Generating choice sets for a full factorial design with 16 profiles and deciding which sets to expose to each respondent involves careful planning to ensure the experiment is efficient, balanced, and practical for respondents. Below, I’ll explain how to generate choice sets, how to assign them to respondents, and how this applies to your scenario from the `dcm01.py` code. I’ll also provide practical steps and considerations to address your question comprehensively.

# Understanding the Context

- Full Factorial Design: You have 16 profiles, representing all combinations of:

- `Size\_Performance` (2 levels: "High performance..." and "Compact Frame size...").

- `Advanced\_Feature` (8 levels: "Higher Electrical life...", "Visible Health indication...", etc.).

- `Price` is assigned to each combination (e.g., 112200, 112500, etc.), making it part of the profile definition.

- This results in \(2 \times 8 = 16\) profiles, covering all combinations of the two categorical attributes.

- Choice Sets: A choice set is a small group of profiles (e.g., 2–4 profiles) presented to a respondent in a single choice task, where they select one profile (or sometimes none).

- Goal: Generate choice sets that allow you to estimate respondent preferences (utilities) for all attribute levels while keeping the task manageable for respondents. Then, assign these choice sets to respondents to ensure coverage of the full factorial design.

# Step-by-Step Process to Generate Choice Sets

1. Determine Choice Set Size:

- Decide how many profiles to include in each choice set. Common sizes are 2, 3, or 4 profiles per set, as larger sets (e.g., 16 profiles) are cognitively overwhelming and reduce response quality.

- For your case (16 profiles), let’s assume 3 profiles per choice set as a reasonable balance between information collection and respondent burden.

- Consideration: Smaller sets (e.g., 2) require more tasks to cover all profiles, while larger sets (e.g., 4) increase complexity per task.

2. Calculate Number of Choice Sets:

- With 16 profiles, you want enough choice sets to cover all profiles and their attribute combinations across the experiment, ensuring the model can estimate utilities for all levels of `Size\_Performance`, `Advanced\_Feature`, and `Price`.

- The total number of possible ways to choose 3 profiles from 16 is given by the combination formula \( \binom{n}{k} = \frac{n!}{k!(n-k)!} \):

\[

\binom{16}{3} = \frac{16 \times 15 \times 14}{3 \times 2 \times 1} = 560 \text{ possible choice sets.}

\]

- You don’t need all 560 sets, but you need enough to ensure statistical efficiency (i.e., all attribute levels appear frequently and in varied combinations).

3. Design the Choice Sets:

- Goal: Create choice sets that are balanced and efficient, meaning:

- Each profile appears roughly the same number of times across all choice sets.

- Attribute levels (e.g., "High performance...", "Higher Electrical life...") are well-represented and not confounded (e.g., avoid sets where all profiles have the same `Size\_Performance`).

- The design allows estimation of main effects (and possibly interactions) for `Size\_Performance`, `Advanced\_Feature`, and `Price`.

- Methods:

- Random Selection: Randomly select subsets of 3 profiles, but check for balance (e.g., ensure no attribute level is over- or under-represented). This is simple but may not be efficient.

- Orthogonal Design: Use a fractional factorial or orthogonal array to select profiles, ensuring attribute levels are balanced. However, with only 16 profiles, the full factorial itself is small, so orthogonality is less critical.

- D-efficient Design: Use specialized software (e.g., Sawtooth, JMP, or Python’s `choice\_model` libraries) to generate choice sets that maximize statistical efficiency (D-efficiency) for estimating utilities. This accounts for the conditional logit model’s requirements.

- Blocking: Divide the choice sets into blocks so each respondent sees a subset of tasks, reducing fatigue.

- **Practical Approach for this Case:**

- Suppose you want each respondent to complete 4 choice tasks (a reasonable number to avoid fatigue).

- To cover 16 profiles with 3 profiles per set, each profile should appear enough times to ensure reliable utility estimates. A rough guideline:

- If each profile appears ~3–4 times across all choice sets, and each set has 3 profiles, you’d need:

\[

\text{Total profile appearances} = 16 \times 4 = 64.

\]

\[

\text{Number of choice sets} = \frac{64}{3} \approx 22 \text{ sets.}

\]

- Generate ~24 choice sets (to round up) with 3 profiles each, ensuring all 16 profiles appear multiple times and attribute levels are balanced.

4. Example Choice Set Generation:

- Use a Python script to generate balanced choice sets. Here’s a simplified approach:

```python

import random

import pandas as pd

from itertools import combinations

# Assume profiles are indexed 0 to 15

profiles = list(range(16))

# Generate all possible combinations of 3 profiles

all\_combinations = list(combinations(profiles, 3))

# Randomly select 24 choice sets (or use a design algorithm for efficiency)

random.seed(42) # For reproducibility

selected\_sets = random.sample(all\_combinations, 24)

# Convert to DataFrame

choice\_sets = pd.DataFrame(

[(i, list(profiles)) for i, profiles in enumerate(selected\_sets)],

columns=['choice\_set\_id', 'profiles\_presented']

)

# Check balance (how many times each profile appears)

profile\_counts = {}

for profiles in choice\_sets['profiles\_presented']:

for p in profiles:

profile\_counts[p] = profile\_counts.get(p, 0) + 1

print("Profile appearances:", profile\_counts)

```

- Output might show each profile appearing 4–5 times, which is reasonably balanced.

- For better efficiency, use a package like `conjoint` or `statsmodels` to generate D-efficient designs, or software like Sawtooth Lighthouse.

5. Verify Attribute Balance:

- Map the profiles back to their attributes (`Size\_Performance`, `Advanced\_Feature`, `Price`) and ensure each level appears frequently. For example:

- `Size\_Performance`: "High performance..." and "Compact Frame size..." should each appear ~50% of the time across all sets.

- `Advanced\_Feature`: Each of the 8 levels should appear ~12.5% of the time.

- Adjust the selection if imbalances are detected (e.g., if one `Advanced\_Feature` level is missing).

# Assigning Choice Sets to Respondents

1. Determine Number of Tasks per Respondent:

- Each respondent should complete a manageable number of choice tasks to avoid fatigue. For 16 profiles, 4 tasks per respondent is reasonable (4 tasks × 3 profiles = 12 profiles seen, covering most but not all profiles per respondent).

- With 24 choice sets, you can assign:

\[

\text{Number of respondents} = \frac{24 \text{ sets}}{4 \text{ sets per respondent}} = 6 \text{ respondents (minimum)}.

\]

- In practice, you’ll want more respondents (e.g., 100, as in your synthetic data) to increase statistical power.

2. Blocking the Design:

- Divide the 24 choice sets into blocks so each respondent sees a unique subset of tasks. For example:

- With 100 respondents and 4 tasks each, you need \(100 \times 4 = 400\) task assignments.

- Since you have 24 choice sets, each set should be assigned:

\[

\frac{400}{24} \approx 16.67 \text{ times.}

\]

- Create 25 blocks of 4 choice sets each (total 100 tasks), and assign each block to 4 respondents (or adjust for balance).

- Example blocking:

- Block 1: Choice sets [0, 1, 2, 3]

- Block 2: Choice sets [4, 5, 6, 7]

- ...

- Block 25: Choice sets [20, 21, 22, 23]

- Randomly assign each respondent to one block.

3. Randomized Assignment:

- Randomly assign choice sets to respondents to avoid order effects. For example:

```python

# Assign 4 choice sets to each of 100 respondents

respondents = list(range(100))

choice\_set\_ids = list(range(24))

# Create blocks

blocks = [choice\_set\_ids[i:i+4] for i in range(0, 24, 4)] \* 17 # Repeat blocks to cover 100 respondents

blocks = blocks[:100] # Trim to 100 assignments

random.shuffle(blocks) # Randomize block order

# Assign to respondents

assignments = []

for r in respondents:

for cs in blocks[r]:

assignments.append({'respondent\_id': r, 'choice\_set': cs})

assignments\_df = pd.DataFrame(assignments)

print(assignments\_df)

```

- This assigns 4 choice sets per respondent, with some repetition of choice sets across respondents to cover all 24 sets.

4. Ensure Coverage:

- Verify that all 16 profiles and attribute levels are seen by enough respondents. With 100 respondents × 4 tasks × 3 profiles = 1200 profile exposures, each profile should appear:

\[

\frac{1200}{16} = 75 \text{ times (average).}

\]

- Check the `profiles\_presented` column in the final `choices` DataFrame to confirm balance.

# Integrating with Your Code

Your synthetic `choices` data in `dcm01.py` assumes each respondent completes 2 choice tasks, with all 16 profiles as alternatives per task:

```python

choices\_data = {

'respondent\_id': list(range(100)) \* 2,

'choice\_set': [i // 2 for i in range(200)],

'chosen\_profile': np.random.randint(0, 16, 200)

}

```

This doesn’t reflect a realistic full factorial experiment, as respondents wouldn’t choose from all 16 profiles per task. Here’s how to update it:

1. Generate Choice Sets:

- Create a list of 24 choice sets with 3 profiles each, as shown above.

- Example:

```python

choice\_sets = [

{'choice\_set': 0, 'profiles\_presented': [0, 2, 4]},

{'choice\_set': 1, 'profiles\_presented': [1, 3, 5]},

...

{'choice\_set': 23, 'profiles\_presented': [13, 14, 15]}

]

choice\_sets\_df = pd.DataFrame(choice\_sets)

```

2. Assign to Respondents:

- Assign 4 choice sets per respondent:

```python

assignments = []

for r in range(100):

# Randomly select 4 choice sets

assigned\_sets = random.sample(range(24), 4)

for cs in assigned\_sets:

profiles = choice\_sets[cs]['profiles\_presented']

# Simulate a choice (replace with real data if available)

chosen = random.choice(profiles)

assignments.append({

'respondent\_id': r,

'choice\_set': cs,

'chosen\_profile': chosen,

'profiles\_presented': profiles

})

choices = pd.DataFrame(assignments)

```

3. Update `prepare\_data`:

- Modify the `prepare\_data` method to use `profiles\_presented`:

```python

def prepare\_data(self):

self.profiles['Price'] = self.profiles['Price'] / 1000

profiles\_encoded = pd.get\_dummies(self.profiles, columns=['Size\_Performance', 'Advanced\_Feature'], drop\_first=True)

choice\_data = self.choices.merge(profiles\_encoded, left\_on='chosen\_profile', right\_index=True)

choice\_data = choice\_data.merge(self.groups, on='respondent\_id')

choice\_sets = []

for choice\_set in choice\_data['choice\_set'].unique():

respondents = choice\_data[choice\_data['choice\_set'] == choice\_set]

for \_, respondent in respondents.iterrows():

for idx in respondent['profiles\_presented']:

profile = profiles\_encoded.loc[idx]

row = profile.copy()

row['respondent\_id'] = respondent['respondent\_id']

row['choice\_set'] = choice\_set

row['group'] = respondent['group']

row['chosen'] = 1 if idx == respondent['chosen\_profile'] else 0

choice\_sets.append(row)

self.choice\_data = pd.DataFrame(choice\_sets)

return self.choice\_data

```

4. Run the Analysis:

- With the updated `choices` DataFrame and `prepare\_data` method, the conditional logit model will only consider the profiles actually presented in each choice set, making the analysis more realistic.

# Practical Considerations

- Number of Respondents: Your 100 respondents are sufficient, but more respondents (e.g., 200–300) improve precision, especially for group-level analyses (e.g., Group A vs. Group B).

- Task Load: 4 tasks per respondent is reasonable. If you reduce to 2 tasks (as in your original code), you’ll need more choice sets or respondents to cover all profiles.

- Software Tools: For real experiments, use conjoint design software (Sawtooth, Qualtrics, or R’s `AlgDesign`) to generate efficient choice sets. Python’s `choice\_model` or `conjoint` libraries are also options.

- Real vs. Synthetic Data: Your current `chosen\_profile` is random. In a real experiment, collect actual choices from respondents. If simulating, use a utility-based model to generate realistic choices.

# Example Output

After generating 24 choice sets and assigning 4 per respondent, your `choices` DataFrame might look like:

```

respondent\_id | choice\_set | chosen\_profile | profiles\_presented

0 | 0 | 2 | [0, 2, 4]

0 | 5 | 3 | [1, 3, 5]

0 | 10 | 7 | [6, 7, 8]

0 | 15 | 10 | [9, 10, 11]

1 | 1 | 5 | [1, 3, 5]

...

```

The `prepare\_data` method would generate 3 rows per choice set per respondent (e.g., 12 rows per respondent for 4 tasks), with `chosen=1` for the selected profile and `chosen=0` for others.

# Summary

To generate choice sets for your 16-profile full factorial design:

1. Choose Set Size: Use 3 profiles per choice set for manageability.

2. Generate Sets: Create ~24 choice sets with 3 profiles each, ensuring all 16 profiles and attribute levels are balanced (use random selection or D-efficient design).

3. Assign to Respondents: Assign 4 choice sets per respondent (100 respondents × 4 tasks = 400 assignments). Use blocking or random assignment to distribute the 24 sets, ensuring each set appears ~16–17 times.

4. Update Data: Modify the `choices` DataFrame to include `profiles\_presented`, and update `prepare\_data` to only include presented profiles as alternatives.

5. Validate: Check that each profile appears ~75 times across all tasks and that attribute levels are balanced.

This approach ensures your experiment leverages the full factorial design while keeping tasks practical for respondents. If you need help implementing the Python code, designing the sets, or analyzing real data, let me know!

To generate a choice experiment design with your 16 profiles for choice sets containing 3 profiles each using `statsmodels`, we need to create a balanced and efficient design suitable for a discrete choice experiment. However, `statsmodels` itself doesn’t have a direct built-in function for generating choice experiment designs (like D-efficient designs or choice sets). Instead, it’s typically used for modeling choice data (e.g., conditional logit models, as in your `dcm01.py` code). For generating choice designs, other tools like Sawtooth, JMP, or Python libraries such as `pyDOE` or `conjoint` are more specialized. However, we can approximate a balanced design using Python and ensure compatibility with your existing code, then use `statsmodels` indirectly to validate or analyze the design later.

Since your request specifically mentions `statsmodels`, I’ll assume you want a method that integrates with your `statsmodels`-based analysis pipeline. I’ll generate a design with 16 profiles, create choice sets of 3 profiles each, and ensure the design is practical for your full factorial setup. I’ll use a combinatorial approach to generate choice sets and check balance, as `statsmodels` doesn’t directly generate choice designs. Then, I’ll show how the resulting data can be used with `statsmodels` for modeling, aligning with your code.

# Step-by-Step Approach

1. Understand the Profiles:

- You have 16 profiles from a full factorial design:

- `Size\_Performance`: 2 levels ("High performance Ics=Icu=Icw 66kA for 1sec", "Compact Frame size 1600A Icu=66kA, Icw 50kA for 1sec").

- `Advanced\_Feature`: 8 levels (e.g., "Higher Electrical life...", "Visible Health indication...", etc.).

- `Price`: Continuous, assigned to each profile (e.g., 112200, 112500, etc.).

- This yields \(2 \times 8 = 16\) profiles, as provided in your `profiles` DataFrame.

- Each choice set will include 3 profiles.

- Goal: Generate enough choice sets to cover all profiles and attribute levels, ensuring balance for estimating utilities with a conditional logit model.

2. Determine Design Parameters:

- Choice Set Size: 3 profiles per set.

- Number of Choice Sets: To cover 16 profiles, each profile should appear multiple times across choice sets for reliable estimation. A rough guideline:

- Assume each profile appears ~4 times (adjustable based on respondents and tasks).

- With 3 profiles per set, total profile appearances = \(16 \times 4 = 64\).

- Number of choice sets = \(64 / 3 \approx 21.33\), so let’s aim for ~24 choice sets to ensure coverage.

- Tasks per Respondent: Assume 4 tasks per respondent (as discussed previously), reasonable for 100 respondents.

- Respondents: Your code uses 100 respondents, so total tasks = \(100 \times 4 = 400\), meaning each choice set appears ~\(400 / 24 \approx 16.67\) times.

3. Generate Choice Sets:

- Since `statsmodels` doesn’t generate choice designs, we’ll use a combinatorial approach to create balanced choice sets, then verify balance. We’ll select subsets of 3 profiles, ensuring all profiles and attribute levels appear frequently.

- We’ll use Python’s `itertools.combinations` to generate candidate sets and filter for balance, mimicking what a D-efficient design might achieve.

4. Assign to Respondents:

- Distribute the choice sets across 100 respondents, with each seeing 4 tasks.

5. Integrate with `statsmodels`:

- Prepare the data for a `statsmodels` conditional logit model, ensuring compatibility with your `DiscreteChoiceAnalyzer` class.

# Implementation

Below is a Python script that generates the choice design, creates a `choices` DataFrame, and updates your `prepare\_data` method. I’ll use your `profiles` DataFrame and ensure the output works with `statsmodels.api.Logit`.

```python

import pandas as pd

import numpy as np

import random

from itertools import combinations

import statsmodels.api as sm

# Your profiles data (from dcm01.py)

profiles\_data = {

'Size\_Performance': [

'High performance Ics=Icu=Icw 66kA for 1sec',

'Compact Frame size 1600A Icu=66kA, Icw 50kA for 1sec',

'High performance Ics=Icu=Icw 66kA for 1sec',

'Compact Frame size 1600A Icu=66kA, Icw 50kA for 1sec',

'High performance Ics=Icu=Icw 66kA for 1sec',

'Compact Frame size 1600A Icu=66kA, Icw 50kA for 1sec',

'High performance Ics=Icu=Icw 66kA for 1sec',

'Compact Frame size 1600A Icu=66kA, Icw 50kA for 1sec',

'High performance Ics=Icu=Icw 66kA for 1sec',

'Compact Frame size 1600A Icu=66kA, Icw 50kA for 1sec',

'High performance Ics=Icu=Icw 66kA for 1sec',

'Compact Frame size 1600A Icu=66kA, Icw 50kA for 1sec',

'High performance Ics=Icu=Icw 66kA for 1sec',

'Compact Frame size 1600A Icu=66kA, Icw 50kA for 1sec',

'High performance Ics=Icu=Icw 66kA for 1sec',

'Compact Frame size 1600A Icu=66kA, Icw 50kA for 1sec'

],

'Advanced\_Feature': [

'Higher Electrical life from 6,000 to 7,500 operations without maintenance',

'Higher Electrical life from 6,000 to 7,500 operations without maintenance',

'Visible Health indication (Breaker Status, trip cause Indication - OL,SC,GF)',

'Visible Health indication (Breaker Status, trip cause Indication - OL,SC,GF)',

'Scalable connectivity at breaker level- Basic Modbus (Breaker Status, contorl, terminal temp alarm)',

'Scalable connectivity at breaker level- Basic Modbus (Breaker Status, contorl, terminal temp alarm)',

'Current Measuremnt and time stamped fault records on mobile app',

'Current Measuremnt and time stamped fault records on mobile app',

'Access trip unit data during tripping events without supply',

'Access trip unit data during tripping events without supply',

'Scalable connectivity at breaker level- Modbus Ethernet',

'Scalable connectivity at breaker level- Modbus Ethernet',

'Operator Safety - Arc Flash reduction during maintenance',

'Operator Safety - Arc Flash reduction during maintenance',

'Terminal Temperature threshold monitoring',

'Terminal Temperature threshold monitoring'

],

'Price': [

112200, 112500, 112500, 112800, 115600, 115900, 112500, 112800,

112200, 112500, 112650, 112950, 112850, 113150, 116350, 116650

]

}

profiles = pd.DataFrame(profiles\_data)

# Step 1: Generate choice sets (3 profiles each)

def generate\_choice\_sets(n\_profiles, set\_size, n\_sets):

"""

Generate choice sets with specified size and number, ensuring balance.

"""

profile\_indices = list(range(n\_profiles))

all\_combinations = list(combinations(profile\_indices, set\_size))

# Randomly select n\_sets, but aim for balance

selected\_sets = []

profile\_counts = {i: 0 for i in profile\_indices}

# Iteratively select sets to balance profile appearances

random.seed(42)

while len(selected\_sets) < n\_sets:

candidate = random.choice(all\_combinations)

temp\_counts = profile\_counts.copy()

for p in candidate:

temp\_counts[p] += 1

# Accept if it doesn't imbalance profiles too much

if max(temp\_counts.values()) - min(temp\_counts.values()) <= 2:

selected\_sets.append(candidate)

for p in candidate:

profile\_counts[p] += 1

# Remove used combination to avoid duplicates

all\_combinations.remove(candidate)

# Create DataFrame

choice\_sets = pd.DataFrame({

'choice\_set': range(n\_sets),

'profiles\_presented': selected\_sets

})

return choice\_sets, profile\_counts

# Generate 24 choice sets with 3 profiles each

n\_profiles = 16

set\_size = 3

n\_sets = 24

choice\_sets, profile\_counts = generate\_choice\_sets(n\_profiles, set\_size, n\_sets)

print("Profile appearances:", profile\_counts)

# Step 2: Assign choice sets to respondents (100 respondents, 4 tasks each)

n\_respondents = 100

tasks\_per\_respondent = 4

assignments = []

for r in range(n\_respondents):

# Randomly select 4 choice sets

assigned\_sets = random.sample(list(choice\_sets['choice\_set']), tasks\_per\_respondent)

for cs in assigned\_sets:

profiles = choice\_sets.loc[cs, 'profiles\_presented']

# Simulate a choice (replace with real data if available)

chosen = random.choice(profiles)

assignments.append({

'respondent\_id': r,

'choice\_set': cs,

'chosen\_profile': chosen,

'profiles\_presented': profiles

})

choices = pd.DataFrame(assignments)

# Step 3: Reuse your groups data

groups\_data = {

'respondent\_id': range(100),

'group': ['Group A'] \* 50 + ['Group B'] \* 50

}

groups = pd.DataFrame(groups\_data)

# Step 4: Update DiscreteChoiceAnalyzer to handle profiles\_presented

class DiscreteChoiceAnalyzer:

def \_\_init\_\_(self, profiles, choices, groups):

self.profiles = profiles

self.choices = choices

self.groups = groups

self.model = None

self.utilities = None

self.feature\_importance = None

def prepare\_data(self):

"""

Prepare data for conditional logit model, using profiles\_presented.

"""

# Scale Price

self.profiles['Price'] = self.profiles['Price'] / 1000

profiles\_encoded = pd.get\_dummies(self.profiles, columns=['Size\_Performance', 'Advanced\_Feature'], drop\_first=True)

# Merge choice data with profiles (for validation)

choice\_data = self.choices.merge(profiles\_encoded, left\_on='chosen\_profile', right\_index=True)

choice\_data = choice\_data.merge(self.groups, on='respondent\_id')

# Create choice set data

choice\_sets = []

for choice\_set in choice\_data['choice\_set'].unique():

respondents = choice\_data[choice\_data['choice\_set'] == choice\_set]

for \_, respondent in respondents.iterrows():

for idx in respondent['profiles\_presented']:

profile = profiles\_encoded.loc[idx]

row = profile.copy()

row['respondent\_id'] = respondent['respondent\_id']

row['choice\_set'] = choice\_set

row['group'] = respondent['group']

row['chosen'] = 1 if idx == respondent['chosen\_profile'] else 0

choice\_sets.append(row)

self.choice\_data = pd.DataFrame(choice\_sets)

return self.choice\_data

def fit\_model(self):

"""

Fit conditional logit model using statsmodels.

"""

X\_cols = [col for col in self.choice\_data.columns if col not in ['respondent\_id', 'choice\_set', 'chosen', 'group']]

X = self.choice\_data[X\_cols]

y = self.choice\_data['chosen']

X = sm.add\_constant(X)

self.model = sm.Logit(y, X).fit(disp=0)

self.utilities = self.model.params

return self.utilities

def calculate\_feature\_importance(self):

"""

Calculate feature importance.

"""

size\_utils = self.utilities.filter(like='Size\_Performance').abs()

adv\_feature\_utils = self.utilities.filter(like='Advanced\_Feature').abs()

price\_util = abs(self.utilities['Price']) \* (self.profiles['Price'].max() - self.profiles['Price'].min())

importance = {

'Size\_Performance': size\_utils.max() - size\_utils.min(),

'Advanced\_Feature': adv\_feature\_utils.max() - adv\_feature\_utils.min(),

'Price': price\_util

}

total = sum(importance.values())

self.feature\_importance = {k: v / total for k, v in importance.items()}

return self.feature\_importance

# Step 5: Run the analysis

analyzer = DiscreteChoiceAnalyzer(profiles, choices, groups)

choice\_data = analyzer.prepare\_data()

utilities = analyzer.fit\_model()

importance = analyzer.calculate\_feature\_importance()

print("Utilities:\n", utilities)

print("Feature Importance:\n", importance)

# Step 6: Save data for further use

profiles.to\_excel("profiles\_updated.xlsx")

choices.to\_excel("choices\_updated.xlsx")

groups.to\_excel("groups\_updated.xlsx")

```

# Explanation of the Code

1. Profile Data:

- Reused your `profiles` DataFrame with 16 profiles, ensuring it matches the full factorial design (\(2 \times 8\)).

2. Choice Set Generation:

- The `generate\_choice\_sets` function creates 24 choice sets, each with 3 profiles.

- Uses `itertools.combinations` to generate all possible sets (\(\binom{16}{3} = 560\)).

- Selects 24 sets iteratively, ensuring each profile appears roughly the same number of times (target ~4 appearances per profile).

- Checks balance by tracking `profile\_counts` (e.g., each profile appears 4–5 times).

- Output: `choice\_sets` DataFrame with columns `choice\_set` (0 to 23) and `profiles\_presented` (list of 3 profile indices).

3. Assignment to Respondents:

- Assigns 4 choice sets to each of 100 respondents using random sampling.

- For each assignment, simulates a `chosen\_profile` by randomly selecting one profile from `profiles\_presented` (replace with real data in a real experiment).

- Creates a `choices` DataFrame with:

- `respondent\_id`: 0 to 99.

- `choice\_set`: ID of the choice set (0 to 23).

- `chosen\_profile`: Index of the chosen profile.

- `profiles\_presented`: List of 3 profile indices shown in the task.

4. Updated `DiscreteChoiceAnalyzer`:

- Modified `prepare\_data` to use `profiles\_presented`, generating rows only for the 3 profiles in each choice set, not all 16.

- For each respondent and `choice\_set`, creates 3 rows in `choice\_data` (one per profile in `profiles\_presented`), with `chosen=1` for the chosen profile and `chosen=0` for others.

- `fit\_model` uses `statsmodels.api.Logit` to fit the conditional logit model, consistent with your original code.

- `calculate\_feature\_importance` computes relative importance of features, unchanged from your code.

5. Balance Check:

- The `profile\_counts` output shows how many times each profile appears (e.g., `{0: 4, 1: 5, 2: 4, ...}`).

- To check attribute balance, you can map profiles back to attributes:

```python

# Check attribute level balance

attribute\_counts = {'Size\_Performance': {}, 'Advanced\_Feature': {}}

for cs in choice\_sets['profiles\_presented']:

for p in cs:

sp = profiles.loc[p, 'Size\_Performance']

af = profiles.loc[p, 'Advanced\_Feature']

attribute\_counts['Size\_Performance'][sp] = attribute\_counts['Size\_Performance'].get(sp, 0) + 1

attribute\_counts['Advanced\_Feature'][af] = attribute\_counts['Advanced\_Feature'].get(af, 0) + 1

print("Attribute appearances:", attribute\_counts)

```

- Expect ~36 appearances per `Size\_Performance` level (\(24 \times 3 / 2\)) and ~9 per `Advanced\_Feature` level (\(24 \times 3 / 8\)).

6. Output:

- Saves `profiles`, `choices`, and `groups` to Excel for transparency.

- Prints utilities and feature importance from the model fit.

- Note: Since `chosen\_profile` is random, utilities and importance are illustrative. Real data would yield meaningful results.

# Why `statsmodels` Isn’t Used for Design Generation

- `statsmodels` excels at statistical modeling (e.g., `Logit` for choice models) but lacks functions for generating choice experiment designs. Design generation requires combinatorial or optimization algorithms (e.g., D-efficiency), typically handled by:

- Sawtooth Lighthouse: Industry-standard for conjoint designs.

- R’s AlgDesign: Generates efficient designs.

- Python’s pyDOE or conjoint: Limited but useful for simple designs.

- I used a combinatorial approach with balance checks to mimic a practical design, ensuring the output is compatible with `statsmodels` for analysis.

# Limitations and Notes

- Random Choices: The `chosen\_profile` is randomly generated, so the model results are not meaningful. In a real experiment, collect actual respondent choices.

- Design Efficiency: The combinatorial approach is not fully D-efficient. For a production experiment, use specialized software to optimize the design (e.g., minimize variance in utility estimates).

- Attribute Balance: The code ensures approximate balance, but you should verify:

- `Size\_Performance`: Each level appears ~50% of the time.

- `Advanced\_Feature`: Each of 8 levels appears ~12.5% of the time.

- Number of Sets: 24 sets is a starting point. You can adjust to 18 or 30 sets based on statistical power needs (e.g., more sets for precise group comparisons).

# Example Output

Running the code might produce:

```

Profile appearances: {0: 4, 1: 5, 2: 4, 3: 5, 4: 4, 5: 4, 6: 5, 7: 4, 8: 4, 9: 5, 10: 4, 11: 4, 12: 5, 13: 4, 14: 4, 15: 5}

Attribute appearances: {

'Size\_Performance': {

'High performance Ics=Icu=Icw 66kA for 1sec': 35,

'Compact Frame size 1600A Icu=66kA, Icw 50kA for 1sec': 37

},

'Advanced\_Feature': {

'Higher Electrical life...': 9,

'Visible Health indication...': 10,

'Scalable connectivity Basic Modbus...': 8,

'Current Measurement...': 9,

'Access trip unit data...': 9,

'Scalable connectivity Modbus Ethernet...': 8,

'Operator Safety...': 9,

'Terminal Temperature...': 10

}

}

Utilities: [Random values due to simulated choices]

Feature Importance: [Illustrative values]

```

The `choices` DataFrame might look like:

```

respondent\_id choice\_set chosen\_profile profiles\_presented

0 0 0 2 [0, 2, 4]

1 0 5 3 [1, 3, 5]

2 0 10 7 [6, 7, 8]

3 0 15 10 [9, 10, 11]

4 1 1 5 [1, 3, 5]

...

```

# Using with `statsmodels`

The generated `choice\_data` from `prepare\_data` is directly compatible with `statsmodels.api.Logit`. The model fits utilities for:

- Dummy variables for `Size\_Performance` (1 coefficient).

- Dummy variables for `Advanced\_Feature` (7 coefficients).

- Continuous `Price` (1 coefficient).

- Constant term.

If you have real choice data, replace the random `chosen\_profile` with actual responses, and the model will estimate meaningful utilities.

# Recommendations

- Real Data: Collect actual respondent choices for accurate results.

- Design Optimization: For a production experiment, use Sawtooth or R’s `AlgDesign` for D-efficient designs.

- Increase Sets: If group differences (Group A vs. Group B) are critical, consider 30–36 choice sets for better precision.

- Validate Balance: After generating `choices`, verify attribute level frequencies to ensure no level is under-represented.

If you need help refining the design, integrating real data, or running the analysis, let me know!