two observations share a timestamp. The data contain 789 events, 96 sessions and 36 users.

Grouping only by user would concatenate independent sessions and could create false cross-session patterns.

3. Subsequence and support

```
def contains_subsequence(sequence, pattern):
    if not pattern:
        return True
    pattern_index = 0
    for item in sequence:
        if item == pattern[pattern_index]:
            pattern_index += 1
            if pattern_index == len(pattern):
                return True
    return False

def trace_support(sequences, pattern):
    sequences = list(sequences)
    count = sum(contains_subsequence(s, pattern) for s in sequences)
    return count, count / len(sequences) if sequences else 0.0
```

Pattern	Session count	Session support
$\langle view_hint, submit \rangle$	59/96	0.615
$\langle submit, read_feedback, retry \rangle$	49/96	0.510
$\langle view_example, bookmark \rangle$	32/96	0.333

The algorithm stops as soon as all elements of the pattern have been matched. Repeated occurrences in the same trace therefore do not increase trace support.

4. Projected databases

```
def project_sequence(sequence, prefix):
    position = 0
    for expected in prefix:
        for index in range(position, len(sequence)):
            if sequence[index] == expected:
                position = index + 1
                break
        else:
            return None
    return sequence[position:]

def project_database(database, prefix):
    projected = []
    for sequence in database:
        suffix = project_sequence(sequence, prefix)
        if suffix is not None:
            projected.append(suffix)
    return projected
```

The earliest embedding leaves the longest possible suffix. It therefore preserves every possible extension of the prefix in the single-item setting.

```
def frequent_extensions(projected_database, minimum_count):
    counts = Counter()
    for suffix in projected_database:
        counts.update(set(suffix))
    return sorted(
        [(item, count) for item, count in counts.items()
         if count >= minimum_count],
        key=lambda pair: (-pair[1], pair[0]))
```

The call to `set(suffix)` is essential: support counts traces, not occurrences.

For $\langle open_question, view_hint \rangle$, the projected database contains 68 suffixes. With a minimum count of 10, frequent extensions are:

Extension	Count	Extension	Count
next_question	59	submit	59
retry	35	read_feedback	21
close_popup	17	view_example	17
open_menu	12		

5. PrefixSpan recursion

```
def prefixspan(database, minimum_support, maximum_length=4):
    minimum_count = math.ceil(minimum_support * len(database))
    patterns = []

    def grow(prefix, projected_database):
        for item, count in frequent_extensions(projected_database,
                                               minimum_count):
            new_prefix = prefix + [item]
            patterns.append((tuple(new_prefix), count,
                             count / len(database)))
            if len(new_prefix) < maximum_length:
                new_projected = project_database(projected_database,
                                                 [item])
                grow(new_prefix, new_projected)

    grow([], database)
    patterns.sort(key=lambda r: (-r[1], -len(r[0]), r[0]))
    return patterns
```

At each level, `projected_database` already contains the suffix following the current prefix. It must therefore be projected only with the new extension, not with the complete prefix again.

Minimum support	All patterns	Length at least 2
0.10	703	690
0.20	302	290
0.35	132	122

Examples with `min_support=0.20` include:

Pattern	Support
$\langle open_course, open_question, submit \rangle$	1.000
$\langle open_question, view_hint \rangle$	0.708
$\langle view_hint, submit \rangle$	0.615
$\langle open_question, view_example \rangle$	0.531
$\langle submit, read_feedback, retry \rangle$	0.510

The first pattern is frequent but almost entirely determined by the platform workflow. Lowering the threshold creates many variants obtained by adding ubiquitous actions. Closed or maximal patterns could reduce this redundancy.

6. User support

```
def user_support(sequences_by_session, session_user, pattern):
    sessions_by_user = defaultdict(list)
    for sid, sequence in sequences_by_session.items():
        sessions_by_user[session_user[sid]].append(sequence)
    count = sum(
        any(contains_subsequence(s, pattern) for s in sessions)
        for sessions in sessions_by_user.values())
    return count, count / len(sessions_by_user)
```

Pattern	Session support	User support
$\langle view_hint, submit \rangle$	$59/96 = 0.615$	$20/36 = 0.556$
$\langle submit, read_feedback, retry \rangle$	$49/96 = 0.510$	$25/36 = 0.694$
$\langle view_example, bookmark \rangle$	$32/96 = 0.333$	$4/36 = 0.111$

The last pattern appears in one third of all sessions but belongs to only four users. Describing it as a widespread population behaviour would therefore be misleading.

7. User-disjoint evaluation

With seed 42, the 70/30 split contains 25 training users and 11 test users, corresponding to 62 and 34 sessions. The unequal session counts result from the deliberately active users.

The correction mines patterns on the 62 training sessions and recomputes their support on the 34 test sessions. For example:

Pattern	Training support	Test support
$\langle open_question, submit \rangle$	1.000	1.000
$\langle view_hint, submit \rangle$	0.532	0.765
$\langle open_question, close_popup, submit \rangle$	0.323	0.206

A random row-level split would place observations from the same session and user in both sets. The test set would then fail to represent genuinely unseen interaction behaviour.

8. Maximum-gap constraint

The full implementation in `prefixspan_solution.py` keeps every currently possible match position. This avoids the failure of a purely greedy approach when a later occurrence enables the pattern.

For $\langle open_question, view_hint, submit \rangle$:

Constraint	Supporting sessions
Unrestricted	59
Maximum gap 0	8
Maximum gap 2	33

The gap constraint changes the semantic claim. The unrestricted pattern means that a hint precedes a submission somewhere in the session. With gap zero, the three actions must be consecutive.

Main conclusions

- Session construction determines the boundaries within which a pattern may exist.
- Trace support prevents one session from contributing repeatedly, but active users may still dominate session-level results.
- PrefixSpan preserves order through projected databases.
- Support thresholds and temporal constraints define what the algorithm regards as a pattern.
- A user-disjoint split is required before claiming generalisation to new users.