Introduction

This assignment builds directly on the work from Assignment 2. The primary objective is to use the Bayesian networks (both the original and the synthetically balanced versions) to identify which specific combinations of demographic variables are most predictive of the four different purchase intention types. By calculating conditional probabilities, this analysis will reveal the most likely customer profiles for each intention category and explore how data balancing techniques impact these demographic insights.

The goal of this assignment is to use the Bayesian network models we built in Assignment 2 to determine which combinations of demographic variables are most likely to lead to each of the four purchase intention types: `Buy`, `Wait`, `Browse`, and `Leave`. We will perform this analysis on networks trained on both the original (unbalanced) data and the synthetically balanced data to critically evaluate the impact of data balancing on probabilistic inference.

Objective:

To determine which combinations of demographic variables are most likely to result in each of the four purchase intention types by analysing both balanced and unbalanced Bayesian networks developed in Assignment 2.

- Assignment 2
- Paper: Data modelling of subsistence retail consumer purchase behavior in South Africa by Author Zulu and Nkuna
- Data: Subsistence Retail Consumer Data

Dependencies

Installing Libraries

```
# Install pgmpy for Bayesian network analysis and other essential libraries
!pip install pandas numpy pickle-mixin pgmpy matplotlib seaborn bnlearn networkx openpyxl
```

Show hidden output

Importing dependencies

```
# Import necessary libraries
import pickle
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from pgmpy.inference import VariableElimination
import bnlearn as bn
import itertools
```

Processing

Data and Model Preparation

Part 1: Loading Data and Pre-trained Models

In this section, we load the essential components from Assignment 2:

1. The Bayesian network model trained on the **original, unbalanced data**.
2. The Bayesian network model trained on the **SMOTE-balanced data**.
3. *(Add any other models you created, e.g., GAN-balanced model)*
4. The original DataFrame, which helps us identify the demographic variables and their possible states.

```
from google.colab import files
uploaded = files.upload()
```

```
Choose Files 4 files
SMOTE_bayesian_models.pkl(n/a) - 115665 bytes, last modified: 2025/10/07 - 100% done
ML_bayesian_models.pkl(n/a) - 143670 bytes, last modified: 2025/10/07 - 100% done
GAN_bayesian_models.pkl(n/a) - 252431 bytes, last modified: 2025/10/07 - 100% done
Subsistence Retail Consumer Data.csv(text/csv) - 23559 bytes, last modified: 2025/09/07 - 100% done
Saving SMOTE_bayesian_models.pkl to SMOTE_bayesian_models.pkl
Saving ML_bayesian_models.pkl to ML_bayesian_models.pkl
Saving GAN_bayesian_models.pkl to GAN_bayesian_models.pkl
Saving Subsistence Retail Consumer Data.csv to Subsistence Retail Consumer Data.csv
```

```
# Load the dictionary of models from the single file
try:
    with open('/content/SMOTE_bayesian_models.pkl', 'rb') as f:
        smote_models = pickle.load(f)
    print("SMOTE models loaded successfully.")

    with open('/content/ML_bayesian_models.pkl', 'rb') as f:
        ml_models = pickle.load(f)
    print("ML models loaded successfully.")

    with open('/content/GAN_bayesian_models.pkl', 'rb') as f:
        gan_models = pickle.load(f)
    print("GAN models loaded successfully.")

    # Combine all models into a single dictionary for easier iteration later
    # Assuming each loaded file contains a dictionary of models, you might need to adjust this
    # based on the actual structure of your pickle files.
    models = {**smote_models, **ml_models, **gan_models}
    print("\nDictionary of all models created successfully.")
    print("Available models:", list(models.keys()))


except FileNotFoundError as e:
    print(f"Error loading model file: {e}. Please ensure the correct files are uploaded.")
    models = None # Set to None to indicate models were not loaded
except Exception as e:
    print(f"An error occurred while loading the models: {e}")
    models = None
```

```
SMOTE models loaded successfully.
ML models loaded successfully.
GAN models loaded successfully.

Dictionary of all models created successfully.
Available models: ['PI1', 'PI2', 'PI3', 'PI4']
```

```
df = pd.read_csv('/content/Subsistence Retail Consumer Data.csv')
display(df.head())
```

	Gender	Age	Marital Status	Employment Status	Level of Education	Regular Customer	Shopping frequency	E1	E2	E3	...	CT5	CT6	CT7	PV1	PV2	PV3	PI1	PI2	PI3
0	3	4	1	1	4	2	3	3	2	4	...	3	2	4	2	4	4	3	4	4
1	3	5	3	1	4	1	1	3	3	3	...	2	4	3	2	3	2	4	4	4
2	2	4	2	2	2	2	2	3	4	2	...	2	4	1	2	3	1	3	3	4
3	1	3	1	1	3	2	1	3	2	4	...	2	3	4	1	2	3	3	4	3
4	3	2	1	2	2	2	3	2	4	3	...	2	4	2	2	4	2	2	3	2

Demographic Variables:

- Extract and analyse demographic attributes.
- Prepare a matrix of all plausible combinations of these demographics for all trained networks.

```
# Define demographic variables and their corresponding mappings
demographic_vars = {
    'Gender': {1: 'Male', 2: 'Female', 3: 'Prefer not to say'},
    'Age': {1: '18–22', 2: '23–28', 3: '29–35', 4: '35–49', 5: '50–65'},
    'Marital Status': {1: 'Married', 2: 'Single', 3: 'Prefer not to say'},
    'Employment Status': {1: 'Employed', 2: 'Unemployed'},
    'Level of Education': {1: 'No formal education', 2: 'Basic education', 3: 'Diploma', 4: 'Degree', 5: 'Postgraduate degree
    'Regular Customer': {1: 'Regular', 2: 'Need-based'},
    'Shopping frequency': {1: '1–2 times per week', 2: '2–3 times per week', 3: '3–4 times per week', 4: '5–6 times per week'
}

# Create an empty list to store data for the DataFrame
demographic_data = []
```

```
# Iterate through each demographic variable and its mapping
for category, mapping in demographic_vars.items():
    # Calculate frequency and percentage for each characteristic in the current category
    value_counts = df[category].value_counts().sort_index()
    percentage = df[category].value_counts(normalize=True).sort_index() * 100

    # Add rows to the list
    for characteristic_code, frequency in value_counts.items():
        characteristic_label = mapping.get(characteristic_code, characteristic_code) # Use mapping if available
        percentage_value = percentage.get(characteristic_code, 0).round(1)
        demographic_data.append([category, characteristic_label, frequency, percentage_value])

# Create the pandas DataFrame
demographic_table_df = pd.DataFrame(demographic_data, columns=['Category', 'Characteristic', 'Frequency', 'Percentage (%)'])

# Display the DataFrame
display(demographic_table_df)
```

Show hidden output

```
from matplotlib import pyplot as plt
import seaborn as sns
figsize = (12, 1.2 * len(demographic_table_df['Category'].unique()))
plt.figure(figsize=figsize)
sns.violinplot(demographic_table_df, x='Percentage (%)', y='Category', inner='stick', palette='Dark2')
sns.despine(top=True, right=True, bottom=True, left=True)
```

Show hidden output

```
# Get the list of unique demographic categories
demographic_categories = demographic_table_df['Category'].unique()

# Generate bar and pie charts for each demographic category, to show class imbalance
for category in demographic_categories:
    # Filter the DataFrame for the current category
    category_df = demographic_table_df[demographic_table_df['Category'] == category]

    # Create a figure and axes for the charts
    fig, axes = plt.subplots(1, 2, figsize=(16, 6))

    # Create the bar chart
    sns.barplot(x='Characteristic', y='Frequency', data=category_df, ax=axes[0])
    axes[0].set_title(f'Frequency Distribution of {category}')
    axes[0].set_xlabel(category)
    axes[0].set_ylabel('Frequency')
    axes[0].tick_params(axis='x', rotation=45) # Removed ha='right'

    # Create the pie chart
    axes[1].pie(category_df['Percentage (%)'], labels=category_df['Characteristic'], autopct='%1.1f%%', startangle=90, colors=s
    axes[1].set_title(f'Percentage Distribution of {category}')
    axes[1].axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.

    plt.tight_layout()
    plt.show()
```

Show hidden output

Part 2: Probabilistic Inference and Analysis

Now for the core of the assignment. We will systematically query our models to find the probability of each purchase intention (`Buy`, `Wait`, `Browse`, `Leave`) given different demographic profiles.

To do this, we'll first identify our **demographic variables** (our evidence) and our **target variable** (`Intention`). We will then perform inference for every possible combination of demographic states.

```
# Define the demographic variables and their states based on the loaded data
# We'll use the unique values from the original DataFrame for each demographic column
demographic_vars_and_states = {
    col: df[col].unique().tolist() for col in demographic_vars.keys()
}

# Generate all possible combinations of demographic states
# The result is a list of dictionaries, where each dictionary represents a unique demographic profile
all_demographic_combinations = [
    dict(zip(demographic_vars_and_states.keys(), combination))
    for combination in itertools.product(*demographic_vars_and_states.values())
]
```

```
print(f"Generated {len(all_demographic_combinations)} unique demographic combinations.")
# Display the first 10 combinations as an example
print("\nFirst 10 demographic combinations:")
for i, combo in enumerate(all_demographic_combinations[:10]):
    print(combo)

# You can now use this list of combinations for inference on your Bayesian networks.
```

```
Generated 4500 unique demographic combinations.

First 10 demographic combinations:
{'Gender': 3, 'Age': 4, 'Marital Status': 1, 'Employment Status': 1, 'Level of Education': 4, 'Regular Customer': 2, 'Shopping
{'Gender': 3, 'Age': 4, 'Marital Status': 1, 'Employment Status': 1, 'Level of Education': 4, 'Regular Customer': 2, 'Shopping
{'Gender': 3, 'Age': 4, 'Marital Status': 1, 'Employment Status': 1, 'Level of Education': 4, 'Regular Customer': 2, 'Shopping
{'Gender': 3, 'Age': 4, 'Marital Status': 1, 'Employment Status': 1, 'Level of Education': 4, 'Regular Customer': 2, 'Shopping
{'Gender': 3, 'Age': 4, 'Marital Status': 1, 'Employment Status': 1, 'Level of Education': 4, 'Regular Customer': 2, 'Shopping
{'Gender': 3, 'Age': 4, 'Marital Status': 1, 'Employment Status': 1, 'Level of Education': 4, 'Regular Customer': 1, 'Shopping
{'Gender': 3, 'Age': 4, 'Marital Status': 1, 'Employment Status': 1, 'Level of Education': 4, 'Regular Customer': 1, 'Shopping
{'Gender': 3, 'Age': 4, 'Marital Status': 1, 'Employment Status': 1, 'Level of Education': 4, 'Regular Customer': 1, 'Shopping
{'Gender': 3, 'Age': 4, 'Marital Status': 1, 'Employment Status': 1, 'Level of Education': 4, 'Regular Customer': 1, 'Shopping
{'Gender': 3, 'Age': 4, 'Marital Status': 1, 'Employment Status': 1, 'Level of Education': 4, 'Regular Customer': 1, 'Shopping
```

```
# Define the target and evidence variables based on your dataset columns
# Please adjust this list to match the exact column names in your DataFrame
# Define demographic variables and their corresponding mappings
demographic_vars = {
    'Gender': {1: 'Male', 2: 'Female', 3: 'Prefer not to say'},
    'Age': {1: '18–22', 2: '23–28', 3: '29–35', 4: '35–49', 5: '50–65'},
    'Marital Status': {1: 'Married', 2: 'Single', 3: 'Prefer not to say'},
    'Employment Status': {1: 'Employed', 2: 'Unemployed'},
    'Level of Education': {1: 'No formal education', 2: 'Basic education', 3: 'Diploma', 4: 'Degree', 5: 'Postgraduate degree
    'Regular Customer': {1: 'Regular', 2: 'Need-based'},
    'Shopping frequency': {1: '1–2 times per week', 2: '2–3 times per week', 3: '3–4 times per week', 4: '5–6 times per week',
}
# The target variable will be one of the PI columns, depending on which model is being used.
# We will set a placeholder here, and the actual target will be determined in the inference loop.
# For now, let's assume we are targeting PI4 for the example and checks.
target_variable = 'PI4' # Placeholder, will be adjusted in the inference loop

# Ensure the variables exist in the models
if models is not None:
    for model_name, model_info in models.items():
        # Access the actual model object from the dictionary
        model = model_info.get('model')
        if model:
            model_vars = set(model.nodes())
            # Check if demographic variables are a subset and the relevant PI variable exists
            if not set(demographic_vars.keys()).issubset(model_vars) or model_name not in model_vars:
                print(f"Warning: Not all demographic variables or target variable ({model_name}) found in the {model_name} m
                print(f"Model contains: {model_vars}")
        else:
            print(f"Warning: Model object not found for {model_name}.")
else:
    print("Models were not loaded. Please ensure 'all_pi_models.pkl' is uploaded and loaded successfully.")


# Get the possible states for our target variables from the original data (PI1, PI2, PI3, PI4)
intention_states_map = {
    f'PI{i}': df[f'PI{i}'].unique().tolist() for i in range(1, 5)
}
print(f"Purchase intention states: {intention_states_map}")


def perform_inference(model, target_var, evidence_vars):
    """
    Performs inference on the Bayesian network model.

    Args:
        model: The Bayesian network model object.
        target_var: The target variable for inference (e.g., 'PI1').
        evidence_vars: A dictionary of evidence variables and their states.

    Returns:
        A dictionary of probabilities for the target variable states.
    """
    # Initialize the inference engine
    inference_engine = VariableElimination(model)

    # Perform the query to get the conditional probability distribution
    # We query the target variable given the evidence
    result_cpd = inference_engine.query(variables=[target_var], evidence=evidence_vars)
```

```
        # Extract the probability values into a clean dictionary
        # Use the states from the intention_states_map for the specific target_var
        if target_var in intention_states_map:
             states = intention_states_map[target_var]
             probabilities = {state: result_cpd.values[i] for i, state in enumerate(states)}
             return probabilities
        else:
             print(f"Error: Target variable {target_var} not found in intention_states_map.")
             return None


# --- Let's test it with one example ---
# Make sure to use integer codes for evidence based on the data
example_evidence = {'Gender': 2, 'Age': 3} # Example: Female (2), Age 29-35 (3)
# Use one of the actual PI target variables for the test
test_target_variable = 'PI4' # Example target
if models and 'PI4' in models and models['PI4'].get('model'):
    try:
        test_result = perform_inference(models['PI4']['model'], test_target_variable, example_evidence)
        print(f"\n--- Test Inference ---\nModel: PI4\nEvidence: {example_evidence}\nResult: {test_result}")
    except Exception as e:
        print(f"Could not perform inference for the example evidence. Error: {e}")
else:
    print("\n--- Test Inference Skipped ---")
    print("Model 'PI4' not found or not loaded successfully.")
```

```
Warning: Not all demographic variables or target variable (PI1) found in the PI1 model.
Model contains: {'C2', 'PS3', 'Gender', 'E3', 'PE2', 'CT4', 'PV3', 'PPQ1', 'PI1', 'Employment Status', 'Shopping frequency'}
Warning: Not all demographic variables or target variable (PI2) found in the PI2 model.
Model contains: {'PPQ2', 'PE3', 'C2', 'PS3', 'Gender', 'E3', 'Level of Education', 'PV3', 'CT5', 'PI2', 'Shopping frequency'}
Warning: Not all demographic variables or target variable (PI3) found in the PI3 model.
Model contains: {'PS3', 'Gender', 'Level of Education', 'PE6', 'CT4', 'PV3', 'E4', 'PPQ4', 'C1', 'PI3', 'Shopping frequency'}
Warning: Not all demographic variables or target variable (PI4) found in the PI4 model.
Model contains: {'C2', 'PS3', 'Gender', 'Age', 'PE6', 'CT4', 'PV3', 'E2', 'PPQ4', 'PI4', 'Shopping frequency'}
Purchase intention states: {'PI1': [3, 4, 2, 5, 1], 'PI2': [4, 3, 1, 2, 5], 'PI3': [4, 3, 2, 1, 5], 'PI4': [4, 3, 2, 5, 1]}

--- Test Inference ---
Model: PI4
Evidence: {'Gender': 2, 'Age': 3}
Result: {4: np.float64(0.17535193345837993), 3: np.float64(0.18363730654936739), 2: np.float64(0.20022674038664884), 5: np.floa
```

```
# Create a list to store all our results
results_list = []

# Loop through each model (Unbalanced, SMOTE, etc.) in the loaded dictionary
if models is not None:
    for model_name, model_info in models.items():
        print(f"\n--- Processing Model: {model_name} ---")
        model = model_info.get('model')
        if model:
            # Determine the target variable for the current model (assuming model_name is like 'PI1', 'PI2', etc.)
            target_variable = model_name

            # Check if the target variable is valid
            if target_variable in intention_states_map:
                # Get the nodes (variables) present in the current model
                model_nodes = set(model.nodes())

                # Loop through each generated demographic combination
                for i, evidence in enumerate(all_demographic_combinations):
                    if (i + 1) % 100 == 0 or i == len(all_demographic_combinations) - 1:
                        print(f"  Processing combination {i + 1}/{len(all_demographic_combinations)} for model {model_name}..

                    # Filter the evidence to include only variables present in the current model
                    filtered_evidence = {
                        var: state for var, state in evidence.items() if var in model_nodes
                    }

                    # Ensure the target variable is not in the evidence (although unlikely with this setup)
                    if target_variable in filtered_evidence:
                        del filtered_evidence[target_variable]

                    # Ensure there is evidence to perform inference
                    if not filtered_evidence:
                         print(f"Skipping inference for combination {i+1} as no demographic evidence variables found in model
                         continue


                    try:
                        # Perform inference for the current combination and model
                        probabilities = perform_inference(model, target_variable, filtered_evidence)
                        # Add the results to our list
                        result_entry = {
                            'Model': model_name,
                            'Demographic Combination': evidence, # Store the original combination
                        }
                        # Add probabilities for each state of the target variable
                        if probabilities:
                            result_entry.update(probabilities)
                            results_list.append(result_entry)

                    except Exception as e:
                        print(f"Could not perform inference for evidence {filtered_evidence} in {model_name} model. Error: {e
            else:
                 print(f"Skipping model {model_name}: Target variable {target_variable} not found in intention_states_map.")
        else:
            print(f"Skipping model {model_name}: Model object not found.")
else:
    print("Models were not loaded. Skipping inference.")


# Convert the list of results into a pandas DataFrame
results_df = pd.DataFrame(results_list)

# Display the first few rows of our final results table
print("\n--- Analysis Complete ---")
display(results_df.head())

# Display info about the resulting DataFrame
print("\n--- Results DataFrame Info ---")
results_df.info()

# Display descriptive statistics
print("\n--- Results DataFrame Description ---")
display(results_df.describe())
```

```
--- Processing Model: PI1 ---
  Processing combination 100/4500 for model PI1...
  Processing combination 200/4500 for model PI1...
  Processing combination 300/4500 for model PI1...
  Processing combination 400/4500 for model PI1...
  Processing combination 500/4500 for model PI1...
  Processing combination 600/4500 for model PI1...
  Processing combination 700/4500 for model PI1...
  Processing combination 800/4500 for model PI1...
  Processing combination 900/4500 for model PI1...
  Processing combination 1000/4500 for model PI1...
  Processing combination 1100/4500 for model PI1...
  Processing combination 1200/4500 for model PI1...
  Processing combination 1300/4500 for model PI1...
  Processing combination 1400/4500 for model PI1...
  Processing combination 1500/4500 for model PI1...
  Processing combination 1600/4500 for model PI1...
  Processing combination 1700/4500 for model PI1...
  Processing combination 1800/4500 for model PI1...
  Processing combination 1900/4500 for model PI1...
  Processing combination 2000/4500 for model PI1...
  Processing combination 2100/4500 for model PI1...
  Processing combination 2200/4500 for model PI1...
  Processing combination 2300/4500 for model PI1...
  Processing combination 2400/4500 for model PI1...
  Processing combination 2500/4500 for model PI1...
  Processing combination 2600/4500 for model PI1...
  Processing combination 2700/4500 for model PI1...
  Processing combination 2800/4500 for model PI1...
  Processing combination 2900/4500 for model PI1...
  Processing combination 3000/4500 for model PI1...
  Processing combination 3100/4500 for model PI1...
  Processing combination 3200/4500 for model PI1...
  Processing combination 3300/4500 for model PI1...
  Processing combination 3400/4500 for model PI1...
  Processing combination 3500/4500 for model PI1...
  Processing combination 3600/4500 for model PI1...
  Processing combination 3700/4500 for model PI1...
  Processing combination 3800/4500 for model PI1...
  Processing combination 3900/4500 for model PI1...
  Processing combination 4000/4500 for model PI1...
  Processing combination 4100/4500 for model PI1...
  Processing combination 4200/4500 for model PI1...
  Processing combination 4300/4500 for model PI1...
  Processing combination 4400/4500 for model PI1...
  Processing combination 4500/4500 for model PI1...

--- Processing Model: PI2 ---
  Processing combination 100/4500 for model PI2...
  Processing combination 200/4500 for model PI2...
  Processing combination 300/4500 for model PI2...
  Processing combination 400/4500 for model PI2...
  Processing combination 500/4500 for model PI2...
  Processing combination 600/4500 for model PI2...
  Processing combination 700/4500 for model PI2...
  Processing combination 800/4500 for model PI2...
  Processing combination 900/4500 for model PI2...
  Processing combination 1000/4500 for model PI2...
  Processing combination 1100/4500 for model PI2...
  Processing combination 1200/4500 for model PI2...
  Processing combination 1300/4500 for model PI2...
  Processing combination 1400/4500 for model PI2...
  Processing combination 1500/4500 for model PI2...
  Processing combination 1600/4500 for model PI2...
  Processing combination 1700/4500 for model PI2...
  Processing combination 1800/4500 for model PI2...
  Processing combination 1900/4500 for model PI2...
  Processing combination 2000/4500 for model PI2...
  Processing combination 2100/4500 for model PI2...
  Processing combination 2200/4500 for model PI2...
  Processing combination 2300/4500 for model PI2...
  Processing combination 2400/4500 for model PI2...
  Processing combination 2500/4500 for model PI2...
  Processing combination 2600/4500 for model PI2...
  Processing combination 2700/4500 for model PI2...
  Processing combination 2800/4500 for model PI2...
  Processing combination 2900/4500 for model PI2...
  Processing combination 3000/4500 for model PI2...
  Processing combination 3100/4500 for model PI2...
  Processing combination 3200/4500 for model PI2...
  Processing combination 3300/4500 for model PI2...
  Processing combination 3400/4500 for model PI2...
  Processing combination 3500/4500 for model PI2...
  Processing combination 3600/4500 for model PI2...
  Processing combination 3700/4500 for model PI2...
  Processing combination 3800/4500 for model PI2...
  Processing combination 3900/4500 for model PI2...
  Processing combination 4000/4500 for model PI2...
  Processing combination 4100/4500 for model PI2...
```

```
    Processing combination 4200/4500 for model PI2...
    Processing combination 4300/4500 for model PI2...
    Processing combination 4400/4500 for model PI2...
    Processing combination 4500/4500 for model PI2...

  --- Processing Model: PI3 ---
    Processing combination 100/4500 for model PI3...
    Processing combination 200/4500 for model PI3...
    Processing combination 300/4500 for model PI3...
    Processing combination 400/4500 for model PI3...
    Processing combination 500/4500 for model PI3...
    Processing combination 600/4500 for model PI3...
    Processing combination 700/4500 for model PI3...
    Processing combination 800/4500 for model PI3...
    Processing combination 900/4500 for model PI3...
    Processing combination 1000/4500 for model PI3...
    Processing combination 1100/4500 for model PI3...
    Processing combination 1200/4500 for model PI3...
    Processing combination 1300/4500 for model PI3...
    Processing combination 1400/4500 for model PI3...
    Processing combination 1500/4500 for model PI3...
    Processing combination 1600/4500 for model PI3...
    Processing combination 1700/4500 for model PI3...
    Processing combination 1800/4500 for model PI3...
    Processing combination 1900/4500 for model PI3...
    Processing combination 2000/4500 for model PI3...
    Processing combination 2100/4500 for model PI3...
    Processing combination 2200/4500 for model PI3...
    Processing combination 2300/4500 for model PI3...
    Processing combination 2400/4500 for model PI3...
    Processing combination 2500/4500 for model PI3...
    Processing combination 2600/4500 for model PI3...
    Processing combination 2700/4500 for model PI3...
    Processing combination 2800/4500 for model PI3...
    Processing combination 2900/4500 for model PI3...
    Processing combination 3000/4500 for model PI3...
    Processing combination 3100/4500 for model PI3...
    Processing combination 3200/4500 for model PI3...
    Processing combination 3300/4500 for model PI3...
    Processing combination 3400/4500 for model PI3...
    Processing combination 3500/4500 for model PI3...
    Processing combination 3600/4500 for model PI3...
    Processing combination 3700/4500 for model PI3...
    Processing combination 3800/4500 for model PI3...
    Processing combination 3900/4500 for model PI3...
    Processing combination 4000/4500 for model PI3...
    Processing combination 4100/4500 for model PI3...
    Processing combination 4200/4500 for model PI3...
    Processing combination 4300/4500 for model PI3...
    Processing combination 4400/4500 for model PI3...
    Processing combination 4500/4500 for model PI3...

  --- Processing Model: PI4 ---
    Processing combination 100/4500 for model PI4...
    Processing combination 200/4500 for model PI4...
    Processing combination 300/4500 for model PI4...
    Processing combination 400/4500 for model PI4...
    Processing combination 500/4500 for model PI4...
    Processing combination 600/4500 for model PI4...
    Processing combination 700/4500 for model PI4...
    Processing combination 800/4500 for model PI4...
    Processing combination 900/4500 for model PI4...
    Processing combination 1000/4500 for model PI4...
    Processing combination 1100/4500 for model PI4...
    Processing combination 1200/4500 for model PI4...
    Processing combination 1300/4500 for model PI4...
    Processing combination 1400/4500 for model PI4...
    Processing combination 1500/4500 for model PI4...
    Processing combination 1600/4500 for model PI4...
    Processing combination 1700/4500 for model PI4...
    Processing combination 1800/4500 for model PI4...
    Processing combination 1900/4500 for model PI4...
    Processing combination 2000/4500 for model PI4...
    Processing combination 2100/4500 for model PI4...
    Processing combination 2200/4500 for model PI4...
    Processing combination 2300/4500 for model PI4...
    Processing combination 2400/4500 for model PI4...
    Processing combination 2500/4500 for model PI4...
    Processing combination 2600/4500 for model PI4...
    Processing combination 2700/4500 for model PI4...
    Processing combination 2800/4500 for model PI4...
    Processing combination 2900/4500 for model PI4...
    Processing combination 3000/4500 for model PI4...
    Processing combination 3100/4500 for model PI4...
    Processing combination 3200/4500 for model PI4...
    Processing combination 3300/4500 for model PI4...
    Processing combination 3400/4500 for model PI4...
    Processing combination 3500/4500 for model PI4...
    Processing combination 3600/4500 for model PI4...
    Processing combination 3700/4500 for model PI4...
    Processing combination 3800/4500 for model PI4
```

```
Processing combination 3800/4500 for model PI4...
Processing combination 3900/4500 for model PI4...
Processing combination 4000/4500 for model PI4...
Processing combination 4100/4500 for model PI4...
Processing combination 4200/4500 for model PI4...
Processing combination 4300/4500 for model PI4...
Processing combination 4400/4500 for model PI4...
Processing combination 4500/4500 for model PI4...

--- Analysis Complete ---
   Model          Demographic Combination        3        4        2        5        1
```

Part 3: Results and Interpretation

Now that we have our probabilities compiled in a DataFrame, we can visualize and interpret them. The best way to compare these results is with a **heatmap**.

A heatmap will allow us to instantly see which demographic profiles have the highest probability for each purchase intention. We will generate separate heatmaps for each model and each intention type to make our comparisons clear and insightful.

```python
def plot_heatmaps_for_model(df, model_name, intention_states_map):
    """
    Generates heatmaps for each purchase intention for a specific model.
    """
    # Filter the DataFrame for the specific model
    model_df = df[df['Model'] == model_name].copy() # Use .copy() to avoid SettingWithCopyWarning

    # Define a mapping from numerical states to Likert scale labels for Purchase Intentions
    intention_label_map = {
        1: 'Strongly Disagree',
        2: 'Disagree',
        3: 'Neutral',
        4: 'Agree',
        5: 'Strongly Agree'
    }

    # Extract simplified demographic profile string for indexing
    # This is a simplification as plotting all 4500 combinations is not practical
    # You might want to refine how you represent demographic combinations based on your analysis needs
    model_df['Demographic Profile'] = model_df['Demographic Combination'].apply(
        lambda x: ", ".join([f"{k}: {v}" for k, v in x.items()])
    )


    # Get the list of intention types (our columns for the heatmap) for this model
    # Sort the states numerically to ensure consistent plotting order (1, 2, 3, 4, 5)
    intention_types = sorted(intention_states_map.get(model_name, []))


    if not intention_types:
        print(f"No intention states found for model {model_name}. Skipping heatmap.")
        return

    # Create a subplot for each intention type
    fig, axes = plt.subplots(1, len(intention_types), figsize=(20, 8), sharey=True)
    fig.suptitle(f'Probability of Purchase Intention by Demographic Profile - {model_name} Model', fontsize=16)

    # Ensure axes is an array even if there's only one subplot
    if len(intention_types) == 1:
        axes = [axes]

    for i, intention_state in enumerate(intention_types):
        # Get the descriptive label for the current intention state
        intention_label = intention_label_map.get(intention_state, str(intention_state))

        # Pivot the data to create a matrix for the heatmap
        # Using 'Demographic Profile' as index and intention_state as value
        pivot_table = model_df.pivot_table(
            values=intention_state,
            index='Demographic Profile',
            aggfunc='mean' # Aggregating if there are duplicate profiles (unlikely with all combinations)
        )
        # Draw the heatmap
        sns.heatmap(pivot_table, ax=axes[i], annot=False, fmt=".2f", cmap="viridis") # annot=False for readability with many
        axes[i].set_title(f'{intention_label}') # Use the descriptive label here
        axes[i].set_ylabel('Demographic Profile')
        axes[i].tick_params(axis='x', rotation=0) # No rotation for single column
        axes[i].tick_params(axis='y', labelsize=6) # Adjust label size for y-axis


    plt.tight_layout(rect=[0, 0, 1, 0.96])
    plt.show()
    print(f'{model_name}')

# Generate the heatmaps for each model in our results
# Pass the intention_states_map to the plotting function
for model_name in results_df['Model'].unique():
    plot_heatmaps_for_model(results_df, model_name, intention_states_map)
```