I have 250k examples of sessionized clickstream data of a banking and insurance company with page sequence list and target variable satisfaction (0-dissatisfied, 1-satisfied) I'm doing a behavior analysis. I used networkx to create graphs for each session and extracted cycles from it using the simple cycle method for both dissatisfied and satisfied group and identified cycles appearing more in the dissatisfied group wrt satisfied group based on difference. Similarly I used simple path method to extract simple paths upto length 5 and found paths appearing more in the dissatisfied group than satisfied group based on difference. Now I end up with a lot of paths and cycles. How will I identity relevant ones causing user frustration or pain point.

ChatGPT said:

Thought for a couple of seconds

When you end up with hundreds or thousands of candidate cycles and paths, the key is to move from "raw patterns" to "actionable pain points" by (1) scoring and ranking patterns, (2) consolidating/reducing redundancy, and (3) validating them against business metrics or user behavior. Here's a systematic approach:

---

1. Score & Rank Patterns by "Interestingness"

Assign each pattern (cycle or path) a quantitative score that captures both its prevalence and its association with dissatisfaction:

1. **Support**:
   - The fraction of all sessions (or all dissatisfied sessions) in which the pattern appears.

2. **Growth Rate (or "Contrast")**:
   GrowthRate=SupportdissatisfiedSupportsatisfied \text{GrowthRate} = \frac{\text{Support}_\text{dissatisfied}}{\text{Support}_\text{satisfied}}GrowthRate=SupportsatisfiedSupportdissatisfied
   Patterns with GrowthRate ≫ 1 occur disproportionately in frustrated users.

3. **Leverage** (difference in support):
   Leverage=Supportdissatisfied−Supportsatisfied \text{Leverage} = \text{Support}_\text{dissatisfied} - \text{Support}_\text{satisfied}Leverage=Supportdissatisfied−Supportsatisfied
4. **Statistical Significance** (e.g. χ^2 test or Fisher's Exact Test) to filter out patterns whose difference could be due to chance.

You can combine these into a single "interest score," for instance:

Interest=Leverage×log(GrowthRate+1)\text{Interest} = \text{Leverage} \times \log(\text{GrowthRate} + 1)Interest=Leverage×log(GrowthRate+1) and then sort descending.

---

2. Reduce & Consolidate Redundant Patterns

Raw mining often yields many near‑duplicates. Techniques to collapse or summarize:

- **Closed or Maximal Patterns**: keep only patterns that aren't strictly contained in any other pattern with equal or greater support.

- **Clustering Similar Sequences**: compute pairwise edit‑distance or Jaccard‑similarity between paths/cycles, then cluster (e.g. hierarchical or k-means on sequence embeddings), and pick the highest‑scoring representative from each cluster.

- **Length Constraints and Pruning**: if very long sequences are hard to act on, set a maximum length (e.g. ≤ 5) and require a minimum leverage threshold before keeping them.

---

3. Validate Against Business Metrics

Not every "statistically interesting" pattern is a real pain point. To prioritize:

1. **Link to Downstream Metrics**
   For sessions containing pattern *P*, compute:

   - **Drop‑off rate** (rate at which users abandon an application or sign‑up),

   - **Time on site** after seeing P,

   - **Conversion rate** (e.g., quote requests, purchases).

2. Patterns where "presence of *P* ⇒ conversion drops dramatically" are high‑priority.

3. **A/B or Time-Window Analysis**
   If you can instrument your site, you might A/B test alternative flows to see if removing or simplifying the step(s) in *P* improves satisfaction.

4. **Human Review + UX Expertise**
   Bring in product or UX analysts to look at the top 10–20 patterns and quickly judge which ones "make sense" (e.g. an error‑page loop, back-and-forth between

loan-requirements screens).

---

4. Build a Supervised “Pattern Importance” Model

Turn your mined patterns into features in a simple predictive model:

1. **Feature Matrix**: each session → binary vector indicating presence/absence of top N patterns.
2. **Train** a regularized logistic regression (L1 or L2) or tree-based model to predict dissatisfaction.
3. **Interpret** feature coefficients or importance scores to see which patterns truly drive the model.

This automatically balances prevalence with predictive power.

---

5. Visualize & Communicate

- **Sankey diagrams** or **alluvial plots** to show how users flow in and out of a cycle/path, and where they drop off.
- **Heatmaps** of time-spent on pages in a cycle vs. satisfaction.
- Simple **bar charts** of your top 10 patterns by “Interest Score” and conversion rates.

Putting It All Together

1. **Mine** cycles and paths as you have.
2. **Score** every pattern by support, leverage, and growth rate.
3. **Filter** by a minimum leverage and p-value cutoff (e.g. $p < 0.01$).

4. **Collapse** redundant patterns via closed set mining or clustering.

5. **Validate** top patterns against conversion/drop-off metrics.

6. **Model** importance via simple supervised learning.

7. **Visualize** the final 5–10 to share with stakeholders.

By moving from raw patterns to scored, pruned, statistically validated, and business-linked insights—then summarizing them visually—you'll be able to pinpoint the handful of cycles or paths that are true pain points and deserving of remediation.

You said:

I took top 100K cycles and paths appearing from both groups and made a dataframe with more cycle appearing in dissatisfied (based on its count) wrt satisfied along with session ids of each cycle in both groups (same for simple paths) From this, how can I identify pain points or user frustration ?

ChatGPT said:

Thought for 6 seconds

Here's how you can turn that 100 K-row DataFrame into a shortlist of true pain-point patterns:

---

1. Compute "Interest" Metrics

Add columns that capture both prevalence and contrast:

Support (relative frequency)

python
CopyEdit
```
df['support_d'] = df['count_dissatisfied'] /
total_dissatisfied_sessions
df['support_s'] = df['count_satisfied']     / total_satisfied_sessions
```

1.

Leverage

python
CopyEdit
```
df['leverage'] = df['support_d'] - df['support_s']
```

2.

Growth Rate

python
CopyEdit

```python
# add a small ε to avoid division by zero
eps = 1e-6
df['growth'] = (df['support_d'] + eps) / (df['support_s'] + eps)
```

3.

Combined Interest Score
A simple combination is
interest=leverage×log(1+growth)\text{interest} = \text{leverage} \times \log(1 + \text{growth})interest=leverage×log(1+growth)
python
CopyEdit

```python
import numpy as np
df['interest'] = df['leverage'] * np.log1p(df['growth'])
```

4.

---

2. Filter & Rank

- **Minimum Prevalence Filter**: drop any pattern with `support_d` below, say, 0.5% (or whatever your cutoff is).
- **Minimum Leverage**: keep only patterns where `leverage > 0.01` (i.e. at least a 1-point difference in %, tune to your business).
- **Sort** by `interest` descending.

That gives you perhaps a few hundred top candidates instead of 100 K.

---

3. Prune Redundancy

- **Closed/Maximal Patterns**: if a cycle/path A is strictly contained in B and both have very similar support, you can drop A or B depending on whether you want the longer or the shorter pattern.

- **Clustering by Similarity**: represent each pattern as a token sequence and cluster (e.g., via edit distance or embedding + k-means). Then pick the highest-interest example in each cluster.

---

4. Link Back to Session-Level Outcomes

Use your session-ID lists to compute, for each remaining pattern P:

- **Drop-off Rate**: among sessions containing P, what % abandoned vs. completed?

- **Conversion Rate**: among those sessions, what % converted (e.g. submitted a form)?

- **Time-on-Site**: how long did those sessions last on average?

Rank patterns by the size of the drop-off or conversion delta versus baseline. Real pain points will show both a strong support-contrast signal and a large adverse effect on these downstream metrics.

---

5. Build a Lightweight Predictive Model

Turn your final ~50 patterns into binary features (one column per pattern: did this session contain P?). Fit a regularized logistic regression to predict dissatisfaction:

- Coefficients $\gg 0$ point to the patterns that are most predictive of frustration.

- Cross-validate to ensure stability.

---

6. Surface & Validate

1. **Inspect** the top 10 patterns manually (with a UX designer or product manager).
2. **Visualize** with Sankey/alluvial diagrams or simple bar charts showing “pattern → drop-off %.”
3. **A/B Test**: tweak the flow around the top 1–2 pain points and measure if satisfaction goes up.

In Practice

You don’t need to try all these at once—here’s a minimal code sketch (pseudocode) you could drop into your notebook:

python
CopyEdit

```
import numpy as np

# assume df has: count_dissatisfied, count_satisfied,
session_ids_dissatisfied, session_ids_satisfied
total_d = TOTAL_DISSATISFIED_SESSIONS
total_s = TOTAL_SATISFIED_SESSIONS

df['support_d'] = df['count_dissatisfied'] / total_d
df['support_s'] = df['count_satisfied']     / total_s
df['leverage']  = df['support_d'] - df['support_s']
df['growth']    = (df['support_d'] + 1e-6) / (df['support_s'] + 1e-6)
df['interest']  = df['leverage'] * np.log1p(df['growth'])

# filter+sort
df_top = (
    df[df['support_d'] >= 0.005]       # at least 0.5% prevalence
      .query('leverage > 0.01')       # at least 1% more often in
unhappy users
      .sort_values('interest', ascending=False)
      .head(200)                      # keep top 200 for deeper
analysis
)
```

```
# for each pattern in df_top, compute drop-off / conversion
def session_metric(session_ids, metric):
    # your function to pull drop-off or conversion rate for a list of
session IDs
    return compute_metric(session_ids)

df_top['drop_off_rate'] =
df_top['session_ids_dissatisfied'].apply(lambda ids:
session_metric(ids, 'dropoff'))
df_top['conversion_rate'] =
df_top['session_ids_dissatisfied'].apply(lambda ids:
session_metric(ids, 'conversion'))

# final ranking by drop-off delta
baseline = overall_drop_off_rate
df_top['dropoff_delta'] = df_top['drop_off_rate'] - baseline
df_final = df_top.sort_values('dropoff_delta',
ascending=False).head(10)
```

That `df_final` is your shortlist of real pain points—patterns that (a) occur disproportionately in frustrated users **and** (b) correlate with massive drop-off or low conversion. From there, you can visualize and plan fixes.

You said:

Can you explain the computation of interest metric step by step ? Any other statistical significance test is required ?

ChatGPT said:

Here's a detailed walkthrough of how you compute the "interest" score for each pattern, plus a note on statistical testing:

---

1. Compute Supports

For each pattern PPP:

1. Count how many **dissatisfied** sessions contain PPP:
 countd=#{sessions in group D with P} \text{count}_d = \#\{\text{sessions in group D with }P\}countd=#{sessions in group D with P}

2. Count how many **satisfied** sessions contain PPP:
   counts=#{sessions in group S with P} \text{count}_s = \#\{\text{sessions in group S with }P\}counts=#{sessions in group S with P}
3. Divide by the total sessions in each group to get **support** (i.e. relative frequency):
   supportd=countdNd,supports=countsNs,\begin{aligned} \text{support}_d &= \frac{\text{count}_d}{N_d},\\ \text{support}_s &= \frac{\text{count}_s}{N_s}, \end{aligned}supportdsupports=Ndcountd,=Nscounts,
   where NdN_dNd and NsN_sNs are the total number of dissatisfied and satisfied sessions, respectively.

---

2. Compute Leverage

Leverage measures the absolute difference in support between the two groups:

leverage=supportd − supports. \text{leverage} = \text{support}_d \;-\; \text{support}_s.leverage=supportd−supports.

This tells you "what percentage more sessions in D contain this pattern than in S."

---

3. Compute Growth Rate

Growth rate (sometimes called "contrast") is the ratio of supports—how many times more common the pattern is in D vs. S:

growth = supportd+εsupports+ε, \text{growth} \;=\; \frac{\text{support}_d + \varepsilon}{\text{support}_s + \varepsilon},growth=supports+εsupportd+ε,

where ε\varepsilonε is a small constant (e.g. 10−610^{-6}10−6) to avoid division by zero when supports=0\text{support}_s=0supports=0.

---

4. Compute Interest Score

A simple way to combine leverage and growth into one metric is:

interest=(supportd−supports)⏟leverage × log(1+growth). \text{interest} = \underbrace{\bigl(\text{support}_d - \text{support}_s\bigr)}_{\text{leverage}} \;\times\; \log\bigl(1 + \text{growth}\bigr).interest=leverage(supportd−supports)×log(1+growth).

- **Leverage** gives you absolute difference—so very rare patterns (low support) don't score highly even if they have huge ratios.
- **log(1+growth)\log(1+\text{growth})log(1+growth)** dampens extremely high ratios while still rewarding patterns much more common in D.

Python snippet

python
CopyEdit

```
import numpy as np

eps = 1e-6

df['support_d'] = df['count_d'] / total_dissatisfied
df['support_s'] = df['count_s'] / total_satisfied

df['leverage'] = df['support_d'] - df['support_s']
df['growth']   = (df['support_d'] + eps) / (df['support_s'] + eps)
df['interest'] = df['leverage'] * np.log1p(df['growth'])
```

---

5. Statistical Significance Testing

Even a big leverage could occur by chance if counts are low. To filter out noisy patterns, you should run a significance test on the 2×2 contingency table:

	Pattern PPP present	**Pattern PPP absent**
Dissatisfied	a=countd\,a = \text{count}_da=countd	Nd−aN_d - aNd−a
Satisfied	b=counts\,b = \text{count}_sb=counts	Ns−bN_s - bNs−b

χ² (Chi-Square) Test

- Good for moderate-to-large counts.
- Null hypothesis: pattern occurrence is independent of group.
- Compute ppp-value; drop patterns with ppp above your threshold (e.g. 0.01).

Fisher's Exact Test

- Preferable when any of a,b,Nd−a,Ns−ba,b,N_d-a,N_s-ba,b,Nd−a,Ns−b is small (say <5).
- Computes an exact ppp-value without relying on large-sample approximations.

Multiple Testing Correction

Since you're testing thousands of patterns, apply a correction to control the false discovery rate:

- **Benjamini–Hochberg** (FDR control) is often a good choice.
- Or a stricter **Bonferroni** correction if you need very high confidence.

---

6. Putting It All Together

1. **Compute** support, leverage, growth, interest for each pattern.
2. **Test** each pattern's 2×2 table with χ^2 or Fisher's exact test → get a ppp-value.
3. **Adjust** ppp-values for multiple comparisons (e.g. Benjamini–Hochberg).
4. **Filter** out any pattern with:
    - support_d below your minimum threshold (e.g. 0.5%),
    - leverage below your business threshold (e.g. 1%),
    - adjusted ppp-value above your chosen alpha (e.g. 0.05 or 0.01).
5. **Rank** the survivors by descending interest score.

The result will be a compact, statistically robust list of patterns that are both common enough and significantly more prevalent in frustrated users—your highest-priority pain points.

You said:

As it is a banking and insurance company, do we need to group cycles and paths into different logical categories after finding the significant ones?

ChatGPT said:

Yes—once you've winnowed down to the truly "interesting" cycles and paths, it's extremely valuable to organize them by **business‑relevant buckets**, rather than treating every sequence as an island. Here's why and how:

Why Group into Logical Categories?

1. **Actionability**
   – A cycle involving "Login → Forgotten Password → Login Error → Forgotten Password" is very different (and higher priority) than one involving "Product Comparison → Back → Product Comparison" in terms of root cause and fix.

2. **Stakeholder Alignment**
   – Product managers for loans care about loan-application flows; customer-service teams care about password resets or claims submissions. Grouping speaks each team's language.

3. **Better Prioritization**
   – You can compare, say, "50% of all painful paths are in the 'Account Access' category vs. 10% in the 'Claims' category," and allocate your UX/design relicensing budget accordingly.

How to Build Your Category Taxonomy

1. **Leverage Page Metadata**
   – If each page has a "page_type" or belongs to a section (e.g. Account, Loan, Claims, Payments, Help/Support), you can automatically map each node in your path/cycle to a bucket.

2. **Define High-Level Journeys**
   – Common top-level journeys in banking/insurance might include:
   - **Account Access & Security** (login, 2FA, password reset)
   - **Balance / Statements** (viewing statements, downloading PDFs)
   - **Fund Transfers & Payments**
   - **Loan or Credit Applications**
   - **Claims Submission & Tracking**
   - **Support & Help** (chat, FAQs, call-me forms)
3. **Rule-Based Mapping**
   – Create a simple lookup: page URL or page ID → category. Then for any extracted path/cycle, map each step and assign the **dominant category** (the one that appears most, or the first step's category).
4. **Multi-Labeling for Hybrid Flows**
   – Some paths cross boundaries (e.g. "Account → Payments → Support"). Tag those as "Mixed" or "Cross-Journey," then decide if they deserve their own bucket.
5. **Clustering on Semantics**
   – If metadata is missing, you can cluster page URLs or page titles via text embeddings (TF–IDF or a small transformer) and then manually label each cluster with a category.

Workflow Example

1. **Create a mapping table**

page_id or URL pattern	category
`/login*`	Account Access & Security
`/forgot-password*`	Account Access & Security
`/transfer*`	Payments & Transfers
`/loan/apply*`	Loan Application

`/claims/new*`	Claims Submission
`/support/chat`	Support & Help

2.

Map your patterns

python
CopyEdit
```
df['categories'] = df['pattern_steps'].apply(
    lambda seq: [lookup_category(step) for step in seq]
)
# pick the majority or first category
df['primary_category'] = df['categories'].apply(lambda cats:
most_common(cats))
```

3.

Summarize by category

python
CopyEdit
```
report = (
  df.groupby('primary_category')
    .agg({
      'interest': 'mean',
      'leverage': 'mean',
      'dropoff_delta': 'sum',
      'pattern': 'count'
    })
    .sort_values('dropoff_delta', ascending=False)
)
```

4. This tells you which **areas** (e.g. “Account Access” vs. “Claims”) have the biggest aggregated pain.

Benefits of Categorization

- **Targeted Fixes**: UX or engineering can tackle "Account Access" separately from "Payments."
- **Progress Tracking**: You can track reduction in "Account Access" pain‑score over time, independent of other areas.
- **Cross‑Team Reporting**: Business stakeholders get crisp dashboards ("Our top 3 pain points in the Claims funnel are…").

In short: yes—after you've identified statistically significant, high-interest cycles and paths, map them into a small set (5–8) of banking/insurance journey categories. That will transform your 10–20 pain-point sequences into clear, domain-aligned insights and make remediation far more tractable.

You said:

Do we need to calculate interest score first or do statistical significance test?

ChatGPT said:

Thought for a couple of seconds

It's easiest and most transparent to think of it as a two-stage pipeline:

1. **Metric Computation Stage**
 – **Compute your raw counts** (a,b,Nd,Nsa, b, N_d, N_sa,b,Nd,Ns) and derive
 • support_d, support_s
 • leverage = support_d – support_s
 • growth = (support_d + ε)/(support_s + ε)
 • interest = leverage × log(1 + growth)

 This is just arithmetic on your DataFrame and is extremely fast, even for 100 K rows.

2. **Statistical Filtering Stage**
 – **Apply your significance test** (χ^2 or Fisher's exact) to each pattern's 2×22\times22×2 contingency table to get a p-value.
 – **Adjust for multiple comparisons** (e.g. Benjamini–Hochberg).
 – **Filter** out any pattern whose adjusted p-value exceeds your α threshold (e.g. 0.01), or whose raw support is below your minimum.

Once you've thrown away the noise, you'll have a much smaller set of "significant, high-interest" patterns to sort by their interest score and then bucket by business category.

Why in that order?

- **Speed & Efficiency**: computing interest is trivial and lets you immediately see which patterns "look" promising.
- **Selective Testing**: if you'd rather not run tens of thousands of exact tests, you can even do a quick pre-filter (e.g. keep only the top 10 % by interest or those with leverage > threshold) and then run significance tests just on that subset.
- **Clean Pipeline**: by separating metric calculation from hypothesis testing, it's easier to log, visualize, and validate each step independently.

Minimal Workflow Sketch

python

CopyEdit

```python
# 1) Compute metrics
eps = 1e-6
df['support_d'] = df['count_d'] / total_d
df['support_s'] = df['count_s'] / total_s
df['leverage'] = df['support_d'] - df['support_s']
df['growth']   = (df['support_d'] + eps) / (df['support_s'] + eps)
df['interest'] = df['leverage'] * np.log1p(df['growth'])

# (Optional) quick pre-filter by interest or support
df_prefilt = df.query('support_d >= 0.005 and leverage > 0.01')

# 2) Statistical tests on df_prefilt (or on full df if you prefer)
from scipy.stats import chi2_contingency, fisher_exact
def p_value(row):
    table = [[row['count_d'], total_d - row['count_d']],
             [row['count_s'], total_s - row['count_s']]]
    # choose test based on counts
    if min(table[0] + table[1]) < 5:
        return fisher_exact(table)[1]
```

```
    else:
        return chi2_contingency(table, correction=False)[1]

df_prefilt['p_raw'] = df_prefilt.apply(p_value, axis=1)
# adjust p-values (e.g. Benjamini-Hochberg), then
df_sig = df_prefilt[df_prefilt['p_adj'] <= 0.01]

# 3) Final ranking
df_final = df_sig.sort_values('interest', ascending=False)
```

That way you've first built your interest score for every candidate, then you've thrown out any that aren't statistically robust, and you end up with a ranked, actionable list of true pain-point patterns.

You said:

At the metric computation stage , we are not removing any patterns right ?

ChatGPT said:

That's right—during the **Metric Computation Stage** you simply calculate support, leverage, growth and interest **for every** pattern in your DataFrame. No rows are dropped here. It's purely an arithmetic pass:

python
CopyEdit

```
# for all patterns
df['support_d'] = df['count_d'] / total_dissatisfied
df['support_s'] = df['count_s'] / total_satisfied
df['leverage']  = df['support_d'] - df['support_s']
df['growth']    = (df['support_d'] + eps) / (df['support_s'] + eps)
df['interest']  = df['leverage'] * np.log1p(df['growth'])
```

Everything stays in place so you can inspect the full distribution of interest scores, plot histograms, etc.

Only once you've got those numbers do you move on to **filtering**—either an optional quick pre-filter (e.g. drop patterns with extremely low support or leverage) or the formal **Statistical Filtering Stage** (χ^2/Fisher's tests plus p-value adjustment).

Keep the two steps distinct:

1. **Compute metrics** for 100 % of your patterns (no removal).
2. **Filter** down to the statistically significant, business-relevant subset.

You said:

Then the remaining cycles and paths we can put into logical groups right? How to find these logical groups, if I have a path like logo n, account-summary, account-summary-credit-card to which logical group it goes ?

ChatGPT said:

Thought for a few seconds

Yes—once you've filtered down to your high-interest cycles and paths, the next step is to tag each one with a business-relevant "journey" or category. Here's a recipe:

---

1. Define Your Category Taxonomy

In banking/insurance you might start with categories like:

Category	Examples of pages
Authentication & Security	login, logout, 2FA, forgot-password
Account Overview	account-summary, balances, statements
Card Services	credit-card-overview, add-card, replace-card
Payments & Transfers	transfer-funds, bill-pay, payee-management
Loans & Credit	loan-application, loan-status, credit-score
Claims & Support	claims-start, claims-status, chat-support, faq

You can refine this list based on your page catalog.

---

2. Build a Page→Category Lookup

Create a simple mapping table (or dict) from each **page ID** or **URL pattern** to one of your categories:

python
CopyEdit
```
page_to_cat = {
  'login':             'Authentication & Security',
  'forgot-password':  'Authentication & Security',
  'account-summary':  'Account Overview',
  'account-summary-credit-card': 'Card Services',
  'transfer-funds':   'Payments & Transfers',
  # …and so on for every page type you care about
}
```

---

3. Tag Each Path or Cycle

For a given sequence like

bash
CopyEdit
```
['login',
 'account-summary',
 'account-summary-credit-card']
```

you'd map each step:

- `login` → **Authentication & Security**
- `account-summary` → **Account Overview**
- `account-summary-credit-card` → **Card Services**

and collect `['Authentication & Security', 'Account Overview', 'Card Services']`.

---

4. Choose a "Primary" Category for the Sequence

There are a few sensible heuristics:

1. **Majority Vote**
   Pick the category that appears most often.
   - Here each appears once, so it's a tie.
2. **First Relevant Step**
   Use the first non-trivial step after authentication.
   - In this path, after `login`, the user goes to `account-summary`, so you'd tag the path as **Account Overview**.
3. **Domain Priority**
   If you know "Account Overview" issues are higher business priority than "Card Services," break ties by that ranking.
4. **Multi-Label**
   If a path truly spans areas, you can tag it with all categories (e.g. `['Auth', 'Account', 'Card']`) and later drill in per category.

In practice, **First Relevant Step** or **Majority Vote** tend to be simplest. For your example, I'd recommend:

> **Primary Category** = **Account Overview**
> (since the core of the flow is "viewing the account summary," and the credit-card page is just one sub-tab)

---

5. Automate & Summarize

Once you've tagged every high-interest pattern:

python
CopyEdit
```
def primary_category(seq):
    cats = [page_to_cat[p] for p in seq if p in page_to_cat]
    # remove "Authentication" if you want to focus on business flows
    cats = [c for c in cats if c != 'Authentication & Security']
    return cats[0]                # or use
collections.Counter(cats).most_common(1)[0][0]
```

```
df['primary_category'] = df['pattern_steps'].apply(primary_category)
```

Then group and aggregate:

python
CopyEdit
```
category_report = (
  df.groupby('primary_category')
    .agg({
      'interest':       'mean',
      'dropoff_delta':  'sum',
      'pattern':        'count'
    })
    .sort_values('dropoff_delta', ascending=False)
)
```

This will tell you, for example, that **Account Overview** paths contribute the largest share of pain-score, so your UX team can prioritize improvements there.

Recap

1. **Define** a small set of journey categories.
2. **Map** each page (URL or ID) to one category.
3. **Tag** every path/cycle by translating its steps into categories.
4. **Choose** a "primary" category per sequence (first, most frequent, or priority-ranked).
5. **Aggregate** your interest/drop-off metrics by category to see which business area is most painful.

That way, even if you have dozens of patterns, you'll boil them down to a handful of actionable buckets (e.g. "Account Overview," "Payments," "Claims") that your product and design teams can tackle in priority order.

You said:

How we are doing the statistical significance test here ?

ChatGPT said:

The statistical test lives in your **Statistical Filtering Stage**. Here's exactly what you do, step by step:

---

1. Build the 2×2 Contingency Table for Pattern PPP

For each pattern PPP, you have:

	PPP present	**PPP absent**
Dissatisfied (D)	aaa	Nd−aN_d - aNd−a
Satisfied (S)	bbb	Ns−bN_s - bNs−b

- a=countda = \text{count}_da=countd = number of dissatisfied sessions containing PPP.
- b=countsb = \text{count}_sb=counts = number of satisfied sessions containing PPP.
- NdN_dNd, NsN_sNs = total number of dissatisfied / satisfied sessions.

---

2. Choose Your Test

Fisher's Exact Test
Use when any of the four cells {a, b, Nd−a, Ns−b}\{a,\,b,\,N_d - a,\,N_s - b\}{a,b,Nd−a,Ns−b} is small (e.g.\ < 5).

python
CopyEdit
```
from scipy.stats import fisher_exact
table = [[a, N_d - a],
         [b, N_s - b]]
p_value = fisher_exact(table)[1]
```

1.

Chi-square Test (χ²)
Use when all cell counts are reasonably large (e.g.\ ≥ 5).

python
CopyEdit
```
from scipy.stats import chi2_contingency
p_value = chi2_contingency(table, correction=False)[1]
```

2.

You pick one per pattern based on its counts.

---

3. Collect Raw ppp-values

Run your chosen test for **every** pattern (or only your pre-filtered top-N by interest). Store the resulting raw ppp-value in a new column:

python
CopyEdit
```
df['p_raw'] = df.apply(lambda row: compute_p_value(row), axis=1)
```

---

4. Correct for Multiple Comparisons

You're testing thousands of patterns, so you must control your false discovery rate. A common approach is **Benjamini–Hochberg**:

python
CopyEdit
```
import numpy as np

p_raw = df['p_raw'].values
n = len(p_raw)
order = np.argsort(p_raw)
ranks = np.empty_like(order)
ranks[order] = np.arange(1, n+1)  # 1-based ranks

# BH adjustment:
df['p_adj'] = np.minimum(1,
```

```
    (p_raw * n / ranks)
    .cumsum()[np.argsort(order)]  # ensure monotonicity
)
```

Alternatively, you can use `statsmodels.stats.multitest.multipletests`.

---

5. Filter Patterns

Drop any pattern where:

- `p_adj > α` (e.g. 0.01 or 0.05), **or**
- support_d is below your practical minimum (e.g. 0.5%), **or**
- leverage is below your business threshold.

What remains are patterns whose difference in occurrence between D and S is **both** large (high interest score) **and** statistically unlikely to be due to chance.

Putting It All Together

python
CopyEdit

```python
from scipy.stats import chi2_contingency, fisher_exact
import numpy as np

def compute_p_value(row):
    a, b = row['count_d'], row['count_s']
    table = [[a, total_d - a],
             [b, total_s - b]]
    if min(a, b, total_d-a, total_s-b) < 5:
        return fisher_exact(table)[1]
    else:
        return chi2_contingency(table, correction=False)[1]

# 1) Compute p-values
```

```
df['p_raw'] = df.apply(compute_p_value, axis=1)

# 2) Adjust with Benjamini–Hochberg
p_raw = df['p_raw'].to_numpy()
n = len(p_raw)
order = np.argsort(p_raw)
ranks = np.empty(n, int)
ranks[order] = np.arange(1, n+1)
bh_values = (p_raw * n / ranks)
# enforce monotonicity
bh_cumsum = np.minimum.accumulate(sorted(bh_values,
reverse=True))[::-1]
df['p_adj'] = bh_cumsum[np.argsort(order)]
df['p_adj'] = np.clip(df['p_adj'], 0, 1)

# 3) Filter
alpha = 0.01
df_sig = df.query('p_adj <= @alpha and support_d >= 0.005 and leverage
> 0.01')
```

By following these steps, you ensure that your final list of cycles and paths is not only **high-interest** but also **statistically robust**.