DEVELOPING PREDICTIVE MODELS FOR SMOKING BEHAVIOUR: ANALYSING SMOKING-RELATED RISK FACTORS

INTRODUCTION

This project aims to explore the relationship between smoking habits and various demographic, socioeconomic, and behavioural factors through data analysis and predictive modelling. The study focuses on cleaning and preprocessing the dataset, identifying key features, and applying classification models to predict the likelihood of an individual smoking. By leveraging statistical methods, visualisations, and machine learning techniques, the project seeks to uncover significant trends, evaluate model performance, and provide insights that could aid in public health decision-making and targeted interventions. All the required libraries were loaded for this project to ensure a smooth workflow.

In [3]:
```
import numpy as np
import pandas as pd
```

1.0 DATA HANDLER

In [5]:
```
class DataHandler:
    """
    Class to handle data loading, cleaning, and pre-processing tasks.
    """

    def __init__(self, file_path):
        """
        Initialise the DataHandler with a file path.
        Parameters:
            file_path (str): Path to the CSV data file.
        """
        try:
            self.data = pd.read_csv(file_path)
            print("Data loaded successfully. Shape:", self.data.shape) # Print the shape of the dataset to confirm loading
        except FileNotFoundError:
            print(f"Error: File '{file_path}' not found.")

    def dataset_head(self, rows, columns=None):
        """
        Shows the dataset or specified columns' first few rows.

        Parameters:
            rows (int): Number of rows to display.
            columns (list, optional): List of column names to display. If None, display all columns.
        """
        data_to_display = self.data[columns] if columns else self.data
        print("First few rows of the dataset:")
        print(data_to_display.head(rows))

    def dataset_tail(self, rows, columns=None):
        """
        Shows the dataset or specified columns' last few rows.
```

```
        Parameters:
            rows (int): Number of rows to display.
            columns (list, optional): List of column names to display. If None, display all columns.
        """
        data_to_display = self.data[columns] if columns else self.data
        print("Last few rows of the dataset:")
        print(data_to_display.tail(rows))

    def check_missing_values(self, columns=None):
        """
        Check for missing values in the dataset.
        Returns:
            pandas.Series: Number of missing values per column.
        """
        # Use specified columns for checking duplicates, or the entire dataset if no columns are provided.
        data_to_check = self.data[columns] if columns else self.data  #
        missing_values = data_to_check.isnull().sum()
        if missing_values.any():
            print("Missing values detected:\n", missing_values[missing_values > 0])
        else:
            print("No missing values found.")
        return missing_values

    def check_duplicates(self, columns=None):
        """
        If specific columns are provided, use only those columns to check for duplicates;
             otherwise, check the entire dataset for duplicate rows.
        """
        data_to_check = self.data[columns] if columns else self.data
        duplicates = data_to_check.duplicated().sum()
        print(f"Number of duplicate rows: {duplicates}" if duplicates else "No duplicate rows found.")
        return duplicates

    def drop_columns(self, columns=None):
        """
        Drop specified columns from the dataset. If no columns are specified,
        drop columns without valid headers (empty or NaN).

        Parameters:
            columns (list or None): List of column names to drop. If None, drop columns without valid headers.
        """
        if columns:
            # Drop specified columns
            self.data.drop(columns=columns, inplace=True, errors='ignore')
            print(f"Dropped specified columns: {columns}")
        else:
            # Identify columns with empty or NaN headers
            columns_to_drop = [col for col in self.data.columns if pd.isnull(col) or str(col).strip() == '']
            if columns_to_drop:
                self.data.drop(columns=columns_to_drop, inplace=True)
                print(f"Dropped columns without valid headers: {columns_to_drop}")
            else:
                print("No columns with missing or empty headers found.")
```

```
    def set_non_smokers_to_zero(self):
        """
        The missing values arise from the 'smoke' column with 'No' responses and empty cells that follow
        in the 'amt_weekends', 'amt_weekdays', and 'type' cells.
        Set 'amt_weekends', 'amt_weekdays' to 0 and 'type' to 'Notype' for non-smokers.
        """
        if all(col in self.data.columns for col in ['smoke', 'amt_weekends', 'amt_weekdays', 'type']):
            self.data.loc[self.data['smoke'] == 'No', ['amt_weekends', 'amt_weekdays', 'type']] = [0, 0, 'Notype']
            print("Updated rows for non-smokers.")
        else:
            print("Some columns required for updating non-smokers are missing.")

    def convert_gross_income(self):
        """
        Convert the 'gross_income' column from range strings to numeric values.
        The income ranges were mapped to their mid-point values,
        and the unknown and refused values were mapped to NaN (not available numbers)
        """
        # Define mapping for each income range
        income_map = {
            "Under 2,600": 1300,
            "2,600 to 5,200": 3900,
            "5,200 to 10,400": 7800,
            "10,400 to 15,600": 13000,
            "15,600 to 20,800": 18200,
            "20,800 to 28,600": 24700,
            "28,600 to 36,400": 32500,
            "Above 36,400": 40000,
            "Refused": np.nan,
            "Unknown": np.nan
        }
        # Apply mapping to the 'gross_income' column
        self.data['gross_income_numeric'] = self.data['gross_income'].map(income_map)
        print("Gross income converted to numeric values.")

    def get_summary(self):
        """
        Print a summary of dataset information, missing values, and duplicates.
        """
        print("\n--- Dataset Information ---")
        self.data.info()
        print("\n--- Missing Values ---")
        self.check_missing_values()
        print("\n--- Duplicate Rows ---")
        self.check_duplicates()

    def get_data(self):
        return self.data

    def save_cleaned_data(self, output_file_path):
        """
        Save the cleaned dataset to a new CSV file.
        Parameters:
            output_file_path (str): Path where the cleaned dataset will be saved.
        """
        self.data.to_csv(output_file_path, index=False)
```

```
        print(f"Cleaned data saved to {output_file_path}")
```

In [6]:
```
# Main Execution
file_path = 'smoking.csv'
data_handler = DataHandler(file_path)
```

```
Data loaded successfully. Shape: (1691, 13)
```

In [7]:
```
#This will print the data frame's top ten rows for each column
data_handler.dataset_head(10)
```

```
First few rows of the dataset:
   Unnamed: 0  gender  age marital_status highest_qualification nationality  \
0           1    Male   38       Divorced      No Qualification     British
1           2  Female   42         Single      No Qualification     British
2           3    Male   40        Married                Degree     English
3           4  Female   40        Married                Degree     English
4           5  Female   39        Married          GCSE/O Level     British
5           6  Female   37        Married          GCSE/O Level     British
6           7    Male   53        Married                Degree     British
7           8    Male   44         Single                Degree     English
8           9    Male   40         Single              GCSE/CSE     English
9          10  Female   41        Married      No Qualification     English

  ethnicity      gross_income     region smoke  amt_weekends  amt_weekdays  \
0     White    2,600 to 5,200  The North    No           NaN           NaN
1     White       Under 2,600  The North   Yes          12.0          12.0
2     White  28,600 to 36,400  The North    No           NaN           NaN
3     White  10,400 to 15,600  The North    No           NaN           NaN
4     White    2,600 to 5,200  The North    No           NaN           NaN
5     White  15,600 to 20,800  The North    No           NaN           NaN
6     White      Above 36,400  The North   Yes           6.0           6.0
7     White  10,400 to 15,600  The North    No           NaN           NaN
8     White    2,600 to 5,200  The North   Yes           8.0           8.0
9     White   5,200 to 10,400  The North   Yes          15.0          12.0

          type
0          NaN
1      Packets
2          NaN
3          NaN
4          NaN
5          NaN
6      Packets
7          NaN
8  Hand-Rolled
9      Packets
```

In [8]:
```
# Data handling operations for checking missing values in the majority of the columns
data_handler.check_missing_values(columns=['gender',    'age',  'marital_status',       'highest_qualification',
                                           'nationality',       'ethnicity',    'gross_income', 'region',       'smoke',
                                           'amt_weekends', 'amt_weekdays',      'type'])
```

```
Missing values detected:
 amt_weekends    1270
amt_weekdays    1270
type            1270
dtype: int64
```

Out[8]:
```
gender                    0
age                       0
marital_status            0
highest_qualification     0
nationality               0
ethnicity                 0
gross_income              0
region                    0
smoke                     0
amt_weekends           1270
amt_weekdays           1270
type                   1270
dtype: int64
```

The information reveals 1,270 missing values across columns including `amt_weekends`, `amt_weekdays`, and `type`.

In [10]:
```
# Check for duplicates in the dataset
data_handler.check_duplicates()
```

```
No duplicate rows found.
```

Out[10]:
```
0
```

In [11]:
```
data_handler.set_non_smokers_to_zero()
```

```
Updated rows for non-smokers.
```

Cross-checking the correlations between variables that had missing values, it became evident that the missing values resulted from columns where 'NO' was entered in the 'smoke' column, and no information was provided for the 'amt_weekends',mns. A function was defined to set these parameters to 0, 0, and NoType in order 'amt_weekdays', and 'type' colu to populate the empty columns.

In [13]:
```
data_handler.get_summary()
```

```
--- Dataset Information ---
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1691 entries, 0 to 1690
Data columns (total 13 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   Unnamed: 0            1691 non-null   int64  
 1   gender                1691 non-null   object 
 2   age                   1691 non-null   int64  
 3   marital_status        1691 non-null   object 
 4   highest_qualification  1691 non-null   object 
 5   nationality           1691 non-null   object 
 6   ethnicity             1691 non-null   object 
 7   gross_income          1691 non-null   object 
 8   region                1691 non-null   object 
 9   smoke                 1691 non-null   object 
 10  amt_weekends          1691 non-null   float64
 11  amt_weekdays          1691 non-null   float64
 12  type                  1691 non-null   object 
dtypes: float64(2), int64(2), object(9)
memory usage: 171.9+ KB

--- Missing Values ---
No missing values found.

--- Duplicate Rows ---
No duplicate rows found.
```

The summary shows there are no missing values and duplicate rows after the dataset has been cleaned.

In [15]:
```
# Drop unnecessary columns
data_handler.drop_columns(columns=['Unnamed: 0'])
```

```
Dropped specified columns: ['Unnamed: 0']
```

The 'Unnamed: 0' column, which served as an ID ranging from 1 to 1691, was removed from the dataset to prevent bias and because it holds no relevance to the analysis process.

In [17]:
```
data_handler.convert_gross_income()  # Creates a new column and converts gross income to numeric values in the new column
```

```
Gross income converted to numeric values.
```

In [18]:
```
# Print the first 6 cells in the 'gross_income' and 'gross_income_numeric' columns to verify the conversion
data_handler.dataset_head(6, columns=['gross_income', 'gross_income_numeric'])
```

```
First few rows of the dataset:
       gross_income  gross_income_numeric
0    2,600 to 5,200                3900.0
1       Under 2,600                1300.0
2  28,600 to 36,400               32500.0
3  10,400 to 15,600               13000.0
4    2,600 to 5,200                3900.0
5  15,600 to 20,800               18200.0
```

In [19]:
```
# Export the processed data to a new file.
data_handler.save_cleaned_data('cleaned_smoking.csv')
```

```
Cleaned data saved to cleaned_smoking.csv
```

The cleaned data was saved into a new file, and subsequent analyses were carried out on this file.

In [ ]:

```

```

2.0 NUMERICAL-STATISTICS

The statistical measures of the numerical columns will be calculated in the section

In [23]:

```
class NumericalStatistics:
    """
    Class to calculate and display individual statistical measures for numeric columns in the dataset.
    """

    def __init__(self, data):
        """
        Initialises the class with the given dataset, keeping only numeric columns.
        Parameters:
            data (DataFrame): The dataset containing numeric and non-numeric columns.
        """
        self.data = data.select_dtypes(include='number')  # Keep only numeric columns
        if self.data.empty:
            print("No numeric columns found in the dataset.")
        else:
            print(f"Numeric columns detected: {self.data.columns.tolist()}")

    def count(self, columns=None):
        """
        Calculate the count of non-missing values for numeric columns.
        Parameters:
            columns (list, optional): List of specific columns to calculate the count for.
            If None, counts for all numeric columns.
        Returns:
            pandas.Series: Counts of non-missing values for the specified columns.
        """
        data_to_calculate = self.data[columns] if columns else self.data
        counts = data_to_calculate.notnull().sum()  # Count non-NaN values for each column
        print("Count for column:")
        print(counts)
        # return counts

    def mean(self, columns=None, print_output=True):
        """
        Calculate the mean (average) for numeric column.
        Parameters:
            columns (list, optional): List of specific columns to calculate the mean for.
            If None, calculate for all numeric columns.
            print_output (bool): Whether to print the result. Default is True.
        Returns:
            pandas.Series: Means of the specified numeric columns.
        """
        data_to_calculate = self.data[columns] if columns else self.data
        means = data_to_calculate.mean()
        if print_output:
```

```
            print("Mean for column:")
            print(means)
        # return means

    def variance(self, columns=None, print_output=True):
        """
        Calculate the variance for numeric columns.
        Parameters:
            columns (list, optional): List of specific columns to calculate the variance for.
            If None, calculate for all numeric columns.
            print_output (bool): Whether to print the result. Default is True.
        Returns:
            pandas.Series: Variances of the specified numeric columns.
        """
        data_to_calculate = self.data[columns] if columns else self.data
        variances = data_to_calculate.var()
        if print_output:
            print("Variance for column:")
            print(variances)
        # return variances

    def std_dev(self, columns=None, print_output=True):
        """
        Calculate the standard deviation for numeric columns.
        Parameters:
            columns (list, optional): List of specific columns to calculate the standard deviation for.
            If None, calculate for all numeric columns.
            print_output (bool): Whether to print the result. Default is True.
        Returns:
            pandas.Series: Standard deviations of the specified numeric columns.
        """
        data_to_calculate = self.data[columns] if columns else self.data
        std_devs = data_to_calculate.std()
        if print_output:
            print("Standard Deviation for column:")
            print(std_devs)
        # return std_devs

    def minimum(self, columns=None):
        """
        Calculate the minimum value for numeric columns.
        Parameters:
            columns (list, optional): List of specific columns to calculate the minimum value for.
            If None, calculate for all numeric columns.
        Returns:
            pandas.Series: Minimum values for the specified numeric columns.
        """
        data_to_calculate = self.data[columns] if columns else self.data
        mins = data_to_calculate.min()
        print("Minimum value for column:")
        print(mins)
        # return mins

    def quantile(self, percentile, columns=None):
        """
        Calculate a specific quantile (percentile) for numeric columns.
```

```
        Parameters:
            percentile (float): The desired percentile (between 0 and 1).
            columns (list, optional): List of specific columns to calculate the quantile for.
            If None, calculate for all numeric columns.
        Returns:
            pandas.Series: Quantile values for the specified numeric columns.
        """
        if not (0 <= percentile <= 1):
            print("Error: Percentile must be between 0 and 1.")
            return None
        data_to_calculate = self.data[columns] if columns else self.data
        quantiles = data_to_calculate.quantile(percentile)
        print(f"{percentile * 100}% Quantiles for specified columns:")
        print(quantiles)
        # return quantiles

    def maximum(self, columns=None):
        """
        Calculate the maximum value for numeric columns.
        Parameters:
            columns (list, optional): List of specific columns to calculate the maximum value for.
            If None, calculate for all numeric columns.
        Returns:
            pandas.Series: Maximum values for the specified numeric columns.
        """
        data_to_calculate = self.data[columns] if columns else self.data
        maxs = data_to_calculate.max()
        print("Maximum value columns:")
        print(maxs)
        # return maxs
```

Numerical statistics were carried out on the cleaned dataset to gain further insights into the properties of each numerical column. These statistical functions are versatile, enabling computation for numeric values across the entire dataset or within specific columns as required.

In [25]:
```
#Load the preprocessed dataset
cleaned_data = pd.read_csv('cleaned_smoking.csv')

# Initialize the NumericalStatistics class
statistics = NumericalStatistics(cleaned_data)
```

```
Numeric columns detected: ['age', 'amt_weekends', 'amt_weekdays', 'gross_income_numeric']
```

In [26]:
```
statistics.count()          # Displays the total count of values in each numerical column.
```

```
Count for column:
age                     1691
amt_weekends            1691
amt_weekdays            1691
gross_income_numeric    1565
dtype: int64
```

In [27]:
```
# Get mean, variance, and standard deviation for specific numeric columns
columns_to_check = ['age']
statistics.mean(columns=columns_to_check)
statistics.variance(columns=columns_to_check)
statistics.std_dev(columns=columns_to_check)
```

```
Mean for column:
age    49.836192
dtype: float64
Variance for column:
age    351.069601
dtype: float64
Standard Deviation for column:
age    18.736851
dtype: float64
```

The mean, variance and standard deviation of the age column were calculated and displayed in a float64 type

In [29]:
```
# Get mean, variance, and standard deviation for all numeric columns
statistics.mean()
statistics.variance()
statistics.std_dev()
```

```
Mean for column:
age                      49.836192
amt_weekends              4.085748
amt_weekdays              3.423418
gross_income_numeric    13498.785942
dtype: float64
Variance for column:
age                     3.510696e+02
amt_weekends            7.471039e+01
amt_weekdays            5.727978e+01
gross_income_numeric    1.093537e+08
dtype: float64
Standard Deviation for column:
age                      18.736851
amt_weekends              8.643517
amt_weekdays              7.568341
gross_income_numeric    10457.233809
dtype: float64
```

In [30]:
```
# Get minimum, quantile and maximum for all numeric column
statistics.minimum()
statistics.quantile(0.25)  # 25% quantile
statistics.quantile(0.50)  # Median (50% quantile)
statistics.quantile(0.70)  # 70% quantile
statistics.maximum()
```

```
Minimum value for column:
age                       16.0
amt_weekends               0.0
amt_weekdays               0.0
gross_income_numeric    1300.0
dtype: float64
25.0% Quantiles for specified columns:
age                       34.0
amt_weekends               0.0
amt_weekdays               0.0
gross_income_numeric    7800.0
Name: 0.25, dtype: float64
50.0% Quantiles for specified columns:
age                       48.0
amt_weekends               0.0
amt_weekdays               0.0
gross_income_numeric    7800.0
Name: 0.5, dtype: float64
70.0% Quantiles for specified columns:
age                        62.0
amt_weekends                0.0
amt_weekdays                0.0
gross_income_numeric    18200.0
Name: 0.7, dtype: float64
Maximum value columns:
age                        97.0
amt_weekends               60.0
amt_weekdays               55.0
gross_income_numeric    40000.0
dtype: float64
```

In [31]:

```
# Get count and quantile for age and, minimum and maximum specifically for amt_weekends.
statistics.count(columns=['age'])
statistics.quantile(0.50, columns=['age'])
statistics.minimum(columns=['amt_weekends'])
statistics.maximum(columns=['amt_weekends'])
```

```
Count for column:
age    1691
dtype: int64
50.0% Quantiles for specified columns:
age    48.0
Name: 0.5, dtype: float64
Minimum value for column:
amt_weekends    0.0
dtype: float64
Maximum value columns:
amt_weekends    60.0
dtype: float64
```

In [ ]:

3.0 VISUALISER

In [33]:

```
from pandas.plotting import lag_plot
```

```python
import matplotlib.pyplot as plt
import seaborn as sns
```

In [34]:
```python
# Visualiser Section
class Visualiser:
    """
    Class to handle visualisation for the dataset.
    """

    def __init__(self, data):
        self.data = data
        sns.set(style="whitegrid")


    def plot_pie_distribution(self, column, exclude_values=None):
        """
        Plot the distribution of a given column as a pie chart.
        This method creates a pie chart to visualize the proportion of each category in the specified column.
        It checks for the presence of the column before plotting.
        Parameters:
            column (str): The column to plot.
            exclude_values (list, optional): Values to exclude from the column before plotting.
        """
        if column not in self.data.columns:
            print(f"Error: Column '{column}' not found in the dataset.")
            return
        # Exclude values if specified
        data_to_plot = self.data[~self.data[column].isin(exclude_values)] if exclude_values else self.data
        # Calculate value counts
        value_counts = data_to_plot[column].value_counts()
        # Plot the pie chart
        plt.figure(figsize=(8, 8))
        plt.pie(value_counts, labels=value_counts.index, autopct='%1.1f%%', startangle=140, colors=sns.color_palette("Set3"))
        plt.title(f"Distribution of {column} (Excluding {exclude_values})" if exclude_values else f"Distribution of {column}")
        plt.show()


    def plot_histogram_distribution(self, column_name, bins):
        """
        Plot distribution of a given column with optional bin customization.
        Parameters:
            column_name (str): The column name for which the distribution will be plotted.
            bins (int): Number of bins for the histogram.
        """
        # Check if the provided column exists in the dataset
        if column_name not in self.data.columns:
            print(f"Error: '{column_name}' column not found in the dataset.")
            return
        # Plot histogram distribution for the specified column
        plt.figure(figsize=(10, 6))
        sns.histplot(self.data[column_name], bins=bins, kde=True, color="skyblue")
        plt.title(f"{column_name} Distribution")
        plt.xlabel(column_name)
        plt.ylabel("Frequency")
        plt.grid(axis='y')
        plt.show()
```

```
    def plot_column_by_status(self, status_column, value_column):
        """
        Plot the distribution of a specified value column grouped by a status column using a boxplot.
        Parameters:
            status_column (str): The column name representing the status to group by (e.g., 'smoke').
            value_column (str): The column name representing the value to be plotted (e.g., 'age').
        """
        # Check if the specified columns exist in the dataset
        if status_column not in self.data.columns or value_column not in self.data.columns:
            print(f"Error: '{status_column}' or '{value_column}' column not found in the dataset.")
            return
        plt.figure(figsize=(10, 6))
        # Create a boxplot for the specified columns
        sns.boxplot(x=status_column, y=value_column, hue=status_column, data=self.data, palette="pastel", dodge=False, legend=False)
        plt.title(f"{value_column.capitalize()} Distribution by {status_column.capitalize()}")
        plt.xlabel(f"{status_column.capitalize()}")
        plt.ylabel(f"{value_column.capitalize()}")
        plt.show()


    def plot_countplot(self, x_column, hue_column, palette="coolwarm", title="", xlabel="", ylabel=""):
        """
        Create a countplot for columns.
        Parameters:
            x_column (str): The column to plot on the x-axis.
            hue_column (str): The column for color grouping.
            palette (str): The color palette for the plot (default is "coolwarm").
        """
        if x_column not in self.data.columns or hue_column not in self.data.columns:
            print(f"Error: '{x_column}' or '{hue_column}' column not found in the dataset.")
            return
        plt.figure(figsize=(12, 8))
        sns.countplot(x=x_column, hue=hue_column, data=self.data, palette=palette)
        plt.title(title)
        plt.xlabel(xlabel)
        plt.ylabel(ylabel)
        plt.xticks(rotation=45)
        plt.legend(title=hue_column)
        plt.grid(axis='y')
        plt.show()


    def plot_scatter(self, x_col, y_col, hue_col=None, title="Scatterplot", xlabel="X", ylabel="Y", palette="muted", filter_smokers=False):
        """
        Scatterplot for any two columns with an optional hue column, with an option to filter data for smokers.
        Parameters:
            x_col (str): The column name for the x-axis.
            y_col (str): The column name for the y-axis.
            hue_col (str): The column name for grouping the data by color (optional).
            title (str): The title of the plot (optional).
            xlabel (str): Label for the x-axis (optional).
            ylabel (str): Label for the y-axis (optional).
            palette (str): Color palette for the plot (optional).
            filter_smokers (bool): If True, filter the data to include only smokers (optional).
```

```
        """
        if x_col not in self.data.columns or y_col not in self.data.columns:
            print(f"Error: '{x_col}' or '{y_col}' column not found in the dataset.")
            return
        # Filter the data to include only smokers if filter_smokers is True
        data_to_plot = self.data
        if filter_smokers:
            data_to_plot = self.data[self.data['smoke'] == 'Yes']
        plt.figure(figsize=(8, 6))
        if hue_col and hue_col in data_to_plot.columns:
            sns.scatterplot(x=x_col, y=y_col, data=data_to_plot, hue=hue_col, palette=palette, alpha=0.6)
        else:
            sns.scatterplot(x=x_col, y=y_col, data=data_to_plot, palette=palette, alpha=0.6)
        plt.title(title)
        plt.xlabel(xlabel)
        plt.ylabel(ylabel)
        plt.show()


    def plot_density(self, column1, column2, filter_column=None, filter_value=None, xlim=None):
        """
        Plot a KDE (density) plot for two specified columns.
        Parameters:
        - column1 (str): The name of the first column for the KDE plot.
        - column2 (str): The name of the second column for the KDE plot.
        - filter_column (str, optional): The column used to filter the data. Defaults to None.
        - filter_value (str, optional): The value to filter the filter_column by. Defaults to None.
        - xlim (tuple, optional): The x-axis limits for the plot. Defaults to None.
        """
        if column1 not in self.data.columns or column2 not in self.data.columns:
            print(f"Error: '{column1}' or '{column2}' column not found in the dataset.")
            return
        # Filter data if a filter_column and filter_value are provided
        filtered_data = self.data
        if filter_column and filter_value is not None:
            if filter_column not in self.data.columns:
                print(f"Error: '{filter_column}' column not found in the dataset.")
                return
            filtered_data = self.data[self.data[filter_column] == filter_value]
        plt.figure(figsize=(12, 6))
        # KDE plot for the first column
        sns.kdeplot(filtered_data[column1], label=column1, color='blue', fill=True)
        # KDE plot for the second column
        sns.kdeplot(filtered_data[column2], label=column2, color='red', fill=True)
        # Set x-axis limits if provided
        if xlim:
            plt.xlim(xlim)
        # Set the plot title and labels
        plt.title(f"Density Plot: {column1} vs. {column2}")
        plt.xlabel("Values")
        plt.ylabel("Density")
        plt.legend()
        plt.show()


    def plot_2d_kde(self, x_col, y_col, filter_col=None, filter_value=None):
```

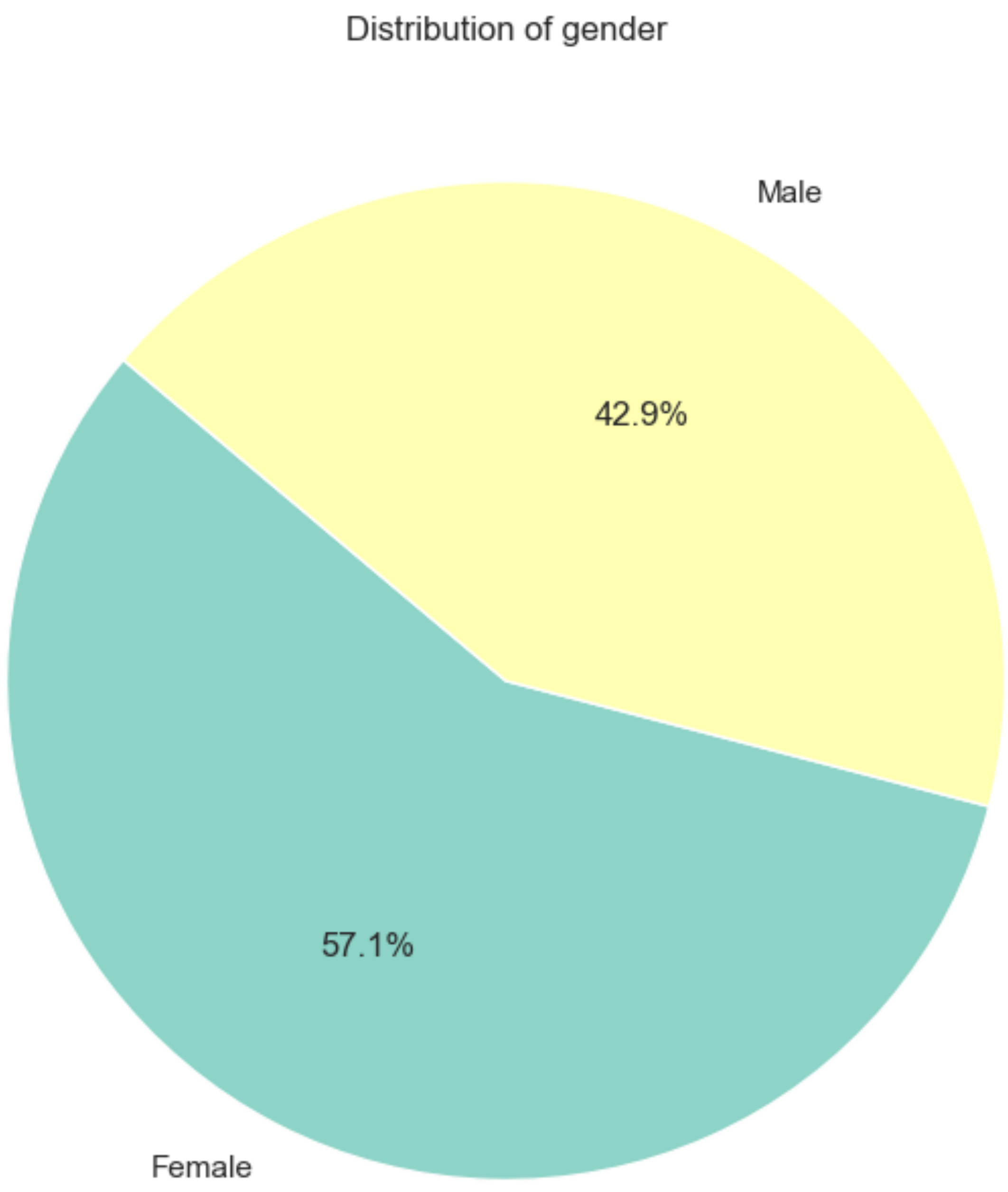
```
        """
        Plot a 2D KDE plot for the relationship between two columns, optionally filtering data by a specific column and value.
        Parameters:
        x_col (str): Specifies the x-axis column, and y_col (str): Specifies the y-axis column.
        filter_col (str, optional): The column to filter the data. Defaults to None.
        filter_value (str, optional): The value to filter the filter_col by. Defaults to None.
        """
        # Check if the specified columns exist in the dataset
        for col in [x_col, y_col, filter_col]:
            if col and col not in self.data.columns:
                print(f"Error: '{col}' column not found in the dataset.")
                return
        # Apply filtering if specified
        filtered_data = self.data
        if filter_col and filter_value is not None:
            filtered_data = filtered_data[filtered_data[filter_col] == filter_value]
        # Plot 2D KDE
        plt.figure(figsize=(12, 8))
        sns.kdeplot(x=filtered_data[x_col], y=filtered_data[y_col], cmap="Blues", fill=True)
        plt.title(f"2D KDE Plot: {x_col} vs. {y_col}" + (f" (Filtered by {filter_col} = {filter_value})" if filter_col else ""))
        plt.xlabel(x_col)
        plt.ylabel(y_col)
        plt.show()


    def plot_lag(self, column_name, lag=1):
        """
        Plot a lag plot for a column.
        Parameters:
            column_name (str): The name of the column to create the lag plot for.
            lag (int): The lag value for the plot (default is set to 1).
        """
        if column_name not in self.data.columns:
            print(f"Error: '{column_name}' column not found.")
            return
        plt.figure(figsize=(8, 8))
        lag_plot(self.data[column_name], lag=lag)
        plt.title(f"Lag Plot of {column_name} (Lag={lag})")
        plt.xlabel(f"{column_name} (Current)")
        plt.ylabel(f"{column_name} (Lagged)")
        plt.show()


    def plot_line_plot_trend(self, age_column, trend_columns, filter_column=None, filter_value=None, title="Line Plot"):
        """
        Plot the trend of specified columns over age using a line plot, with optional filtering.
        Parameters:
        - age_column (str): The name of the column representing age.
        - trend_columns (list): A list of column names to plot trends for.
        - filter_column (str, optional): The column name for filtering (e.g., 'smoke').
        - filter_value (str, optional): The value to filter the filter_column by (e.g., 'Yes').
        - title (str, optional): The title of the plot.
        """
        # Check if required columns are in the dataset
        missing_columns = [col for col in [age_column] + trend_columns + ([filter_column] if filter_column else []) if col not in self.data.columns]
        if missing_columns:
```

```
            print(f"Error: Columns {missing_columns} not found in the dataset.")
            return
        # Filter the data if filtering is specified
        data_to_plot = self.data
        if filter_column and filter_value is not None:
            data_to_plot = self.data[self.data[filter_column] == filter_value]
        # Check if there is any data after filtering
        if data_to_plot.empty:
            print("No data found after applying the filter.")
            return
        # Plot the trends
        plt.figure(figsize=(12, 6))
        for column in trend_columns:
            plt.plot(data_to_plot[age_column], data_to_plot[column], label=column)
        plt.title(title)
        plt.xlabel(age_column.capitalize())
        plt.ylabel("Value")
        plt.legend()
        plt.grid(True)
        plt.show()


    def plot_correlation_heatmap_for_columns(self, columns=None):
        """
        Plot a correlation heatmap for the specified columns or all numeric columns if none are specified.
        Parameters:
            columns (list or None): List of column names to include in the heatmap.
                                    If None, all numeric columns are used.
        """
        # Filter the dataset for the specified columns or select all numeric columns
        if columns:
            data_for_corr = self.data[columns].select_dtypes(include=np.number)
        else:
            data_for_corr = self.data.select_dtypes(include=np.number)

        if data_for_corr.empty:
            print("Error: No numeric columns available for correlation.")
            return
        # Compute the correlation matrix
        corr_matrix = data_for_corr.corr()
        # Plot the heatmap
        plt.figure(figsize=(10, 8))
        sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt=".2f")
        plt.title("Correlation Heatmap")
        plt.show()


    def plot_correlation_with_target(self, target_column):
        """
        Plot a correlation heatmap focusing on relationships with the specified target column.
        Parameters:
            target_column (str): The name of the column to calculate correlations with.
        """
        # Ensure the target column is in the dataset
        if target_column not in self.data.columns:
            print(f"Error: '{target_column}' column not found in the dataset.")
```

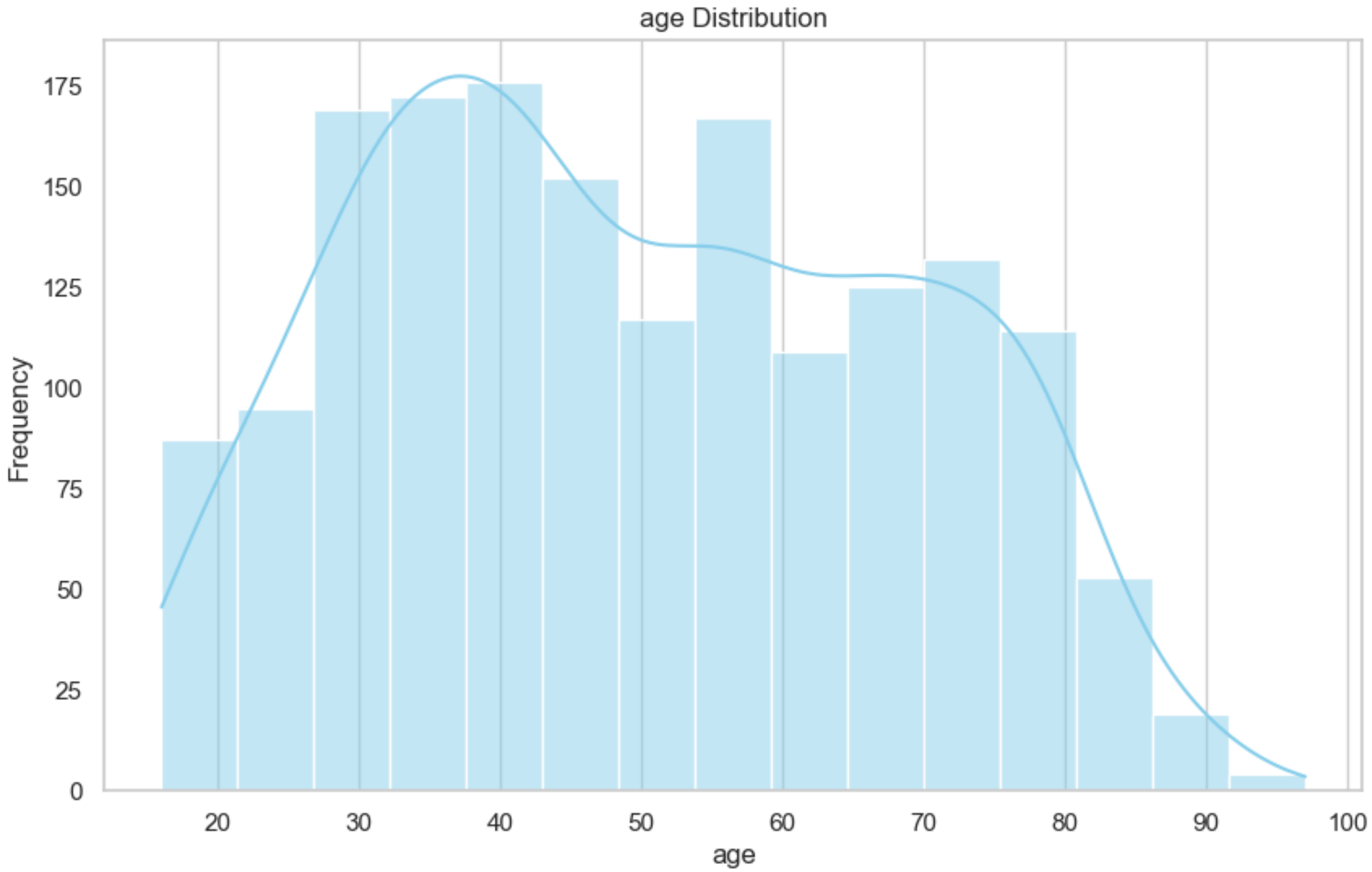
```
            return

        # Create a copy of the data to avoid modifying the original
        data_for_corr = self.data.copy()

        # Convert the target column to numeric if it's not already
        if data_for_corr[target_column].dtype == 'object':
            unique_values = data_for_corr[target_column].unique()
            if len(unique_values) == 2:  # Assuming binary values for correlation
                mapping = {unique_values[0]: 0, unique_values[1]: 1}
                data_for_corr[target_column] = data_for_corr[target_column].map(mapping)
            else:
                print(f"Error: '{target_column}' contains non-numeric and non-binary values.")
                return
        # Check if there are numeric columns
        if data_for_corr.select_dtypes(include=np.number).empty:
            print("Error: No numeric columns found in the dataset for correlation.")
            return
        # Calculate the correlation matrix
        corr_matrix = data_for_corr.select_dtypes(include=np.number).corr()
        # Plot only the correlations with the target column
        plt.figure(figsize=(12, 10))
        sns.heatmap(
            corr_matrix[[target_column]].sort_values(by=target_column, ascending=False),
            annot=True, cmap='coolwarm', fmt=".2f", cbar=True
        )
        plt.title(f"Correlation of Features with '{target_column}'")
        plt.xlabel(f"Correlation with '{target_column}'")
        plt.show()
```

In [35]:
```
# Visualisations
visualiser = Visualiser(cleaned_data)
```

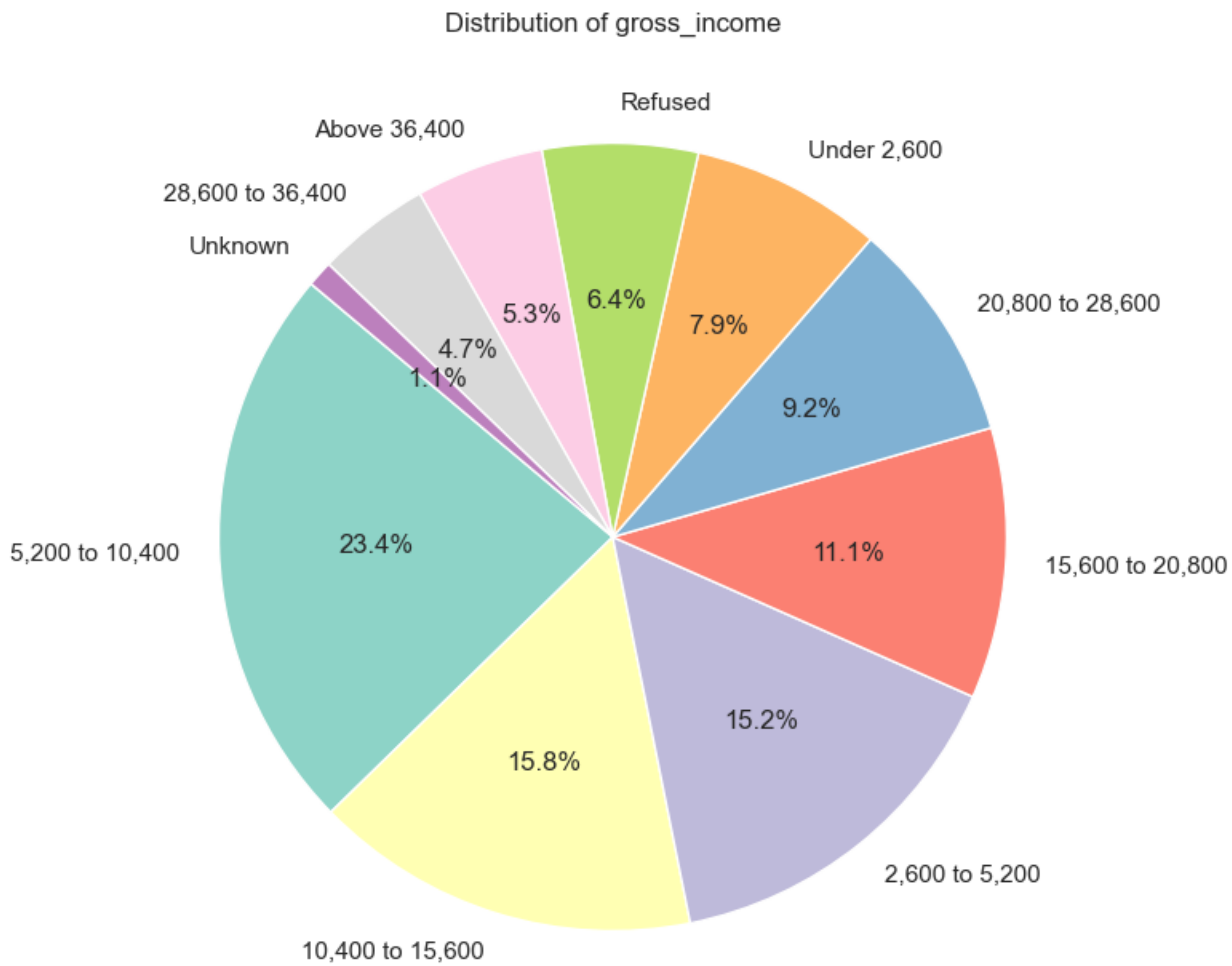
In [36]:
```
# For gender distribution
visualiser.plot_pie_distribution('gender')
```



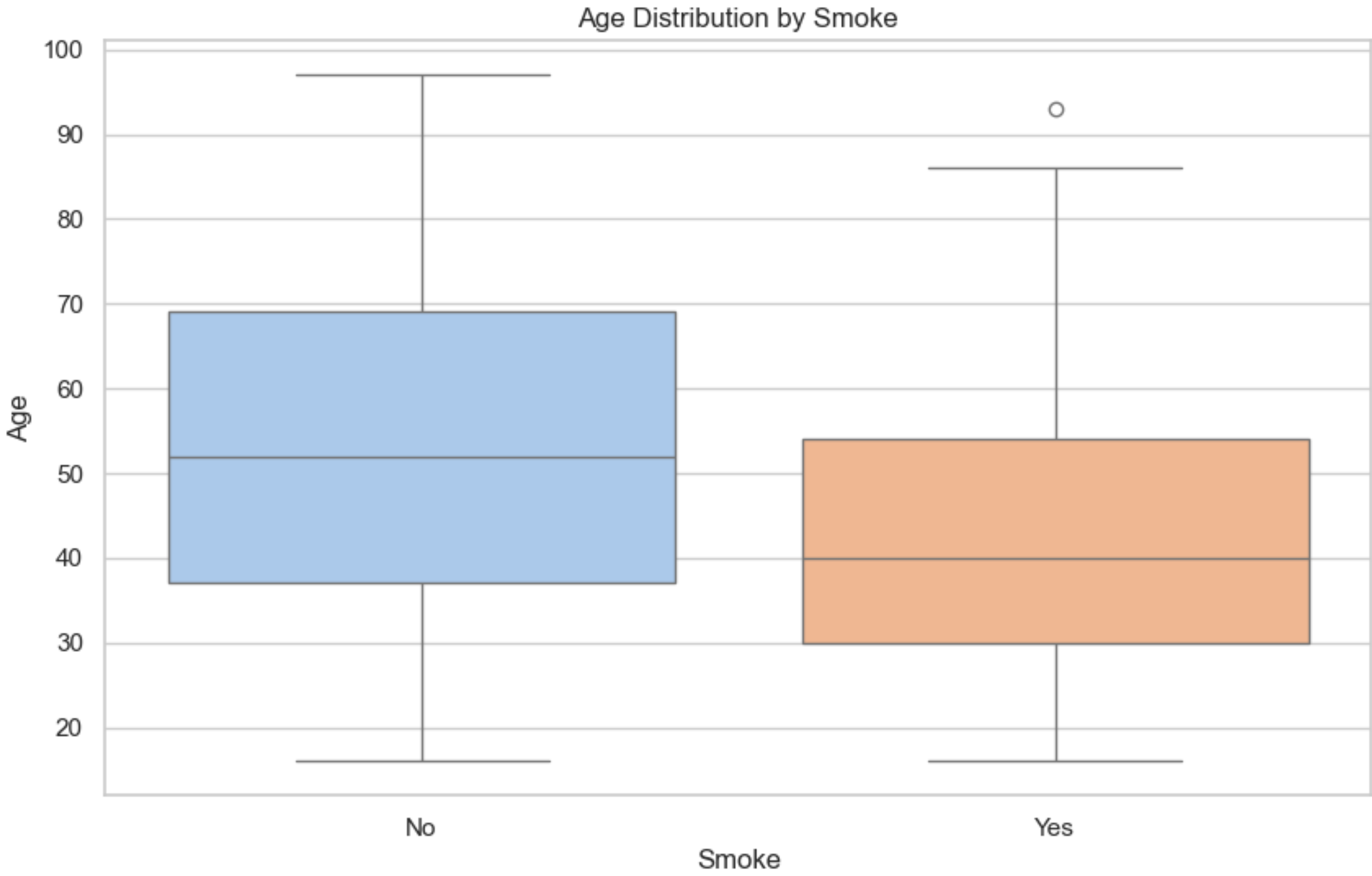
In [37]:
```
#Plot age distribution with 15 bins
visualiser.plot_histogram_distribution('age', bins=15)
```



In [38]:
```
# Plot gross income categories as a pie chart
visualiser.plot_pie_distribution('gross_income')
```

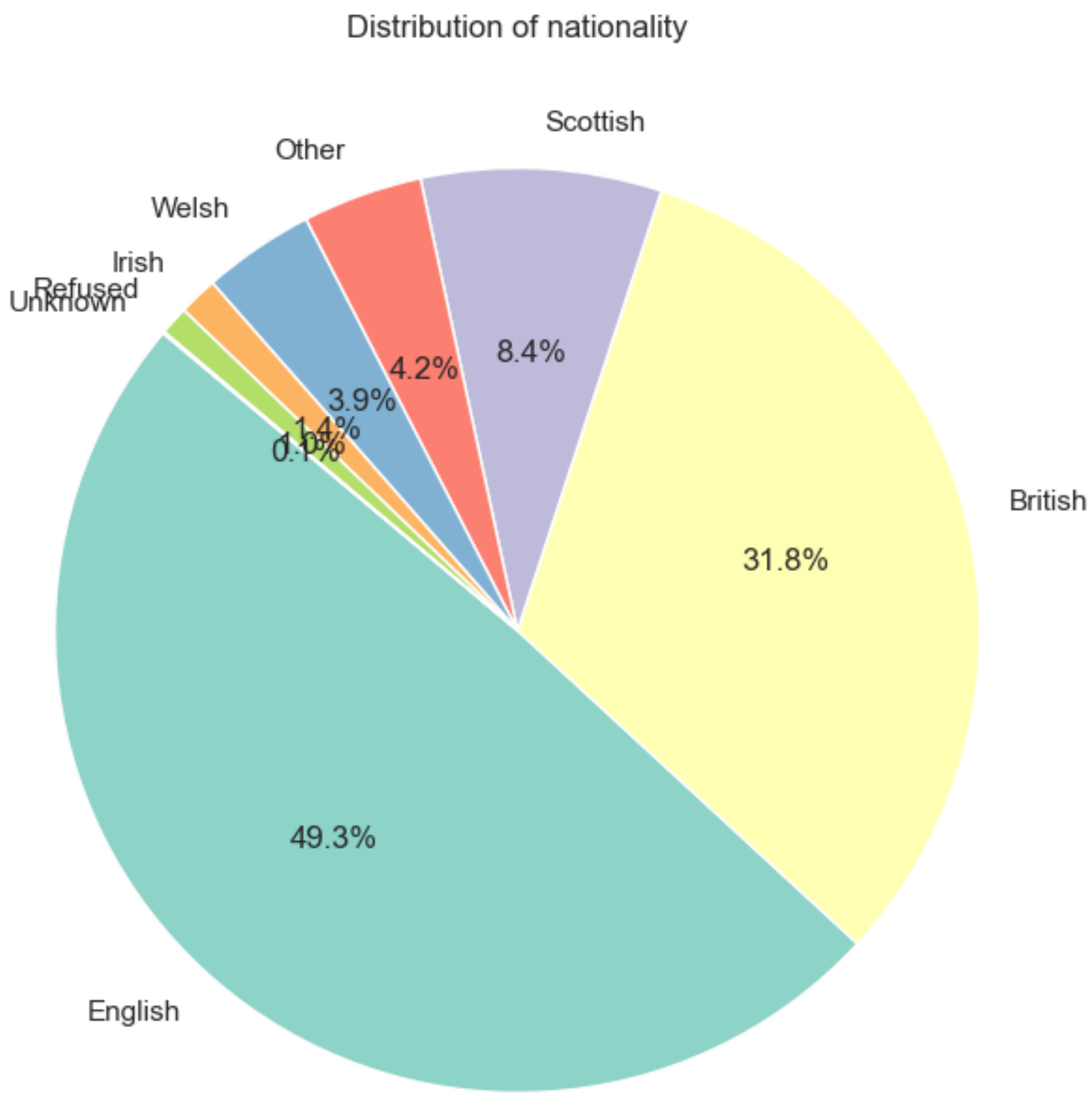


In [39]:
```
# To plot age by smoking status
visualiser.plot_column_by_status('smoke', 'age')
```

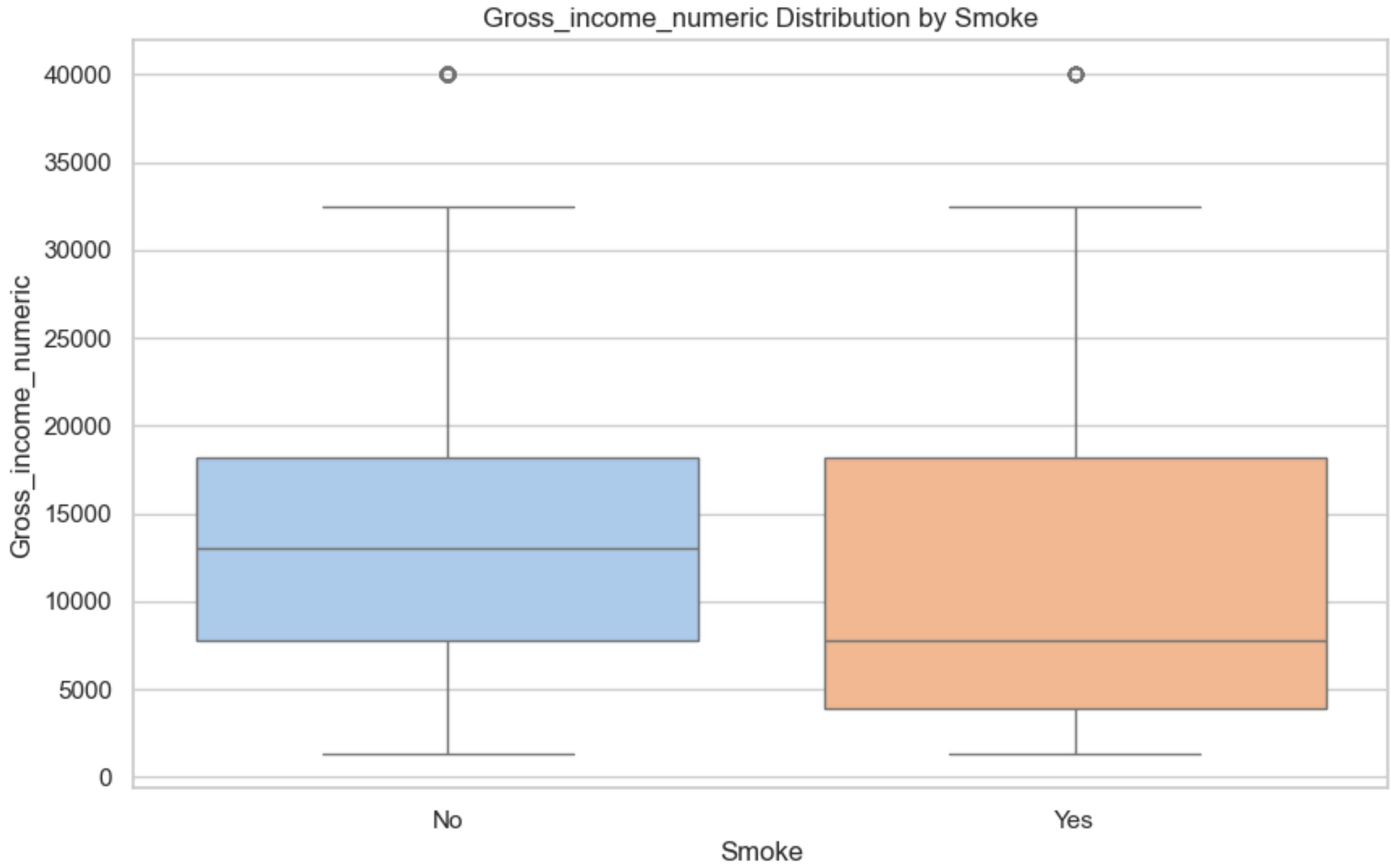


In [40]:

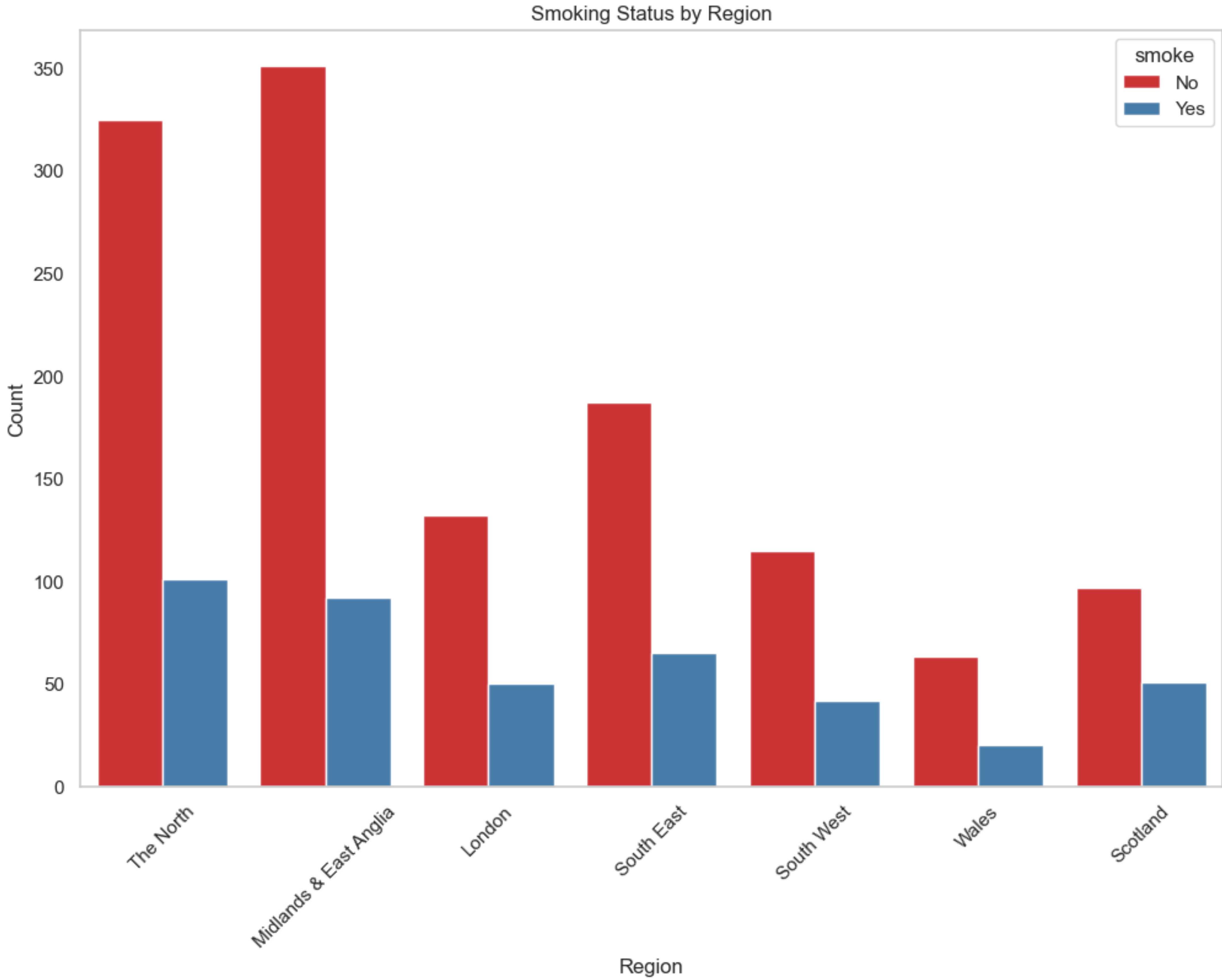
```
# For nationality distribution
visualiser.plot_pie_distribution('nationality')
```



In [41]:
```
# To plot gross_income_numeric by smoking status
visualiser.plot_column_by_status('smoke', 'gross_income_numeric')
```

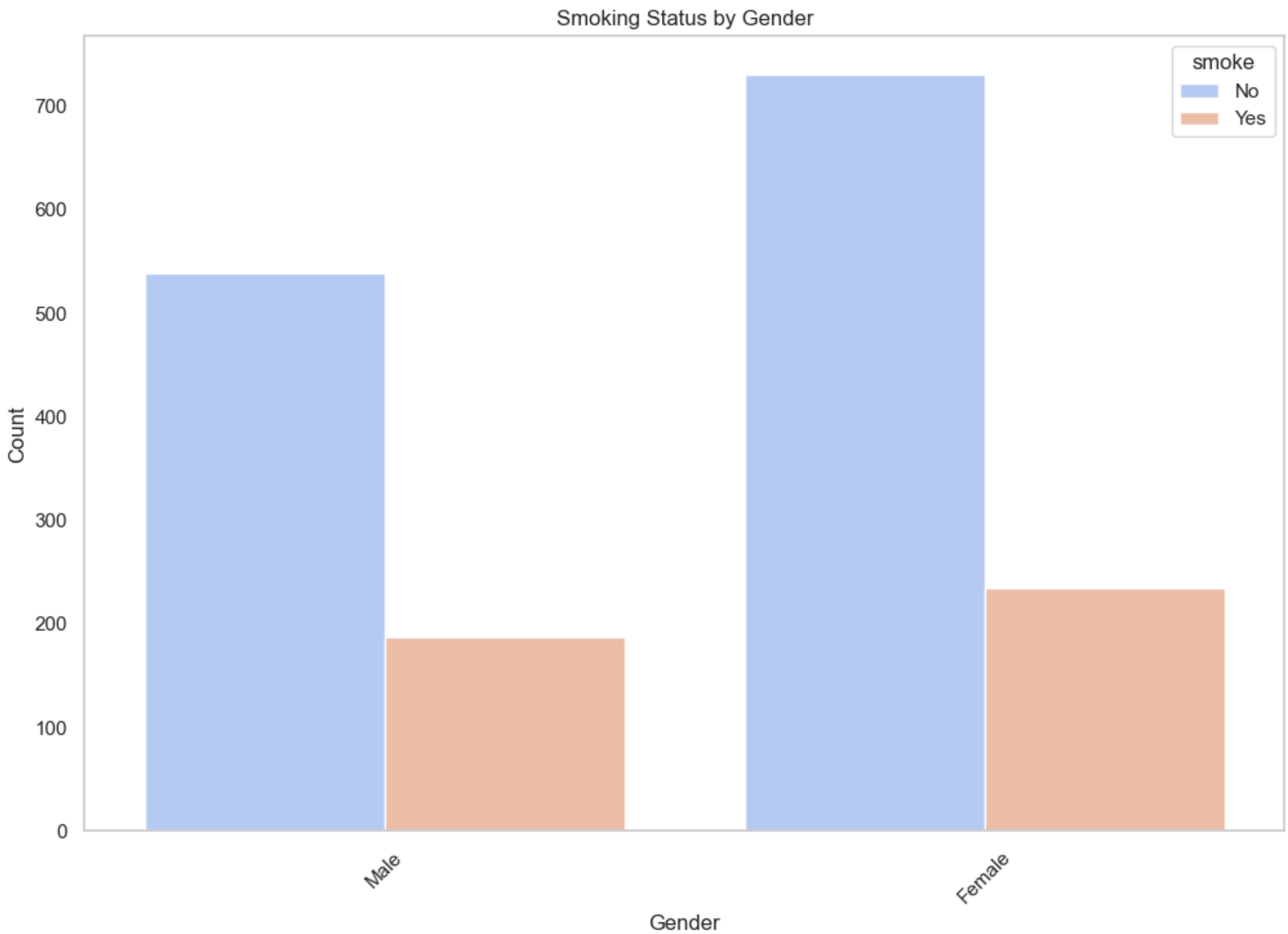


In [42]:
```
#Plot smoking status against region.
visualiser.plot_countplot(
    x_column='region',        # The column to plot on the x-axis
    hue_column='smoke',       # The column for colour grouping (smoking status)
    palette="Set1",       # Color palette
    title="Smoking Status by Region",  # Plot title
    xlabel="Region",          # label for x-axis
    ylabel="Count"            # label for y-axis
)
```

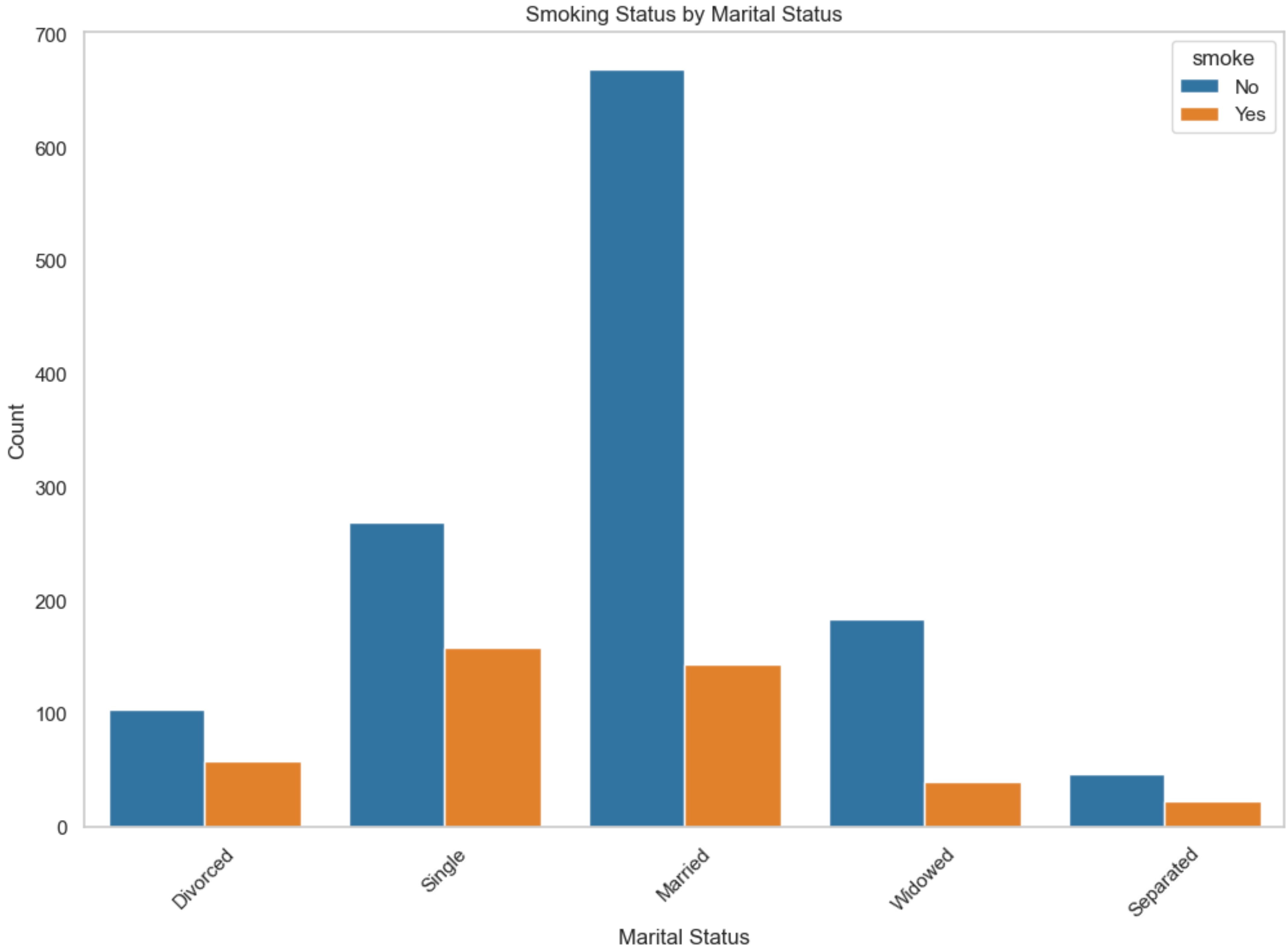


In [43]:
```
#Plot smoking status against gender.
visualiser.plot_countplot(
    x_column='gender',        # The column to plot on the x-axis (gender)
    hue_column='smoke',       # The column for colour grouping (smoking status)
    palette="coolwarm",           # Color palette
    title="Smoking Status by Gender",  # Plot title
    xlabel="Gender",          # label for x-axis
```

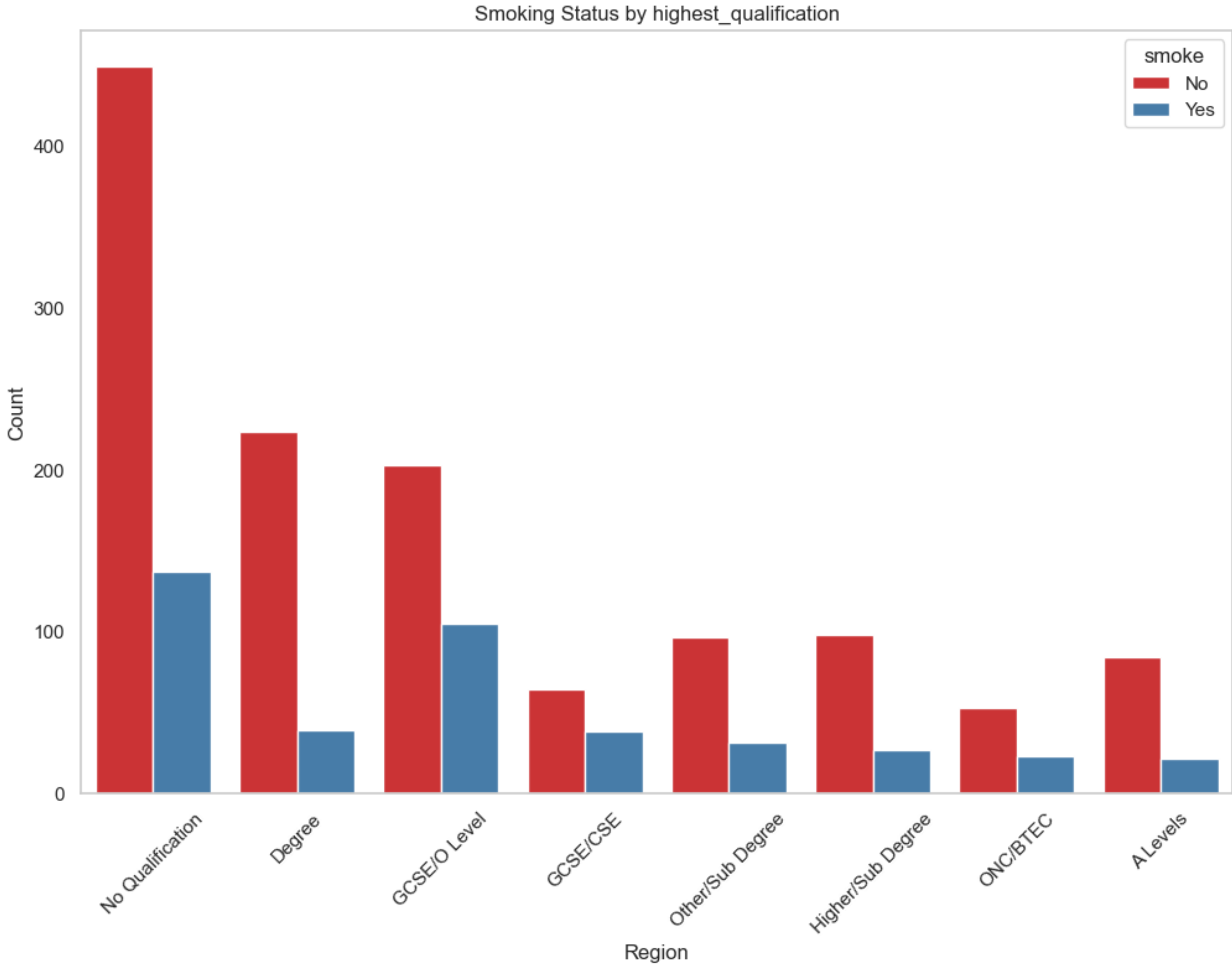
```
    ylabel="Count"            # label for y-axis
)
```



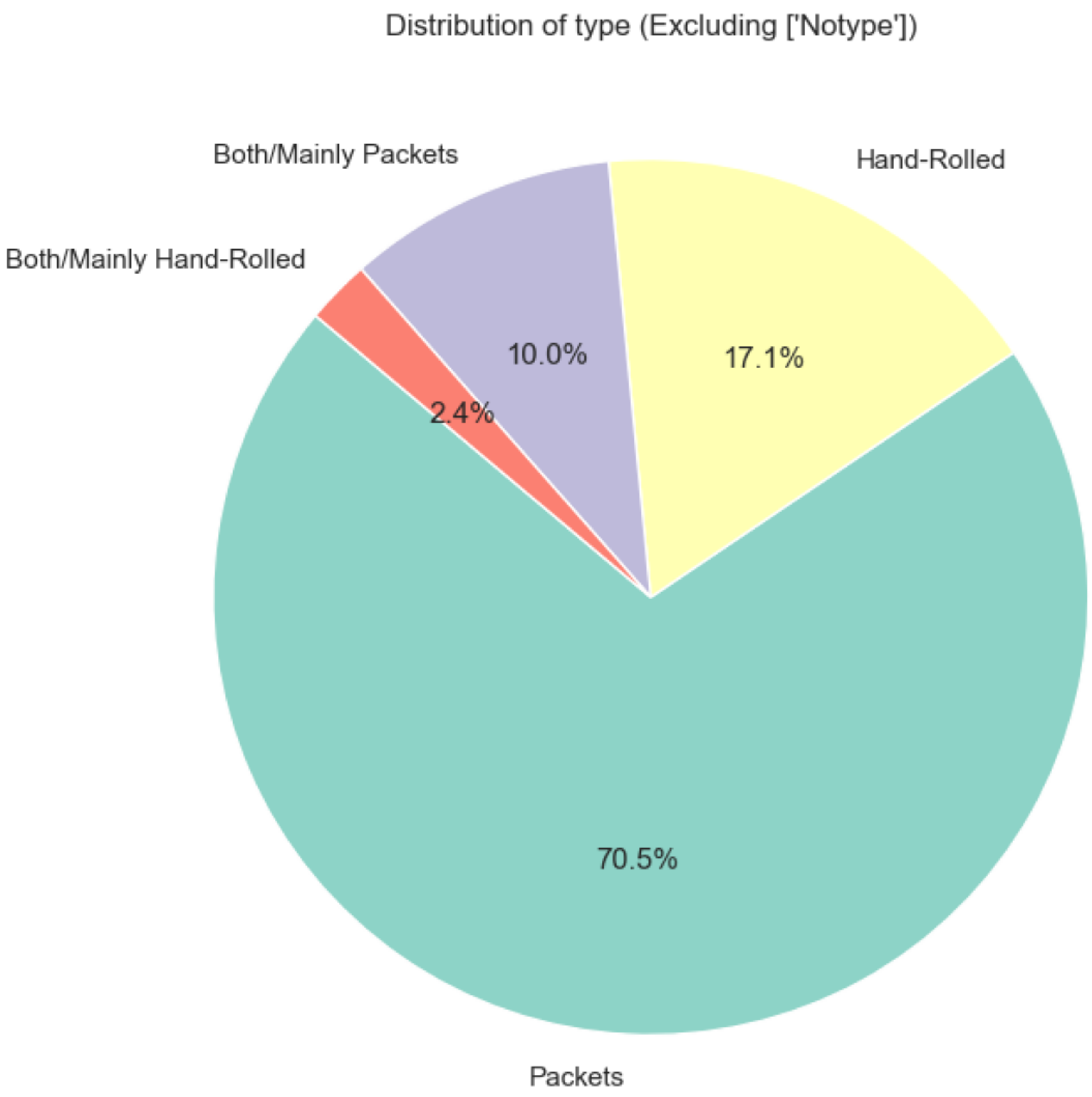
In [44]:
```
#Plot smoking status against marital status.
visualiser.plot_countplot(
    x_column='marital_status',        # The column to plot on the x-axis (marital_status)
    hue_column='smoke',               # The column for colour grouping (smoking status)
    palette=["#1f77b4", "#ff7f0e"],   # Color codes
    title="Smoking Status by Marital Status",  # Plot title
    xlabel="Marital Status",          # label for x-axis
    ylabel="Count"                    # label for y-axis
)
```



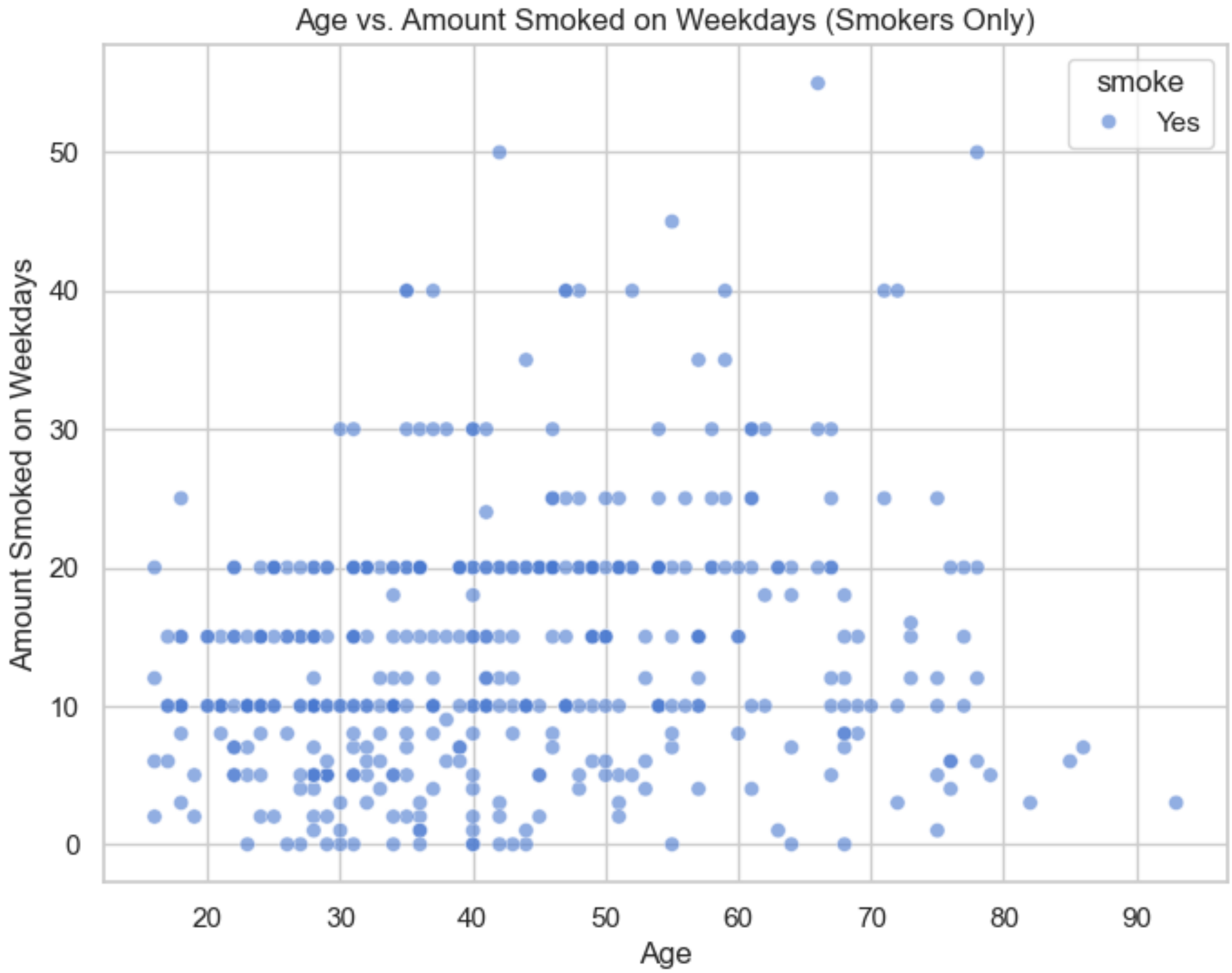
In [45]:
```
#Plot smoking status against region.
visualiser.plot_countplot(
    x_column='highest_qualification',        # The column to plot on the x-axis
    hue_column='smoke',       # The column for colour grouping (smoking status)
    palette="Set1",       # Color palette
    title="Smoking Status by highest_qualification",  # Plot title
    xlabel="Region",          # label for x-axis
    ylabel="Count"            # label for y-axis
)
```



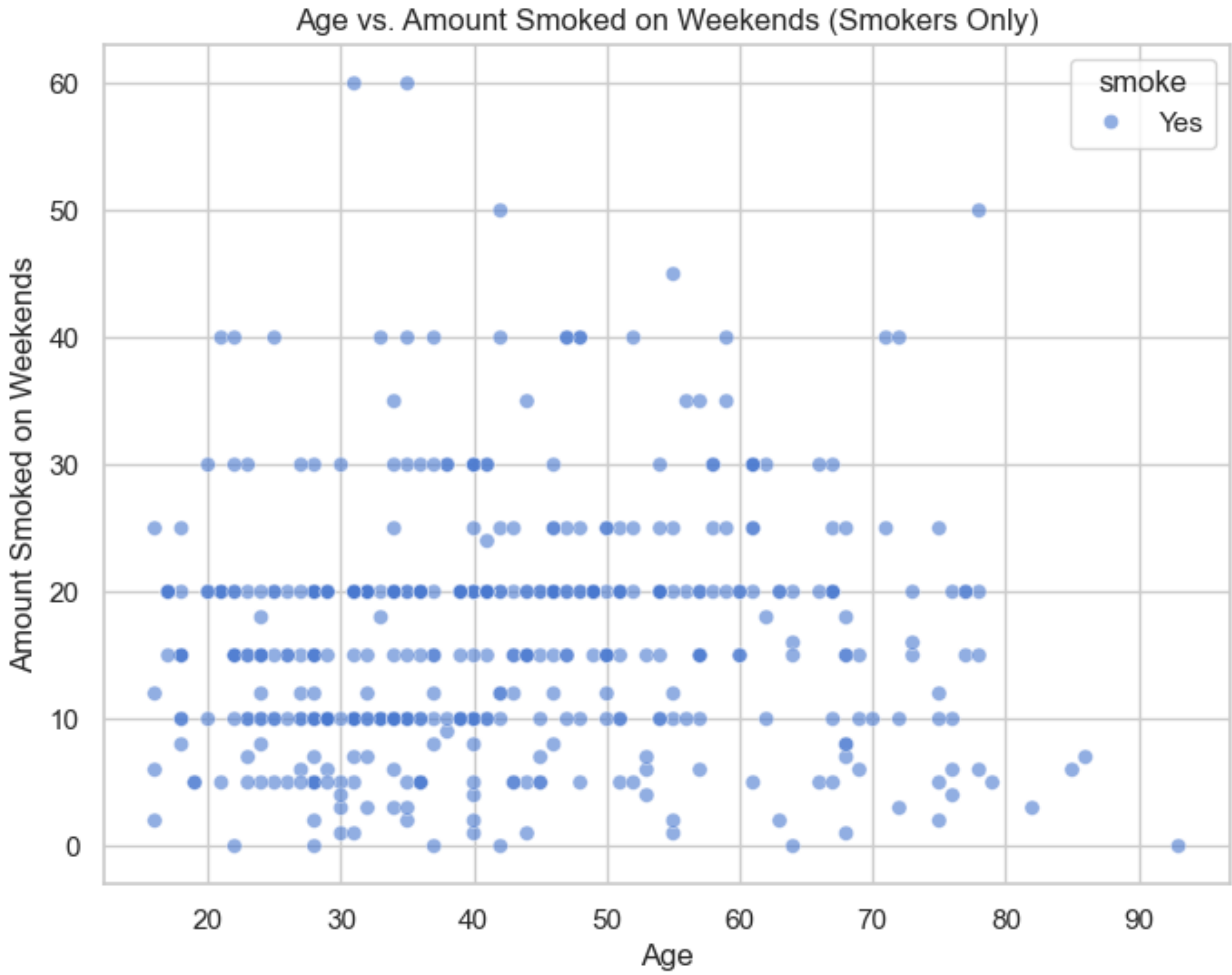
In [46]:
```
# Exclude "Notype" from the 'type' column because 'Notype' was used to fill non-smokers cells in the type column.
visualiser.plot_pie_distribution('type', exclude_values=['Notype'])
```



In [47]:
```
# Scatterplot of age vs. amount smoked on weekdays for smokers only
visualiser.plot_scatter(x_col='age', y_col='amt_weekdays', hue_col='smoke',
                        title="Age vs. Amount Smoked on Weekdays (Smokers Only)", xlabel="Age",
                        ylabel="Amount Smoked on Weekdays", filter_smokers=True)
```

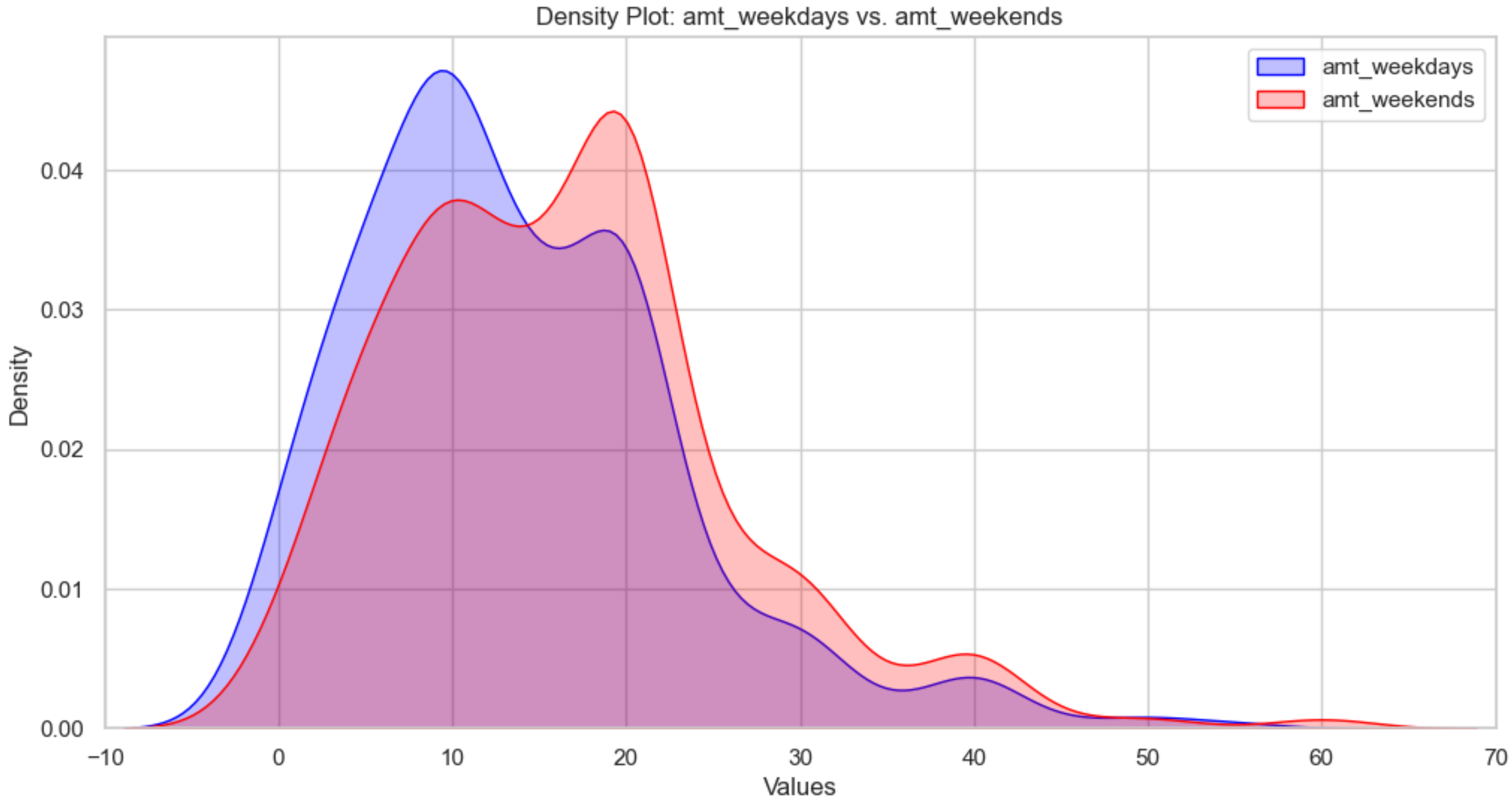


In [48]:
```
# Scatterplot of another set of columns for smokers only
visualiser.plot_scatter(x_col='age', y_col='amt_weekends', hue_col='smoke',
                        title="Age vs. Amount Smoked on Weekends (Smokers Only)", xlabel="Age",
                        ylabel="Amount Smoked on Weekends", filter_smokers=True)
```

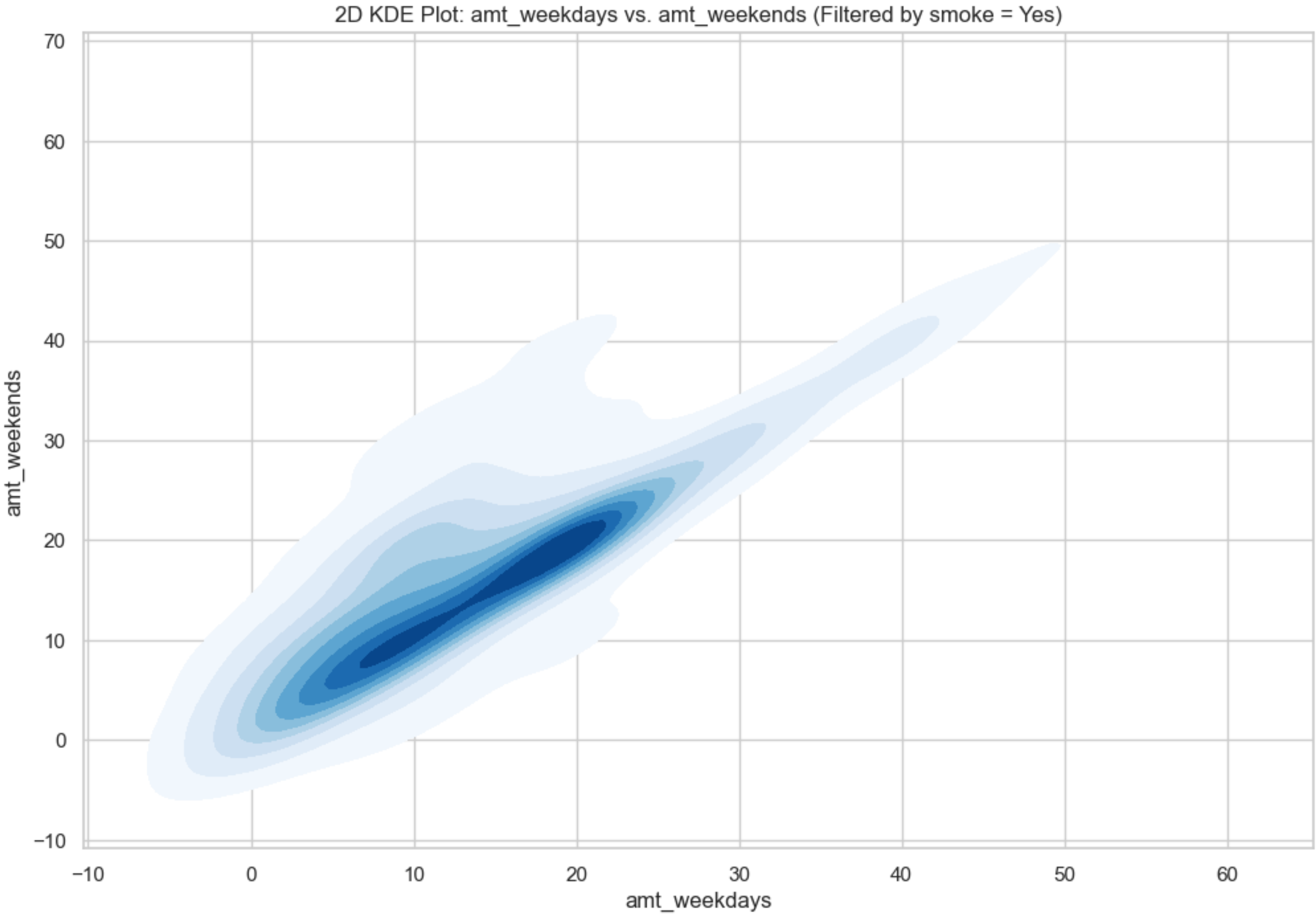


In [49]:

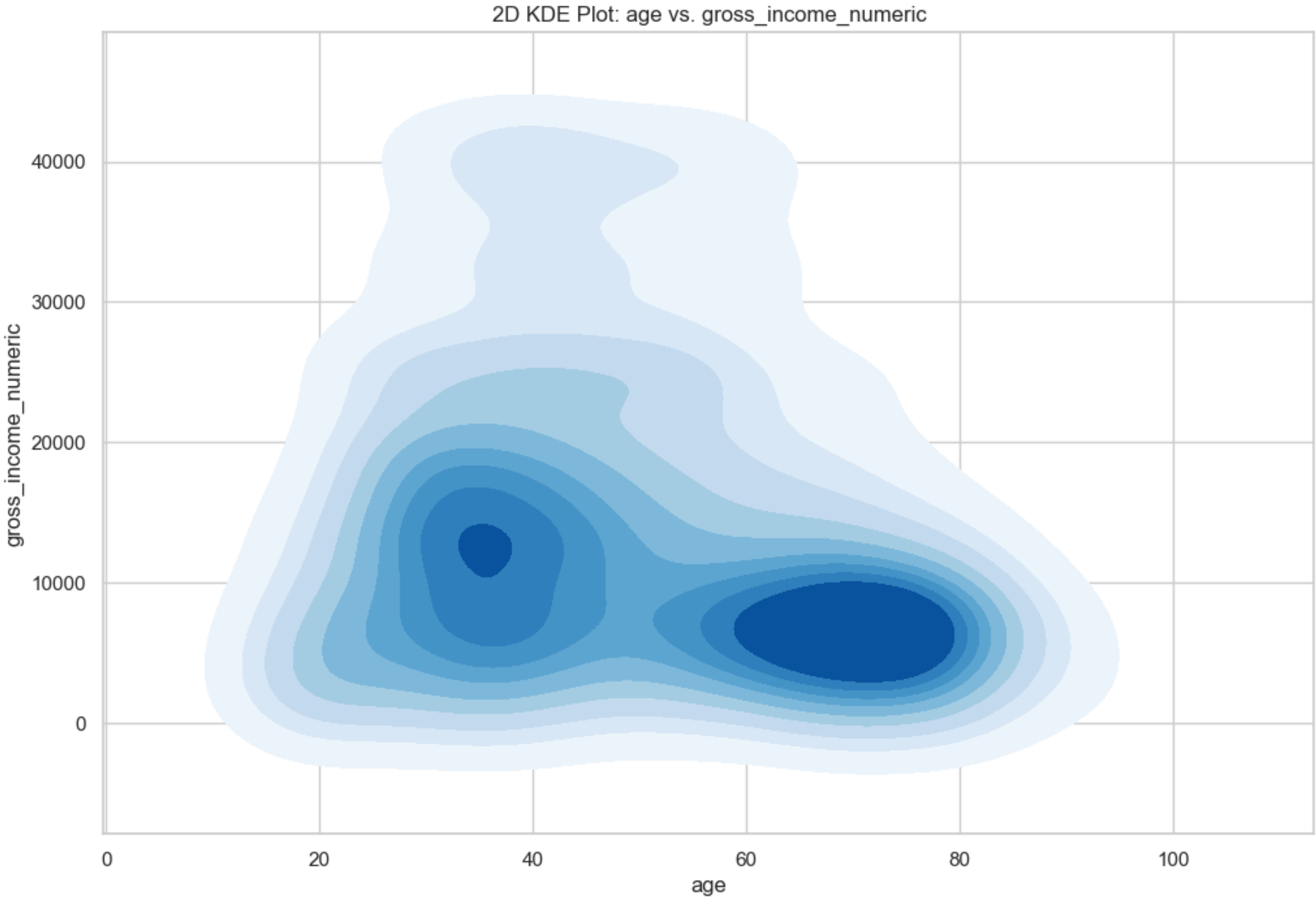
```
        """
        A KDE plot estimates the probability density function of a continuous variable,
        giving a smooth curve that shows how data points are distributed over a range.
        This KDE plot simplifies the visualization and make it easier to see where most values concentrate
        without the noisiness that often comes from frequencies in smaller datasets.

        Plot the KDE (density) plot for smoking amounts on weekdays and weekends,
        excluding non-smokers ('smoke' == 'No'), with x-axis limits set from -10 to 70,
        and adjusted bandwidth for clearer curves.
        """
# Plot density for 'amt_weekdays' and 'amt_weekends', filtering by 'smoke' == 'Yes'
visualiser.plot_density('amt_weekdays', 'amt_weekends', filter_column='smoke', filter_value='Yes', xlim=(-10, 70))
```



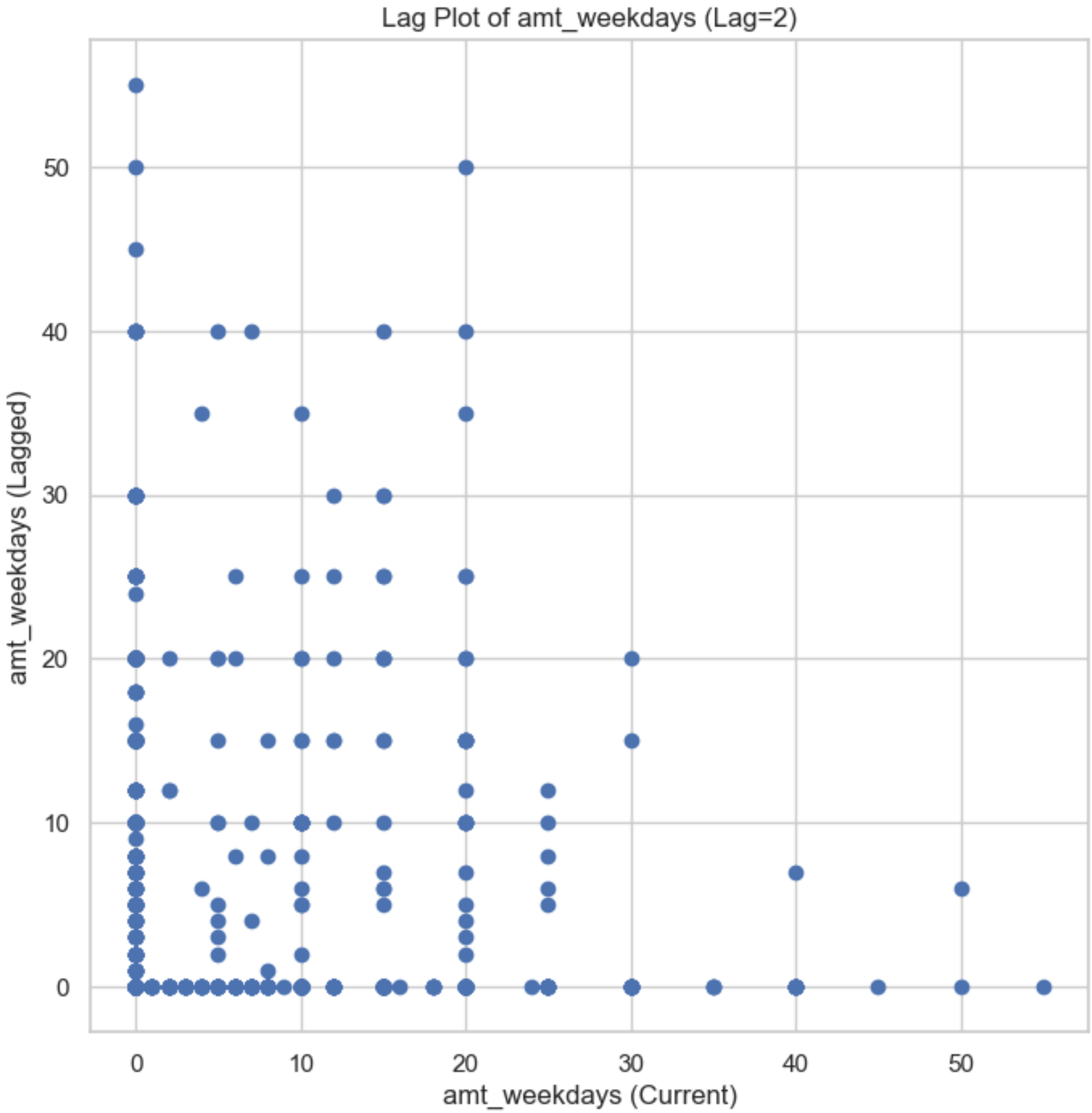
In [50]:
```
# Plot relationship between 'amt_weekdays' and 'amt_weekends', filtering smokers only
visualiser.plot_2d_kde('amt_weekdays', 'amt_weekends', filter_col='smoke', filter_value='Yes')
```



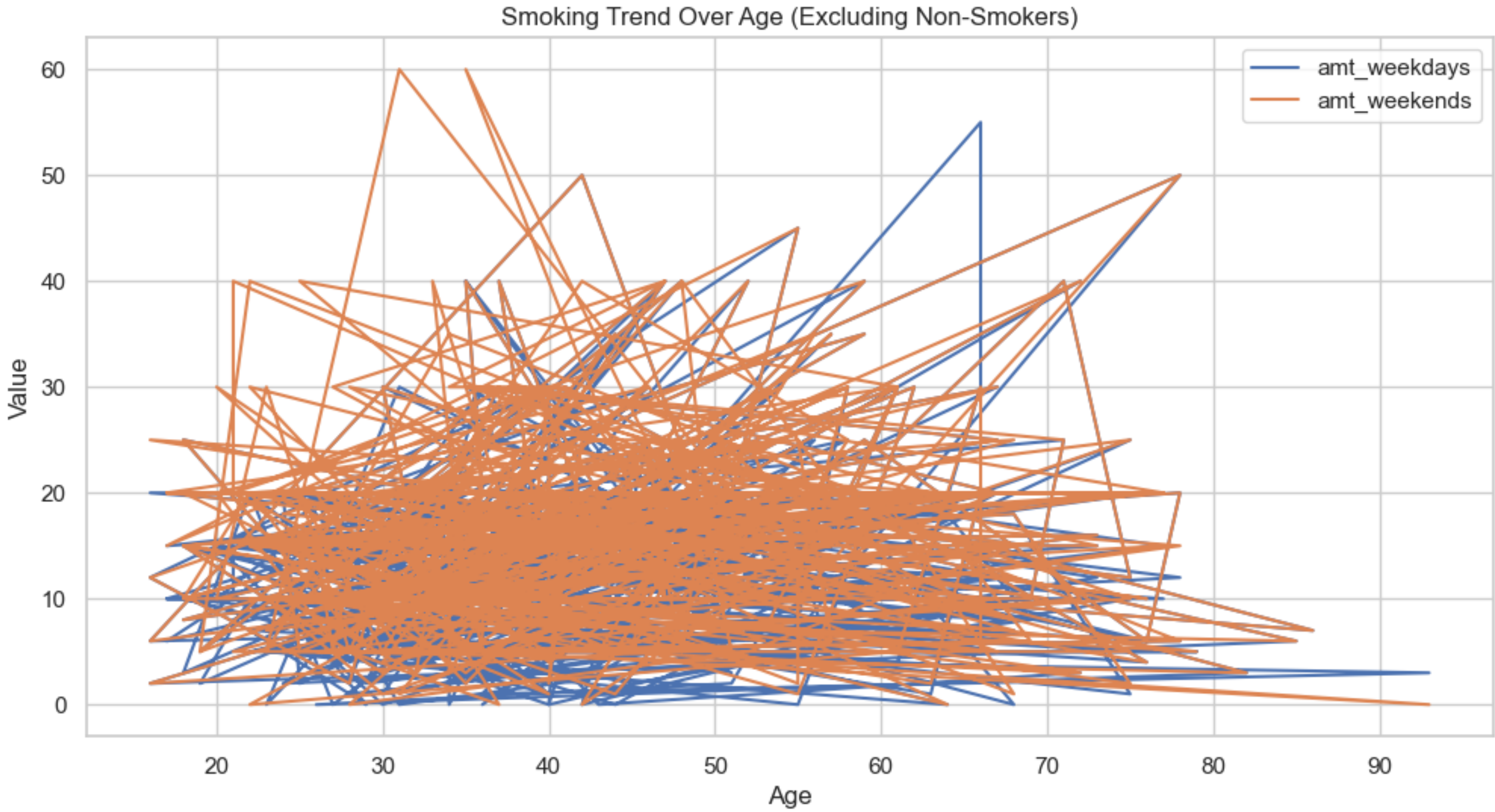
In [51]:
```
# Plot relationship between 'age' and 'gross_income_numeric', without filtering
visualiser.plot_2d_kde('age', 'gross_income_numeric')
```



In [52]:
```
# Call the method with a specific column name
visualiser.plot_lag('amt_weekdays', lag=2)
```



In [53]:
```
#Plot the trend of smoking amounts over age
visualiser.plot_line_plot_trend(
    age_column='age',
    trend_columns=['amt_weekdays', 'amt_weekends'],
    filter_column='smoke',
    filter_value='Yes',
    title="Smoking Trend Over Age (Excluding Non-Smokers)"
)
```



VISUALISER

The dataset visualisation analysis reveals several key insights into gender distribution, age demographics, income levels, regional diversity, and smoking habits.

Gender and Age Distribution: The gender distribution indicates that females make up a larger proportion of the dataset (57.1%) compared to males (42.9%). The age distribution frequency also reveals that there were more people aged between 30 and 40 in the dataset. Focusing on gender and age are important factors (Groot et al., 2018; Leiter, Veluswamy and Wisnivesky, 2023) that aid in guiding decisions concerning the prediction of an individual's likelihood of smoking.

Age and Smoking Habits: The age box plot displays differences between smokers and non-smokers. The median age for non-smokers is 51 years, while for smokers, it is 40 years. For non-smokers, the interquartile range (IQR) extends from 38 to 69 years, falling between 16 and 97 years. For smokers, the IQR ranges from 30 to 54 years, with most ages between 16 and 86 years. An outlier at 93 years indicates one exceptionally aged smoker.

Gross Income Distribution: Analysis of gross income shows that the 5,200 to 10,400 income range accounts for the highest percentage (23.4%) of the dataset, while the unknown category and 28,600 to 36,400 range have the lowest representation at 1.1% and 4.7%, respectively. The numeric gross income obtained from mapping the categorical gross income to numerical points was also visualised using a box plot and it shows that there were outliers at 32500 for both non-smokers and smokers.

Regional Smoking Trends: Data on smoking by region highlights diversity across the United Kingdom. The North and Midlands & East Anglia regions represent the largest populations in the dataset and include the highest number of smokers.

Gender and Smoking Habits: Further visualisations explored the gender distribution of smokers and non-smokers, revealing patterns that enhance understanding of demographic trends within the dataset.

Marital Status and Smoking: Histograms indicate that married individuals represent the largest group among smokers, further contributing to the overall smoking numbers in the dataset. Further analysis into their smoking patterns and types indicated that smokers smoked more packets with 70.5%, followed by hand-rolled with a percentage of 17.1%, both/mainly packets with 10.0%, and both/mainly hand-rolled with 2.4%. The 'Notype' was excluded because it was used to fill in the smoking cells for non-smokers.

Smoking Patterns on Weekdays and Weekends: A scatter plot for smokers (excluding non-smokers) was used to explore the amount smoked on weekdays and weekends. While no linear relationship was evident, the visualisation identified a few outliers.

KDE Plot for Smoking Amounts: A Kernel Density Estimate (KDE) plot generates a smooth curve to represent the probability density function (PDF) of a continuous variable, showcasing the distribution of data points across a range (Zambom and Dias, 2012). This plot simplifies visualisation by highlighting areas of concentration while minimising the visual noise often found in smaller datasets. The (KDE) plot provides a smooth representation of smoking amounts on weekdays (amt_weekdays) and weekends (amt_weekends) for smokers. By focusing on smokers only and customising the x-axis range and smoothing levels, the KDE plot highlights areas of concentration while reducing distractions from outliers or uneven data distribution. The 2D Kernel Density Estimate (KDE) plot above shows the distribution of the amount on weekends across different amounts on weekdays. The distribution of gross_income_numeric across different ages was also visualised with 2D - KDE. This visualisation helps to identify where gross_income_numeric and amounts on weekends are most densely aggregated, which is useful for pinpointing areas of high activity.

Smoking Trend Over Age: A line plot titled "Smoking Trend Over Age (Excluding Non-Smokers)" visualises smoking amounts for smokers only across different ages. The x-axis represents age, while separate lines compare smoking amounts on weekdays (amt_weekdays) and weekends (amt_weekends). This visualisation aims to provide insights into how smoking habits evolve with age among smokers; however, no clear insight could be deduced from this.

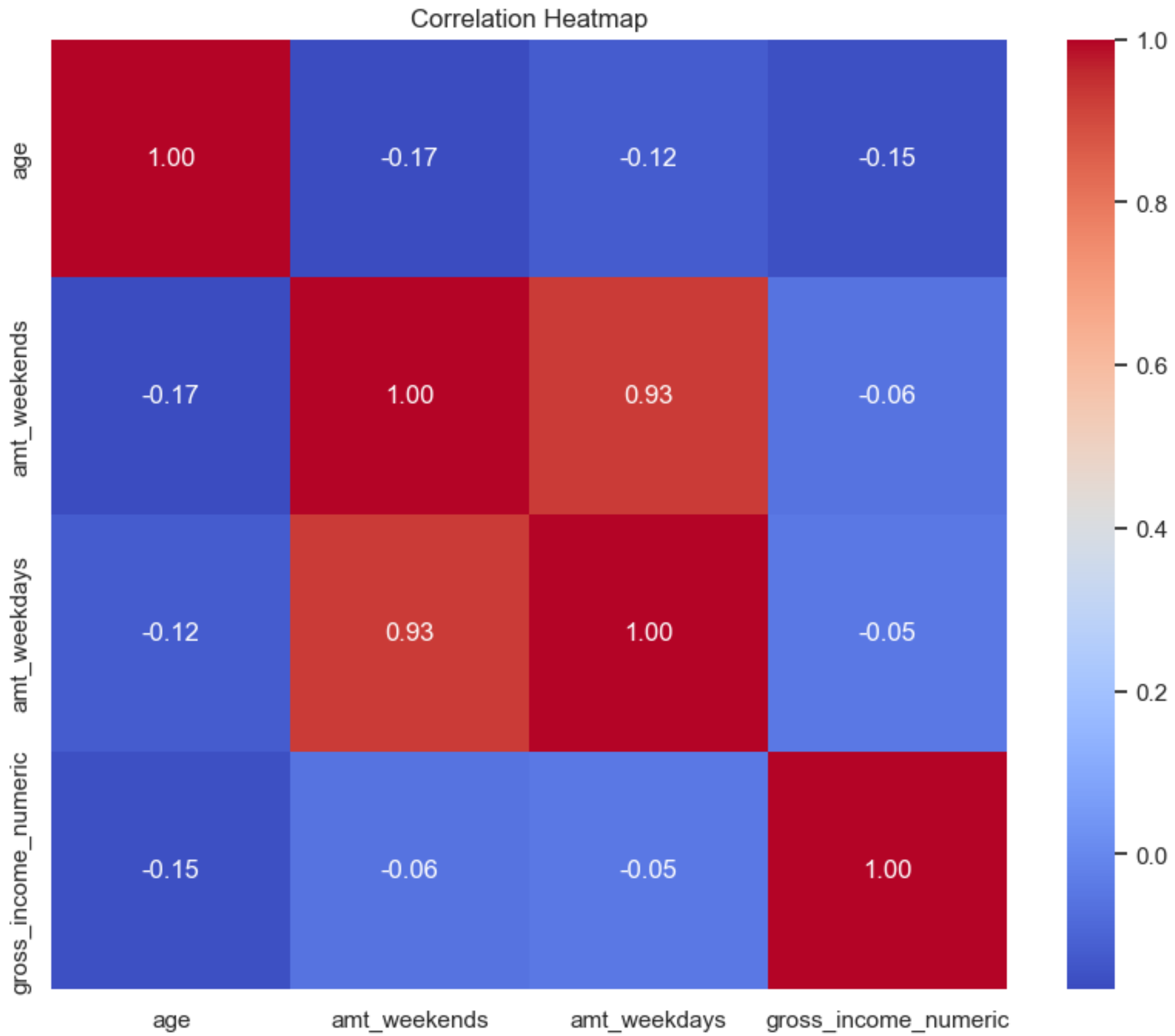
The combination of visualisations and statistical analyses offers a comprehensive understanding of the dataset, highlighting essential factors such as gender, age, income, and regional diversity in predicting smoking habits. Tools like box plots, KDE plots, and scatter plots make these trends accessible and actionable.

In [ ]:

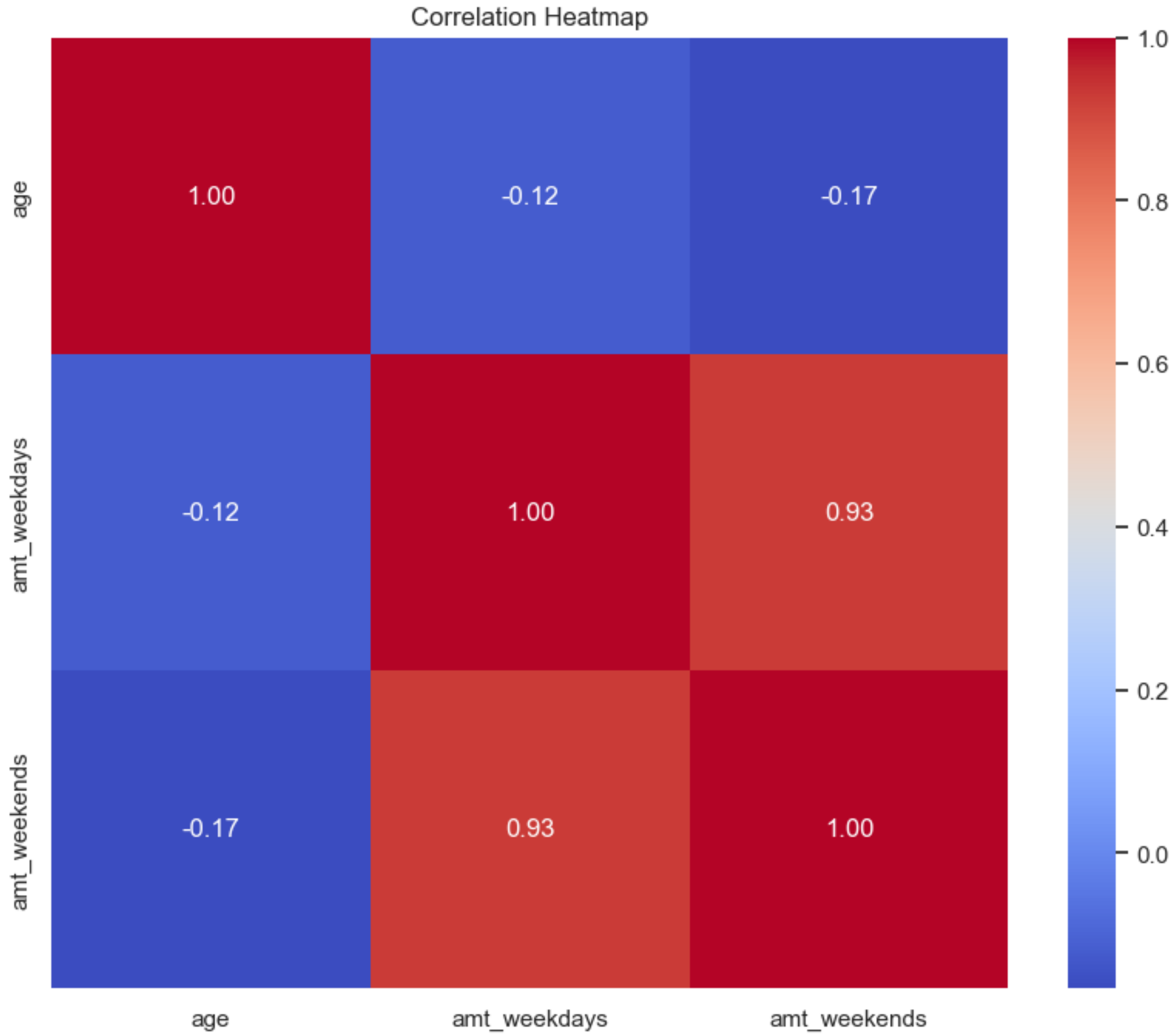
```
```

In [55]:

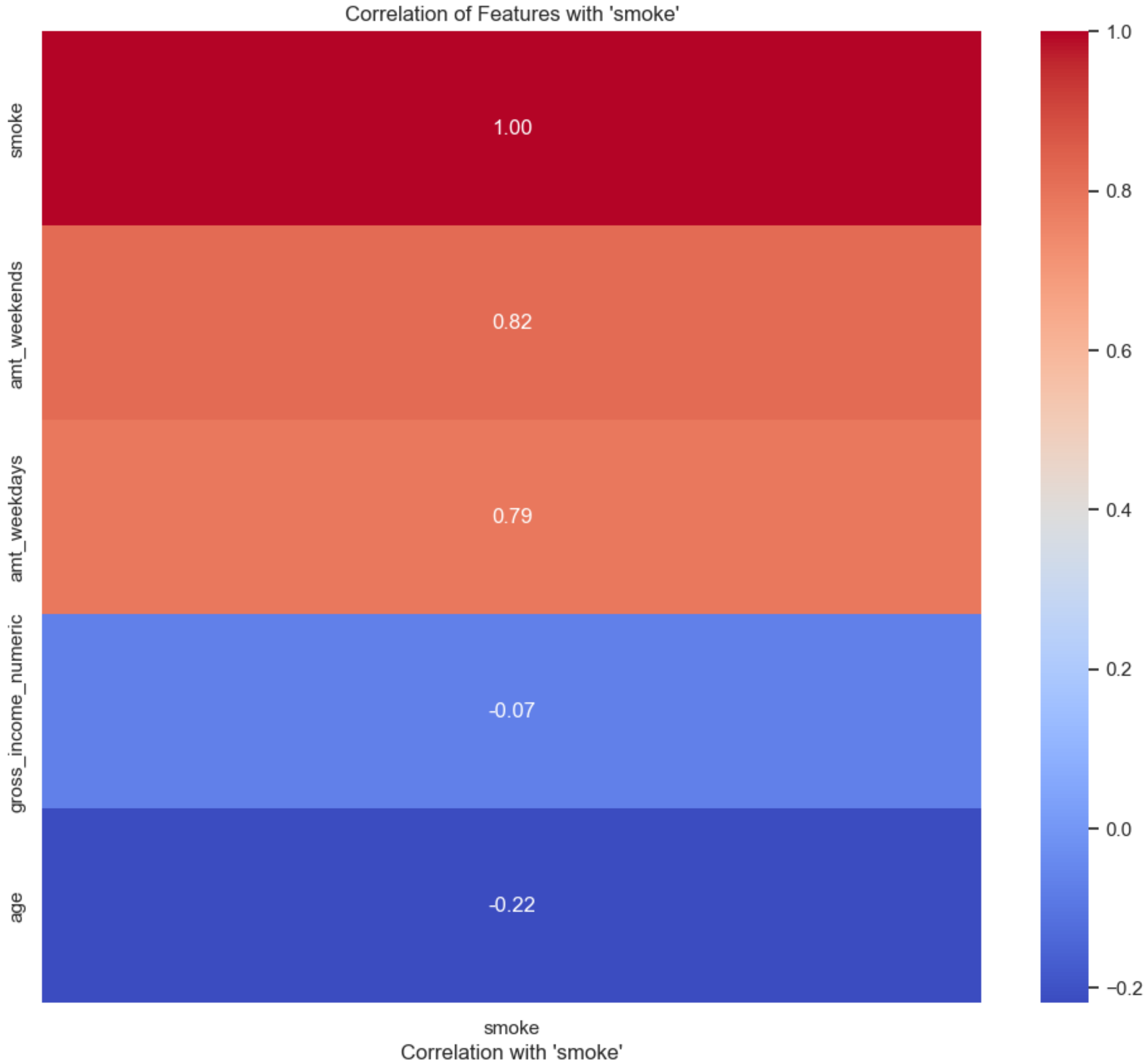
```
# Plot correlation heatmap for all numeric columns
visualiser.plot_correlation_heatmap_for_columns()
```



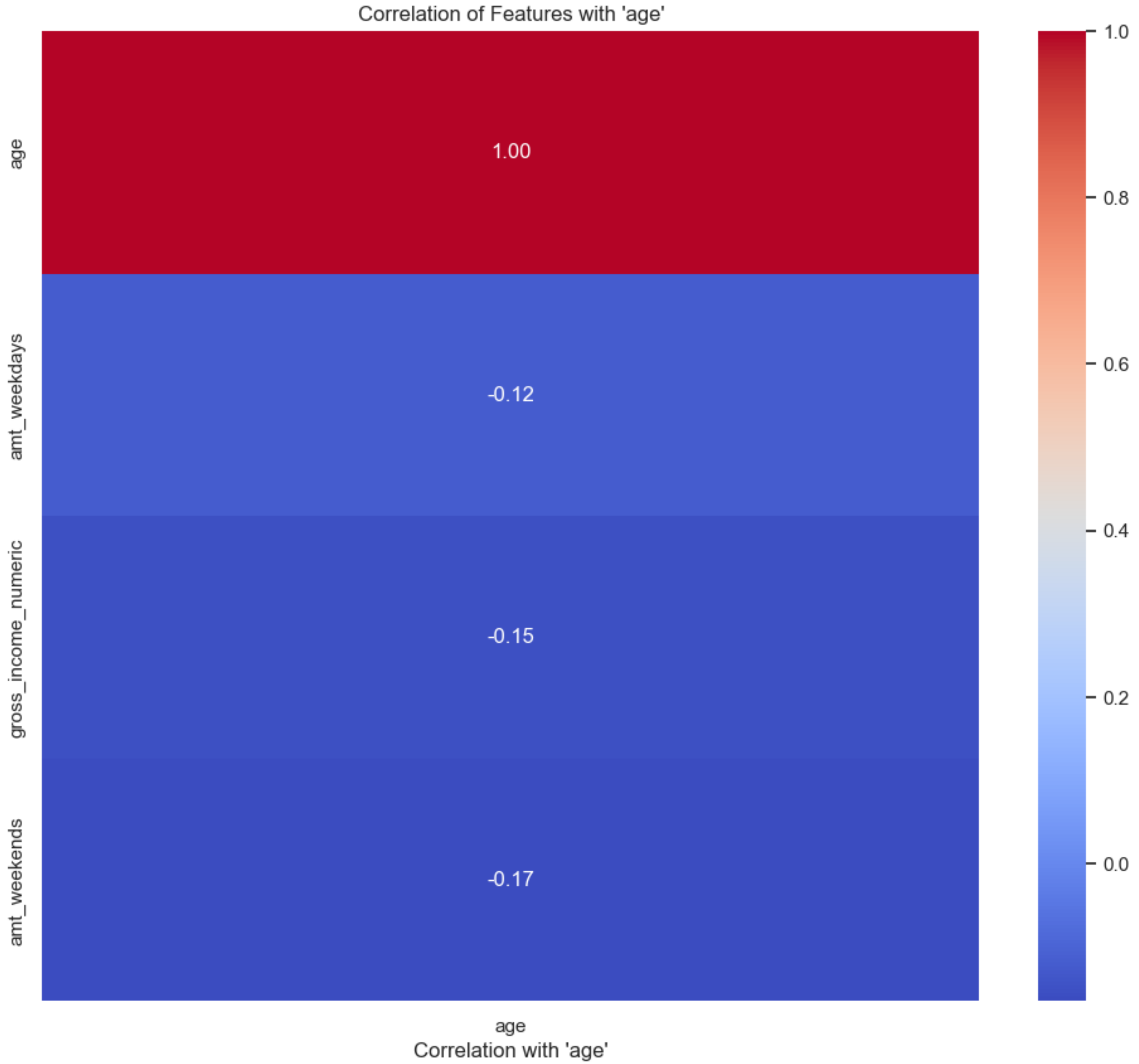
In [56]:
```
# Plot correlation heatmap for selected columns
visualiser.plot_correlation_heatmap_for_columns(['age', 'amt_weekdays', 'amt_weekends'])
```



In [57]:
```
#Plot correlation of features with smoking status
visualiser.plot_correlation_with_target('smoke')
```



In [58]:
```
#Plot correlation of features with age
visualiser.plot_correlation_with_target('age')
```



CORRELATION HEATMAP

The correlation heatmaps for all numerical columns in the dataset were visualised to examine their relationships with the target variable, smoke. The analysis revealed the following:

Age demonstrated a negative correlation coefficient of -0.22, indicating an inverse relationship with the likelihood of smoking. This suggests that as age increases, the likelihood of smoking tends to decrease. The amount smoked on weekdays and the amount smoked on weekends both exhibited strong positive correlation coefficients of 0.79 and 0.82, respectively. These values suggest a strong direct relationship, where individuals who smoke more on weekdays are also likely to smoke more on weekends. The numeric gross income column exhibited a near-zero correlation, suggesting a minimal or insignificant relationship with the smoking variable.

4.0 MACHINE LEARNING

In [61]:
```
#Feature Selection
from sklearn.feature_selection import SelectKBest, f_classif

# Model Evaluation
from sklearn.metrics import accuracy_score, classification_report
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler

# Classification Algorithms
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
```

In [62]:
```
# Modeling Section
class ClassificationModel:
    def __init__(self, data, target_column):
        """
        Initialise the model with the dataset and target column.
        Parameters:
            data (DataFrame): The dataset to work with.
            target_column (str): The name of the target column for classification.
        """
        self.original_data = data  # Store the original dataset for reference or reuse
        self.data = data.copy()  # Work with a copy of the dataset to avoid altering the original
        self.target_column = target_column
        self.encoder = LabelEncoder()  # Encoder for categorical columns
        self.model_accuracies = {}  # Dictionary to store model accuracies
        self.feature_columns = []  # To store feature columns after encoding

    def encode_categorical_columns(self, data=None):
        """
        Convert all categorical columns in the dataset, including the target column, into encoded numerical representations.
        """
        if data is None:
            data = self.data  # Use the instance's data if no data is passed
        for col in data.select_dtypes(include='object').columns:
            data[col] = self.encoder.fit_transform(data[col])
        print("Categorical columns encoded.")
        # Update the feature columns after encoding
        self.feature_columns = data.columns.tolist()
        return data

    def dataset_head(self, rows, columns=None):
        """
```

```
        Shows the dataset or specified columns' first few rows.
        Parameters:
            rows (int): Number of rows to display.
            columns (list, optional): List of column names to display. If None, display all columns.
        """
        data_to_display = self.data[columns] if columns else self.data
        print("First few rows of the dataset:")
        print(data_to_display.head(rows))

    def dataset_tail(self, rows, columns=None):
        """
        Shows the dataset or specified columns' last few rows.
        Parameters:
            rows (int): Number of rows to display.
            columns (list, optional): List of column names to display. If None, display all columns.
        """
        data_to_display = self.data[columns] if columns else self.data
        print("Last few rows of the dataset:")
        print(data_to_display.tail(rows))

    def align_columns(self, new_data):
        """
        Align the columns of new data with the training data.
        Parameters:
            new_data (DataFrame): New data to align with the model's features.
        Returns:
            DataFrame: The new data with columns aligned to the training data.
        """
        return new_data[self.feature_columns]

    def drop_columns(self, columns):
        """
        Drop specified columns from the data frame.
        Parameters:
            columns (list): List of column names to drop.
        """
        self.data.drop(columns=columns, inplace=True, errors='ignore')
        print(f"Dropped columns: {columns}")

    def select_best_features(self, X, y, k=8):
        """
        Select the best features based on ANOVA F-test scores and visualises them.
        Parameters:
            X (DataFrame): Feature dataset.
            y (Series): Target variable.
            k (int, optional): Number of features to select (default is 8).
        Returns:
            DataFrame: DataFrame with selected features and their corresponding scores.
        """
        # Initialize SelectKBest to select the best k features based on ANOVA F-test
        selector = SelectKBest(f_classif, k=k)
        # Apply the selector to the data
        X_selected = selector.fit_transform(X, y)
        # Create a list of selected feature names and their scores
        selected_features = X.columns[selector.get_support()]
        feature_scores = selector.scores_[selector.get_support()]
```

```
        # Create a DataFrame to store features and their corresponding scores
        feature_score_df = pd.DataFrame({'Features': selected_features, 'Scores': feature_scores})
        # Sort the DataFrame in descending order based on scores
        feature_score_df = feature_score_df.sort_values(by='Scores', ascending=False)

        # Plotting a barplot for better visualisation of features and their scores
        plt.figure(figsize=(12, 8))
        ax = sns.barplot(x=feature_score_df['Scores'], y=feature_score_df['Features'])
        plt.title('Feature Scores', fontsize=18)
        plt.xlabel('Scores', fontsize=16)
        plt.ylabel('Features', fontsize=16)
        # Annotating the barplot with feature scores
        for lab in ax.containers:
            ax.bar_label(lab)
        # Show the plot
        plt.tight_layout()
        plt.show()
        return feature_score_df

    def split_data(self, test_size=0.20, random_state=42):
        """
        Split the dataset into training and testing sets, and ensure data format consistency.
        Returns:
            X_train, X_test, y_train, y_test: All as NumPy arrays.
        """
        if self.target_column not in self.data.columns:
            print(f"Error: Target column '{self.target_column}' not found in the dataset.")
            return
        # Define features (X) and target (y)
        X = self.data.drop(columns=[self.target_column]).values  # Convert to NumPy array
        y = self.data[self.target_column].values  # Convert to NumPy array

        # Perform the train-test split
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size, random_state=random_state)
        print("Dataset split into training and testing sets.")
        return X_train, X_test, y_train, y_test


    def train_and_evaluate_models(self, X_train, y_train, X_test, y_test, model, name):
        """
        Train and evaluate a single classification model with consistent data formatting.
        """
        # Scale the data
        scaler = StandardScaler()
        X_train_scaled = scaler.fit_transform(X_train)  # Ensure consistent format (NumPy array)
        X_test_scaled = scaler.transform(X_test)        # Ensure consistent format (NumPy array)
        # Train the model
        model.fit(X_train_scaled, y_train)
        # Predict on test data
        y_pred = model.predict(X_test_scaled)
        # Compute accuracy
        accuracy = accuracy_score(y_test, y_pred)
        # Store the accuracy in a dictionary
        self.model_accuracies[name] = accuracy
        # Print evaluation metrics
        print(f"Model: {name}")
```

```
        print(f"Accuracy: {accuracy}")
        print(f"Classification Report:\n{classification_report(y_test, y_pred, zero_division=1)}")


    def predict_on_test_set(self, model, X_test):
        """
        Predict on the test set using the trained model.
        Parameters:
            model (object): The trained classification model.
            X_test (DataFrame): The test features.
        Returns:
            numpy.ndarray: The predicted values for the test set.
        """
        if not hasattr(model, "predict"):
            print("Error: The provided model does not have a 'predict' method.")
            return None
        try:
            y_pred = model.predict(X_test)
            print("Predictions on test set completed.")
            return y_pred
        except NotFittedError:
            print("Error: The provided model is not fitted. Please train the model before prediction.")
            return None

    def plot_model_accuracies(self):
        """
        Plot the accuracies of different models as a histogram.
        Indicate it if no models have been trained.
        Returns:
            None: Displays a bar plot of model accuracies.
        """
        if not self.model_accuracies:
            print("No models have been trained yet. Please train models before plotting accuracies.")
            return
        # Extract model names and their accuracies
        models = list(self.model_accuracies.keys())
        accuracies = list(self.model_accuracies.values())
        # Plot the histogram
        plt.figure(figsize=(8, 6))
        plt.bar(models, accuracies, color='skyblue', edgecolor='black')
        # Add labels and title
        plt.title("Model Accuracies", fontsize=14)
        plt.xlabel("Models", fontsize=12)
        plt.ylabel("Accuracy", fontsize=12)
        plt.ylim(0, 1)  # Accuracy is between 0 and 1
        # Annotate bars with accuracy values
        for i, accuracy in enumerate(accuracies):
            plt.text(i, accuracy + 0.02, f"{accuracy:.2f}", ha='center', fontsize=10)
        # Show the plot
        plt.tight_layout()
        plt.show()
```

In [63]:

```
# Create the classification model
classification_model = ClassificationModel(cleaned_data, target_column='smoke')
```

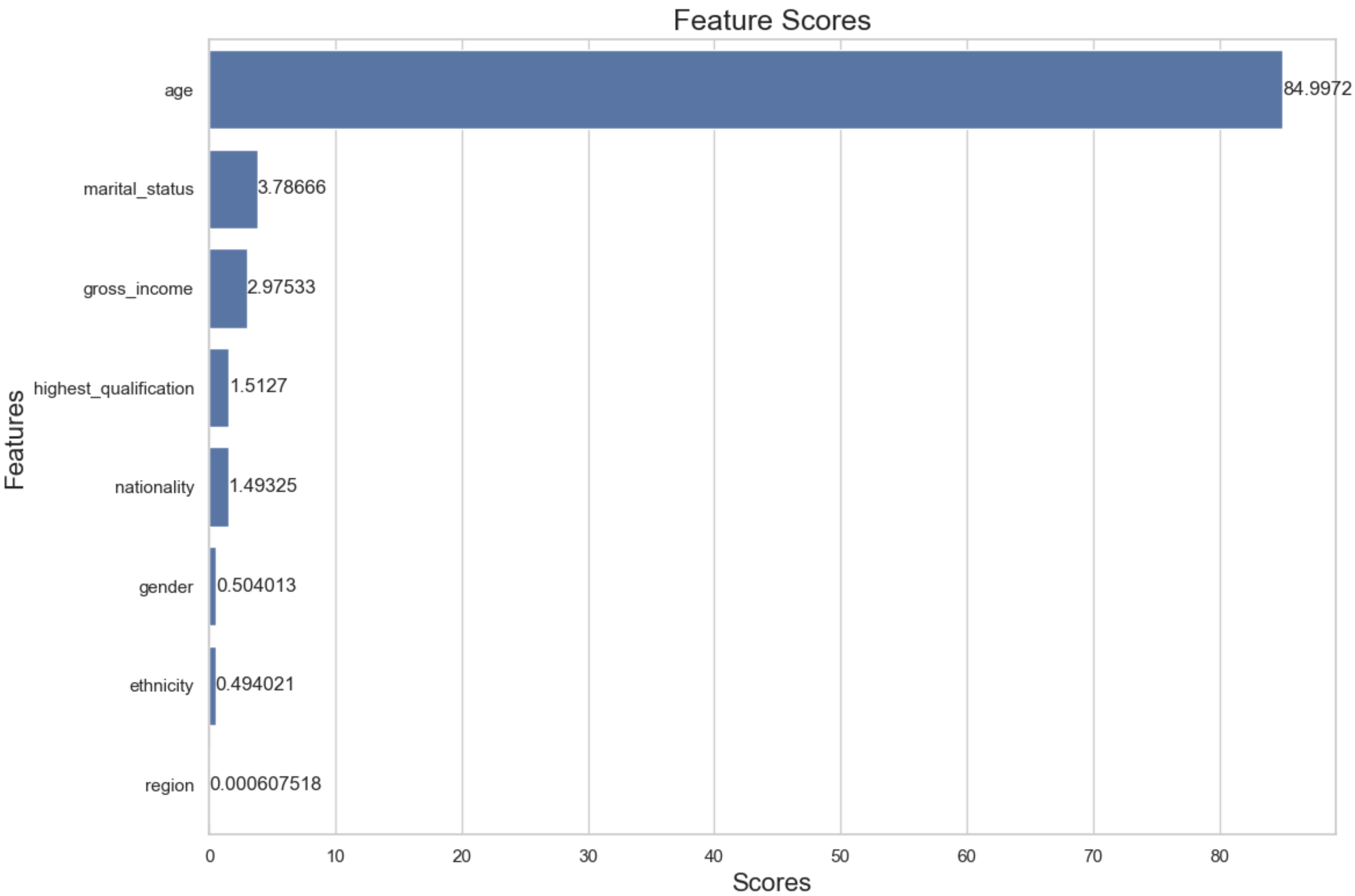
In [64]:
```
# Encode categorical columns
classification_model.encode_categorical_columns()

# Print the bottom of the dataset to confirm encoding
classification_model.dataset_tail(5)
```

```
Categorical columns encoded.
Last few rows of the dataset:
      gender  age  marital_status  highest_qualification  nationality  \
1686       1   22               3                      5            5
1687       0   49               0                      7            1
1688       1   45               1                      7            5
1689       0   51               1                      5            1
1690       1   31               1                      1            5

      ethnicity  gross_income  region  smoke  amt_weekends  amt_weekdays  \
1686          6             2       2      0           0.0           0.0
1687          6             2       2      1          20.0          20.0
1688          6             5       2      0           0.0           0.0
1689          6             2       2      1          20.0          20.0
1690          6             0       2      0           0.0           0.0

      type  gross_income_numeric
1686     3                3900.0
1687     2                3900.0
1688     3                7800.0
1689     4                3900.0
1690     3               13000.0
```

In [65]:
```
# Drop columns
columns_to_drop = ['type', 'gross_income_numeric', 'amt_weekends', 'amt_weekdays']
classification_model.drop_columns(columns_to_drop)
```

```
Dropped columns: ['type', 'gross_income_numeric', 'amt_weekends', 'amt_weekdays']
```

In [66]:
```
# Feature Scores
# Step 1: Prepare X (features) and y (target)
X = classification_model.data.drop(columns=[classification_model.target_column])
y = classification_model.data[classification_model.target_column]

# Step 2: Call the select_best_features method to find and plot the best features
classification_model.select_best_features(X, y, k=8)
```


Feature Scores
age
marital_status
gross_income
highest_qualification
nationality
gender
ethnicity
region
84.9972
3.78666
2.97533
1.5127
1.49325
0.504013
0.494021
0.000607518
0
10
20
30
40
50
60
70
80
Scores
Features

Out[66]:

	Features	Scores
1	age	84.997211
2	marital_status	3.786659
6	gross_income	2.975331
3	highest_qualification	1.512702
4	nationality	1.493248
0	gender	0.504013
5	ethnicity	0.494021
7	region	0.000608

In [67]:

```
# Drop columns due to low feature scores less than 0.5 threshold
columns_to_drop = ['region', 'ethnicity']
classification_model.drop_columns(columns_to_drop)
```

```
Dropped columns: ['region', 'ethnicity']
```

In [68]:

```
# Print the bottom of the dataset to confirm encoding
classification_model.dataset_head(4)
```

```
First few rows of the dataset:
   gender  age  marital_status  highest_qualification  nationality  \
0       1   38               0                      5            0
1       0   42               3                      5            0
2       1   40               1                      1            1
3       0   40               1                      1            1

   gross_income  smoke
0             2      0
1             8      1
2             4      0
3             0      0
```

In [69]:

```
# Split the data into training and testing sets with 25% allocated to the test set instead of the default 20%.
X_train, X_test, y_train, y_test = classification_model.split_data(test_size=0.25)
```

```
Dataset split into training and testing sets.
```

In [70]:

```
# Define the model and its name together
first_model = {"model": LogisticRegression(), "name": "Logistic Regression"}

# Call the method using the combined dictionary
classification_model.train_and_evaluate_models(
    X_train, y_train, X_test, y_test, first_model["model"], first_model["name"]
)
```

```
Model: Logistic Regression
Accuracy: 0.7659574468085106
Classification Report:
              precision    recall  f1-score   support

           0       0.78      0.97      0.86       328
           1       0.39      0.07      0.12        95

    accuracy                           0.77       423
   macro avg       0.59      0.52      0.49       423
weighted avg       0.69      0.77      0.70       423
```

In [71]:
```
# Define the model and its name together
second_model = {"model": RandomForestClassifier(), "name": "Random Forest"}

# Call the method using the combined dictionary
classification_model.train_and_evaluate_models(
    X_train, y_train, X_test, y_test, second_model["model"], second_model["name"]
)
```

```
Model: Random Forest
Accuracy: 0.7210401891252955
Classification Report:
              precision    recall  f1-score   support

           0       0.81      0.84      0.82       328
           1       0.36      0.31      0.33        95

    accuracy                           0.72       423
   macro avg       0.58      0.57      0.58       423
weighted avg       0.71      0.72      0.71       423
```

In [72]:
```
# Define the model and its name together
third_model = {"model": DecisionTreeClassifier(), "name": "Decision Tree"}

# Call the method using the combined dictionary
classification_model.train_and_evaluate_models(
    X_train, y_train, X_test, y_test, third_model["model"], third_model["name"]
)
```

```
Model: Decision Tree
Accuracy: 0.6595744680851063
Classification Report:
              precision    recall  f1-score   support

           0       0.80      0.74      0.77       328
           1       0.29      0.37      0.33        95

    accuracy                           0.66       423
   macro avg       0.55      0.56      0.55       423
weighted avg       0.69      0.66      0.67       423
```

In [73]:
```
# Define the model and its name together
fourth_model = {"model": KNeighborsClassifier(), "name": "K-Nearest Neighbors"}
```

```
# Call the method using the combined dictionary
classification_model.train_and_evaluate_models(
    X_train, y_train, X_test, y_test, fourth_model["model"], fourth_model["name"]
)
```

```
Model: K-Nearest Neighbors
Accuracy: 0.7257683215130024
Classification Report:
              precision    recall  f1-score   support

           0       0.80      0.87      0.83       328
           1       0.34      0.23      0.28        95

    accuracy                           0.73       423
   macro avg       0.57      0.55      0.55       423
weighted avg       0.69      0.73      0.71       423
```

In [74]:

```
# Train and evaluate SVC model
fifth_model = {"model": SVC(), "name": "Support Vector Machine"}

classification_model.train_and_evaluate_models(
    X_train, y_train, X_test, y_test, fifth_model["model"], fifth_model["name"]
)
```

```
Model: Support Vector Machine
Accuracy: 0.7754137115839244
Classification Report:
              precision    recall  f1-score   support

           0       0.78      1.00      0.87       328
           1       0.50      0.01      0.02        95

    accuracy                           0.78       423
   macro avg       0.64      0.50      0.45       423
weighted avg       0.71      0.78      0.68       423
```

In [75]:

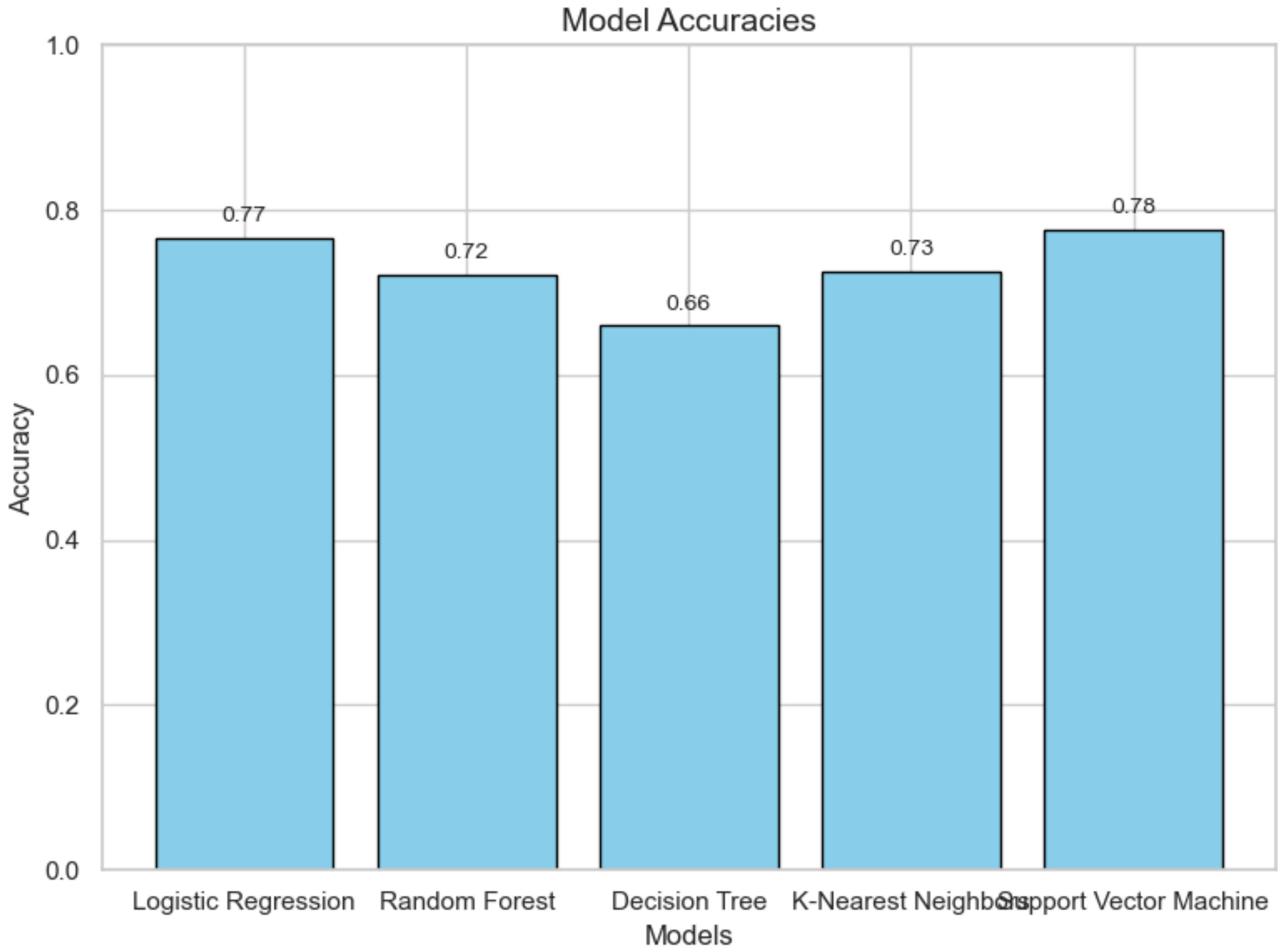
```
# Make predictions on the test set using a trained model.
y_pred = classification_model.predict_on_test_set(third_model["model"], X_test)

# Display predictions
print("Test Set Predictions:", y_pred)
```

```
Predictions on test set completed.
Test Set Predictions: [0 0 0 1 1 0 1 1 0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 1 0 1 1 1 1 0 0 1 0 1 0 1 0
 0 0 1 0 0 0 0 1 0 1 0 1 1 0 0 0 0 0 1 0 0 0 0 1 0 1 0 1 0 0 0 0 0 1 1 1 0
 0 1 1 1 1 0 0 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 1 0 0 1 0 0 1 0 0
 0 1 0 1 0 0 0 0 1 1 1 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 1 0 1 0 0 0 1 0 0
 1 0 0 1 1 0 1 0 0 0 0 0 0 1 0 0 0 1 1 1 0 0 0 1 0 1 0 0 0 0 0 0 1 0 1 1 0
 1 0 0 1 0 1 0 0 0 1 1 1 0 0 1 1 0 0 0 0 0 0 1 0 1 0 0 0 0 1 1 0 0 1 0 0 0
 1 1 0 0 0 1 0 1 0 0 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0
 0 0 1 0 1 0 0 0 0 1 0 0 0 0 0 0 0 1 1 1 1 0 1 0 0 1 0 0 1 0 0 0 0 0 0 1 1
 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0
 0 1 0 0 0 1 0 1 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 1 0 0 1 1 0 0 0 1 1 0 1
 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 1 1 0 0 1 0 1 0 0 1 0 0 0 1 0 0 0 1 0 1 1
 0 0 0 0 0 1 0 0 1 1 0 1 0 0 0 0]
```

In [76]:
```python
print("Actual labels:", y_test)
```

```
Actual labels: [0 0 1 0 0 0 1 0 0 1 0 0 0 1 0 0 1 1 1 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 1 0 0 0 0 0 1 0 0 1 0 0 0 0 1 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0
 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 1 0 1 0 0 0 1 0 1 0 0 0
 0 0 0 0 0 1 0 0 1 1 0 0 0 1 1 0 0 0 0 0 0 1 1 0 0 1 1 0 0 0 0 0 1 1 0 1 0
 1 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1
 1 0 0 1 0 0 0 0 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0
 0 1 0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0
 0 0 1 0 0 0 0 1 0 1 0 1 0 1 0 0 0 0 1 0 0 0 1 0 0 1 1 0 0 0 0 0 1 0 0 0 0
 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0
 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 1 0 1 0 1 1 0 0 0 0 0 0 0
 0 0 0 0 0 1 1 0 1 0 0 0 0 0 0 0]
```

In [77]:
```python
# Plot accuracies of the models
classification_model.plot_model_accuracies()
```



MODELLING

A classification pipeline was implemented to predict an individual's likelihood of smoking based on a dataset containing demographic and socioeconomic variables. A copy of the cleaned dataset was created for use, in order to avoid altering the original in case a different approach or encoding process different from what was done is to take place. This section process began with data preprocessing, where categorical columns were encoded numerically using a label encoder to ensure compatibility with machine learning models. The Target Column (SMOKE) was encoded as Yes = 1 and No = 0. To enhance model performance and reduce computational complexity, columns with low feature importance, identified through ANOVA F-tests, were excluded. The dataset was then divided into training and testing sets with a 75-25 ratio, ensuring consistency by converting both features and target variables into NumPy arrays. Feature scaling was performed using StandardScaler to standardise the feature space, optimising the model's performance on the scaled data.

According to Fatima (2024), supervised machine learning techniques, including K-Nearest Neighbours, Decision Trees, and Support Vector Machines, are effective for making predictions and classifying data into different groups. Similar to this, several machine learning models, including Logistic Regression, Random Forest, Decision Tree, K-Nearest Neighbors, and Support Vector Machine (SVM), were trained and evaluated in this analysis for developing predictive models for smoking behaviour. Each model was trained on the training set and evaluated using the test set. Accuracy, along with additional performance metrics, including precision, recall, and F1-score, was computed to assess the models' predictive effectiveness. The SVM model recorded the highest accuracy at 77.5%, making it the most dependable model for prediction in this dataset. It was followed by Logistic Regression, Random Forest, and K-Nearest Neighbors, with the Decision Tree model yielding the lowest performance.

Below, predictions were further validated with new data inputs that aligned with the training set's feature structure, confirming the models' ability to generalise. The results demonstrated the models' strengths and limitations, highlighting the importance of feature selection, scaling, and robust evaluation in classification tasks.

PREDICTING USING NEW DATA SETS

In [80]:
```
# New data for prediction, ensuring all columns are included
new_data = {
    'gender': ['Female', 'Male'],
    'age': [45, 57],
    'marital_status': ['Divorced', 'Single'],
    'highest_qualification': ['No Qualification', 'Degree'],
    'nationality': ['British', 'Scottish'],
    'gross_income': ['Under 2,600', '5,200 to 10,400'],
    'ethnicity': ['Asian', 'Caucasian'],  # Placeholder values for 'ethnicity'
    'region': ['North', 'South']  # Placeholder values for 'region'
}

# Convert new data into a DataFrame
new_data_df = pd.DataFrame(new_data)

# Encode the new data using the same method used for training data
new_data_encoded = classification_model.encode_categorical_columns(new_data_df)

# Align columns of the new data to match the training data
new_data_encoded = new_data_encoded[X.columns]  # Ensure the column order matches

# Using the best model
models = [
    (SVC(), "Support Vector Machine")
]

# Predicting process
predictions = {}
for model, model_name in models:
    model.fit(X, y)  # Fit model again
    prediction = model.predict(new_data_encoded)  # Ensure 'new_data_encoded' has the same format as training data
    predictions[model_name] = prediction

# Display the predictions
print(predictions)
```

```
Categorical columns encoded.
{'Support Vector Machine': array([0, 0])}
```

In [81]:
```
#Using the other models to get a prediction
models = [
    (LogisticRegression(), "Logistic Regression"),
    (RandomForestClassifier(), "Random Forest"),
    (DecisionTreeClassifier(), "Decision Tree"),
    (KNeighborsClassifier(), "K-Nearest Neighbors")
]

# Predicting process
predictions = {}
for model, model_name in models:
    model.fit(X, y)  # Fit model again
```

```
    prediction = model.predict(new_data_encoded)  # Ensure 'new_data_encoded' has the same format as training data
    predictions[model_name] = prediction

# Display the predictions
print(predictions)
```

```
{'Logistic Regression': array([0, 0]), 'Random Forest': array([0, 0]), 'Decision Tree': array([1, 0]), 'K-Nearest Neighbors': array([0, 0])}
```

In [ ]:

CONCLUSION

The analysis and machine learning modelling conducted on the smoking behaviour dataset provided comprehensive insights into the factors influencing smoking habits and the efficacy of predictive algorithms. Data cleaning and preprocessing formed the foundation of the study, addressing issues such as missing values in non-smoker-related columns and converting income ranges into numeric values for better analysis. Exploratory data analysis highlighted critical patterns, including gender and age distributions, income levels, and regional diversity, which informed the subsequent modelling process. Visualisations such as scatterplots, box plots, and Kernel Density Estimate (KDE) plots further highlighted relationships between demographic factors and smoking behaviour.

Feature selection was also vital in identifying the most influential predictors of smoking behaviour. Using the ANOVA F-test, features such as age, marital status, and gross income emerged as significant contributors to the prediction task. To streamline the dataset, less impactful features were dropped, thereby improving model performance. Several machine learning models were developed and evaluated for their predictive capabilities, with the Support Vector Machine (SVM) achieving the highest accuracy.

In conclusion, this analysis provided a structured approach to understanding smoking-related risk factors and predictive modelling. This project developed a model that can be implemented to predict the likelihood of an individual smoking, taking into account the essential features. It also demonstrated the integration of data preprocessing, feature selection, and machine learning to derive actionable insights, offering a foundation for further studies and real-world applications in public health and behaviour prediction.

In [ ]:

REFERENCES

Fatima, S. (2024). Predictive Models For Early Detection Of Chronic Diseases Like Cancer. [online] International Journal of Engineering Research & Management Technology. doi: https://doi.org/10.13140/RG.2.2.13988.28809.

Groot, P.M. de , Wu, C.C., Carter, B.W. and Munden, R.F. (2018). The epidemiology of lung cancer. Translational Lung Cancer Research, [online] 7(3), pp.220–233. doi: https://doi.org/10.21037/tlcr.2018.05.06.

Leiter, A., Veluswamy, R.R. and Wisnivesky, J.P. (2023). The global burden of lung cancer: current status and future trends. Nature Reviews Clinical Oncology, [online] 20(20), pp.1–16. doi: https://doi.org/10.1038/s41571-023-00798-3.

Zambom, A. and Dias, R. (2012). A Review of Kernel Density Estimation with Applications to Econometrics 20 A Review of Kernel Density Estimation with Applications to Econometrics. International Econometric Review (IER), [online] 5(1), pp.20–42. Available at: https://www.econstor.eu/bitstream/10419/238805/1/ier-v05-i1-p020-042.pdf [Accessed 16 Jan. 2025].