Group Information

1. MUHAMMAD ARIF BIN MOHAMAD AMIR 2022206726
2. NOOR ARMALISA BINTI ZURAIMI 2022830378
3. RAHIMAH HAZIRAH BINTI ROSLI 2022424226

Methodlogy and Work Division

Methodlogy

For this project, the methodology that we used is the CRISP-DM (Cross-Industry Standard Process for Data Mining). Below is the short explaination for each phase in the CRISP-DM:

1) Business Understanding:

- Define the objective: Determine the goal of the project, such as predicting diabetes occurrence based on certain features.

2) Data Understanding:

- Obtain the diabetes dataset with relevant features (e.g., age, BMI, cholesterol levels, smoking habits).
- Analyze the dataset to understand its structure, check for missing values, outliers, and data distributions.
- Identify 'Diabetes_binary' as the target variable to predict.

3) Data Preparation:

- Handle missing values and outliers using appropriate techniques
- Choose relevant features for modeling, considering domain knowledge

4) Modeling:

- Split data: Divide the preprocessed data into training and testing sets (e.g., 70% training, 30% testing).
- Model selection: Choose KNN, DT, and NB classifiers as the models to predict diabetes.
- Model training: Train each classifier using the training data.

5) Evaluation:

- Evaluate the accuracy, precision, recall, and F1-score of each model on the testing data.
- Determine the model with the highest overall performance for diabetes prediction.

6) Deployment:

- Summarize the findings, including the best model's performance and key insights in report.
- Create informative visualizations to present the results effectively in the report.
- Deploy the model using Flask.

Work Division

1. EDA (Lisa, Arif, Rahimah)
2. Data Preparation (Arif, Lisa)

3. Machine Learning Model (Arif, Lisa, Rahimah)
4. Deploy Model (Arif)
5. Report (Arif, Lisa, Rahimah)

1) Import Library

To start running this jupyter file, make sure you install the required library by running below command:

pip install pandas
pip install matplotlib
pip install numpy
pip install scikit-learn
pip install seaborn
pip install plotly
pip install chart-studio

In [1]:

```
#Import Library

import pandas as pd
import matplotlib
import numpy as np
import sklearn
import seaborn as sns


#graph & plot library
import plotly.graph_objects as go
import plotly.offline as pyoff
import chart_studio.plotly as py
import matplotlib.pyplot as plt
import plotly.express as px

from sklearn.utils import resample
from sklearn.preprocessing import StandardScaler, LabelEncoder

print("Pandas version:", pd.__version__)
print("Matplotlib version:", matplotlib.__version__)
print("NumPy version:", np.__version__)
print("Scikit-learn version:", sklearn.__version__)
print("Seaborn version:", sns.__version__)
```

```
Pandas version: 2.0.3
Matplotlib version: 3.7.1
NumPy version: 1.25.0
Scikit-learn version: 1.3.0
Seaborn version: 0.12.2
```

2) Read Dataset

In [2]:

```
# Read CSV file
df = pd.read_csv('diabetes.csv')
df.head()
```

Out[2]:

	Diabetes_binary	HighBP	HighChol	CholCheck	BMI	Smoker	Stroke	HeartDiseaseorAttack	Phy
0	0	1	1	1	40.0	1.0	0	0	
1	0	0	0	0	25.0	1.0	0	0	
2	0	1	1	1	28.0	0.0	0	0	
3	0	1	0	1	27.0	0.0	0	0	
4	0	1	1	1	24.0	0.0	0	0	

5 rows × 22 columns

Explanation
The code reads a CSV file named 'diabetes_2_missing_values.csv' into a pandas DataFrame called 'df' and then displays the first few rows of the DataFrame using the 'head()' function.

3) Explotary Data Analysis (EDA)

3.1) Check Data Structure

In [3]:

```
# Checking Data structure
df.dtypes
```

Out[3]:

```
Diabetes_binary           int64
HighBP                    int64
HighChol                  int64
CholCheck                 int64
BMI                     float64
Smoker                  float64
Stroke                    int64
HeartDiseaseorAttack      int64
PhysActivity              int64
Fruits                    int64
Veggies                   int64
HvyAlcoholConsump         int64
AnyHealthcare             int64
NoDocbcCost               int64
GenHlth                   int64
MentHlth                  int64
PhysHlth                  int64
DiffWalk                  int64
Sex                       int64
Age                     float64
Education                 int64
Income                    int64
dtype: object
```

Explanation

df.dtypes is to check the data types of the columns in the DataFrame 'df'. When a DataFrame is created from a CSV file, pandas automatically infers the data types for each column based on the content of the CSV file. This information is useful to understand how pandas has interpreted the data and to ensure that the data types are appropriate for further analysis or processing.

In [4]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 253680 entries, 0 to 253679
Data columns (total 22 columns):
 #   Column                Non-Null Count   Dtype  
---  ------                --------------   -----  
 0   Diabetes_binary       253680 non-null  int64  
 1   HighBP                253680 non-null  int64  
 2   HighChol              253680 non-null  int64  
 3   CholCheck             253680 non-null  int64  
 4   BMI                   253654 non-null  float64
 5   Smoker                253597 non-null  float64
 6   Stroke                253680 non-null  int64  
 7   HeartDiseaseorAttack  253680 non-null  int64  
 8   PhysActivity          253680 non-null  int64  
 9   Fruits                253680 non-null  int64  
 10  Veggies               253680 non-null  int64  
 11  HvyAlcoholConsump     253680 non-null  int64  
 12  AnyHealthcare         253680 non-null  int64  
 13  NoDocbcCost           253680 non-null  int64  
 14  GenHlth               253680 non-null  int64  
 15  MentHlth              253680 non-null  int64  
 16  PhysHlth              253680 non-null  int64  
 17  DiffWalk              253680 non-null  int64  
 18  Sex                   253680 non-null  int64  
 19  Age                   253485 non-null  float64
 20  Education             253680 non-null  int64  
 21  Income                253680 non-null  int64  
dtypes: float64(3), int64(19)
memory usage: 42.6 MB
```

Explaination

df.info() provides a summary of the DataFrame 'df' by showing the column names, non-null counts, and data types of each column. It helps to understand the data's structure, identify missing values, and get an overview of the data types in the DataFrame.

In [5]:

```
df.isnull().sum().sort_values(ascending=False)
```

Out[5]:

```
Age                     195
Smoker                   83
BMI                      26
Diabetes_binary           0
AnyHealthcare             0
Education                 0
Sex                       0
DiffWalk                  0
PhysHlth                  0
MentHlth                  0
GenHlth                   0
NoDocbcCost               0
HvyAlcoholConsump         0
HighBP                    0
Veggies                   0
Fruits                    0
PhysActivity              0
HeartDiseaseorAttack      0
Stroke                    0
CholCheck                 0
HighChol                  0
Income                    0
dtype: int64
```

Explanation

df.isnull().sum().sort_values(ascending=False) calculates the number of missing values (null values) in each column of the DataFrame 'df' and then sorts them in descending order based on the count of missing values. This allows you to identify the columns with the highest number of missing values at the top of the list.

In [6]:

```
df.describe()
```

Out[6]:

	Diabetes_binary	HighBP	HighChol	CholCheck	BMI	Smoker
count	253680.000000	253680.000000	253680.000000	253680.000000	253654.000000	253597.000000
mean	0.139333	0.429001	0.424121	0.962670	28.382352	0.443140
std	0.346294	0.494934	0.494210	0.189571	6.608865	0.496757
min	0.000000	0.000000	0.000000	0.000000	12.000000	0.000000
25%	0.000000	0.000000	0.000000	1.000000	24.000000	0.000000
50%	0.000000	0.000000	0.000000	1.000000	27.000000	0.000000
75%	0.000000	1.000000	1.000000	1.000000	31.000000	1.000000
max	1.000000	1.000000	1.000000	1.000000	98.000000	1.000000

8 rows × 22 columns

Explanation

The code ***df.describe()*** provides a summary of descriptive statistics for the numerical columns in the DataFrame 'df'. It includes statistics such as count, mean, standard deviation, minimum, 25th percentile (Q1), median (50th percentile or Q2), 75th percentile (Q3), and maximum for each numerical column. This summary helps in understanding the distribution and central tendency of the data in the DataFrame.

3.2) Data Visualization (Categorical Data)

Explantion

In this part, we are going to visualize each columns using bar plot. Below codes creates a horizontal bar plot using Plotly to visualize the number of people with and without diabetes based on the specify column (e.g. Income, Smoker, Education, Age, etc).

The DataFrame ***'df_copy'*** is first prepared by setting the target column as the index and converting the 'Diabetes_binary' column to numeric. The data is then grouped by target column and 'Diabetes_binary', and the counts are calculated.

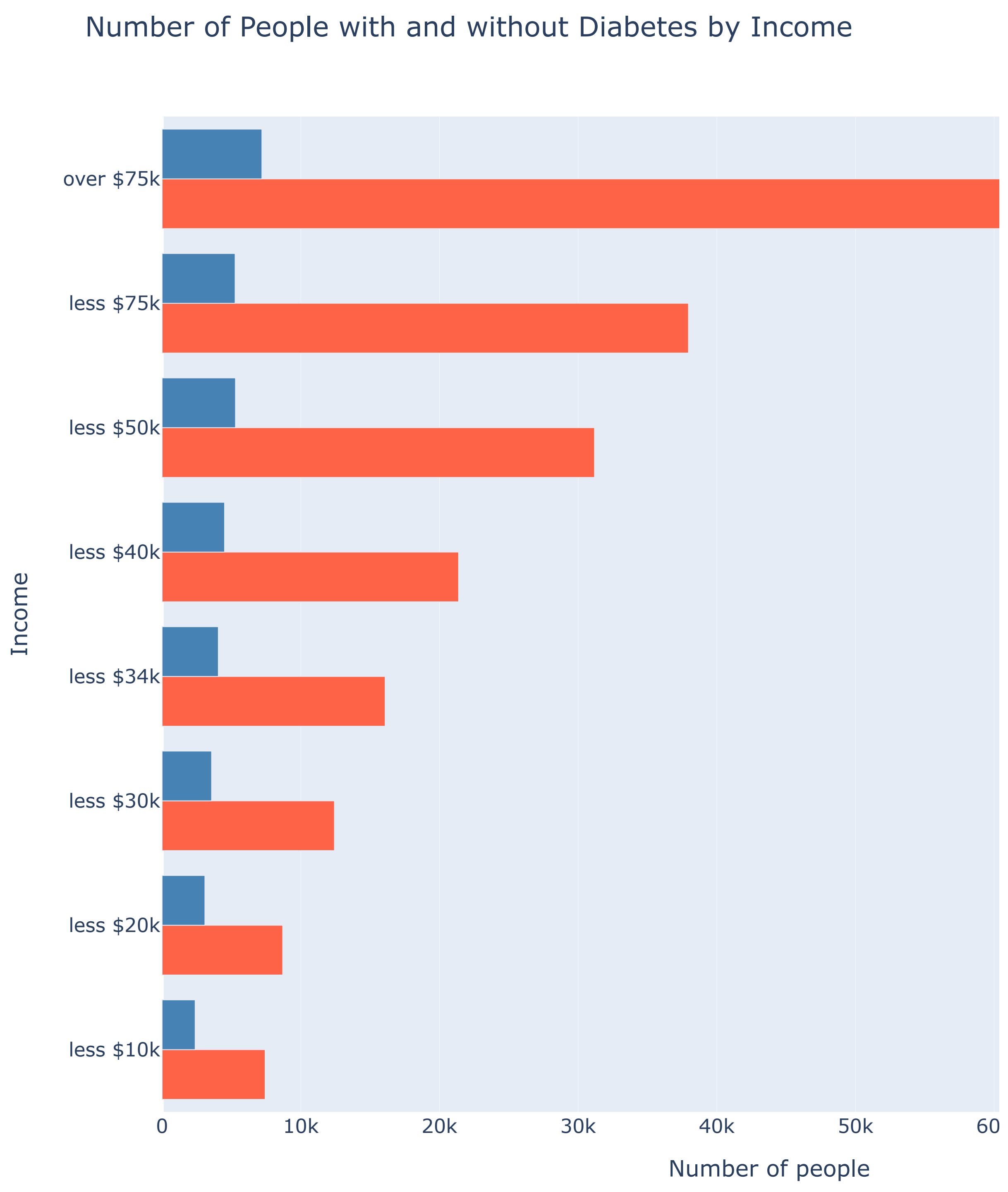
Two bar traces are added to the figure, one for people without diabetes and another for people with diabetes. Each bar represents the number from each column. The bars are color-coded, with 'tomato' representing no diabetes and 'steelblue' representing diabetes cases.

The layout is updated with relevant titles and labels for axes, and the plot is displayed using Plotly's 'show' method. This plot allows for a quick comparison of diabetes prevalence for different columns.

3.2.1) Visualization for Income

In [7]:

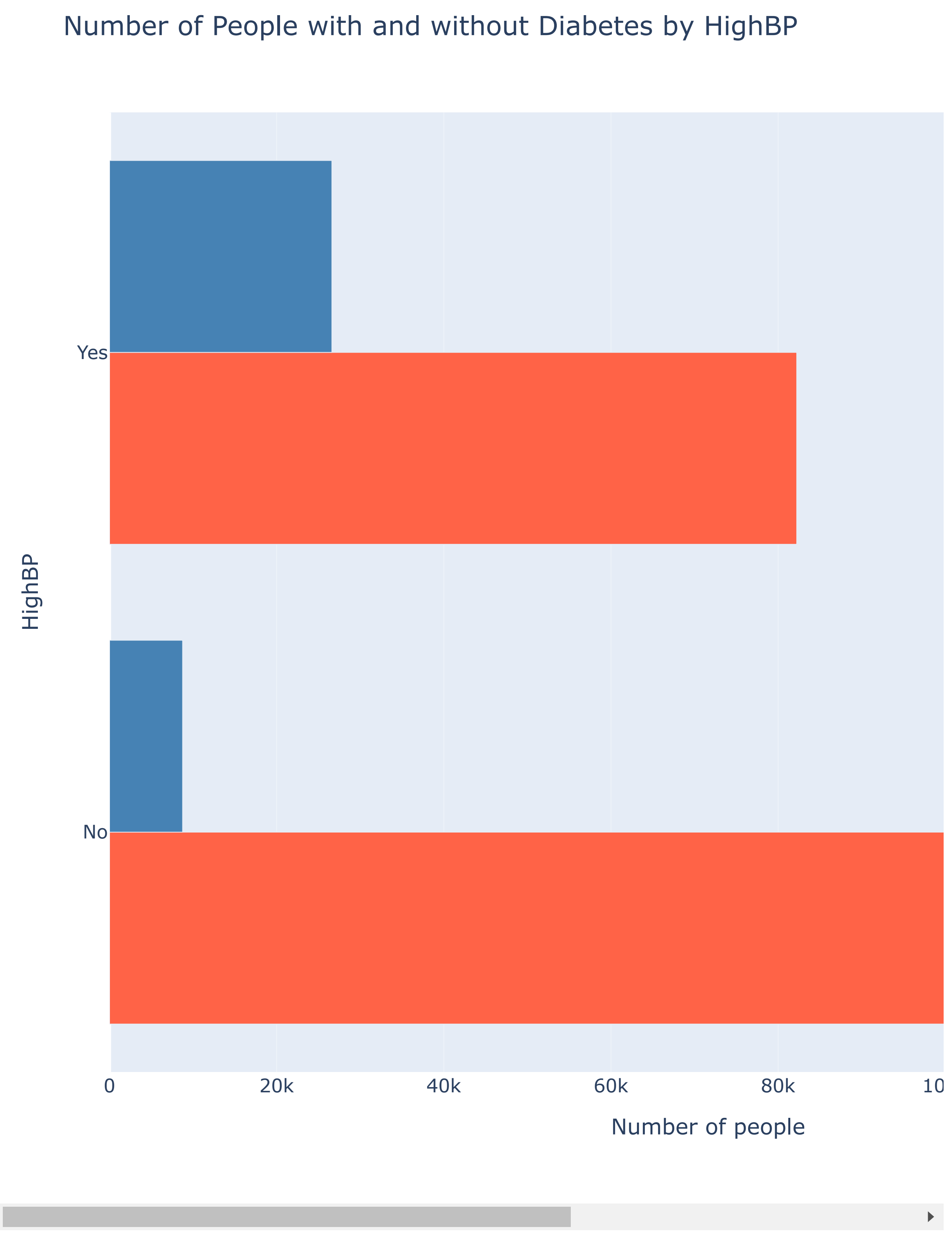
```
# Make a copy of the DataFrame
df_copy = df.copy()

# Set the 'Income' column as the index
df_copy.set_index('Income', inplace=True)

# Select the 'Diabetes_binary' column and convert it to numeric if needed
df_copy['Diabetes_binary'] = pd.to_numeric(df_copy['Diabetes_binary'], errors='coerce')

# Group the data by 'Diabetes_binary' and 'Income' columns and calculate counts
grouped = df_copy.groupby(['Income', 'Diabetes_binary']).size().unstack().fillna(0)

# Create the figure using plotly.graph_objects
fig = go.Figure()

# Add the 'No Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[0],
    orientation='h',
    name='No Diabetes',
    marker=dict(color='tomato')
))

# Add the 'Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[1],
    orientation='h',
    name='Diabetes',
    marker=dict(color='steelblue')
))

# Update the layout
fig.update_layout(
    title='Number of People with and without Diabetes by Income',
    xaxis_title='Number of people',
    yaxis=dict(
        title='Income',
        tickmode='array',
        tickvals=[1, 2, 3, 4, 5, 6, 7, 8],
        ticktext=["less $10k", 'less $20k', 'less $30k', 'less $34k', 'less $40k', 'less $50k'
    ),
    legend_title='Diabetes',
    width=1000,
    height=800
)
# # Define the custom y-axis labels
# y_labels = []

# # Update the y-axis labels
# fig.update_yaxes(ticktext=y_labels, tickvals=grouped.index, tickangle=90)

# Show the plot using plotly.offline
fig.show()
```


Number of People with and without Diabetes by Income
Income
over $75k
less $75k
less $50k
less $40k
less $34k
less $30k
less $20k
less $10k
0
10k
20k
30k
40k
50k
60
Number of people

3.2.2) Visualization for HighBP

In [8]:

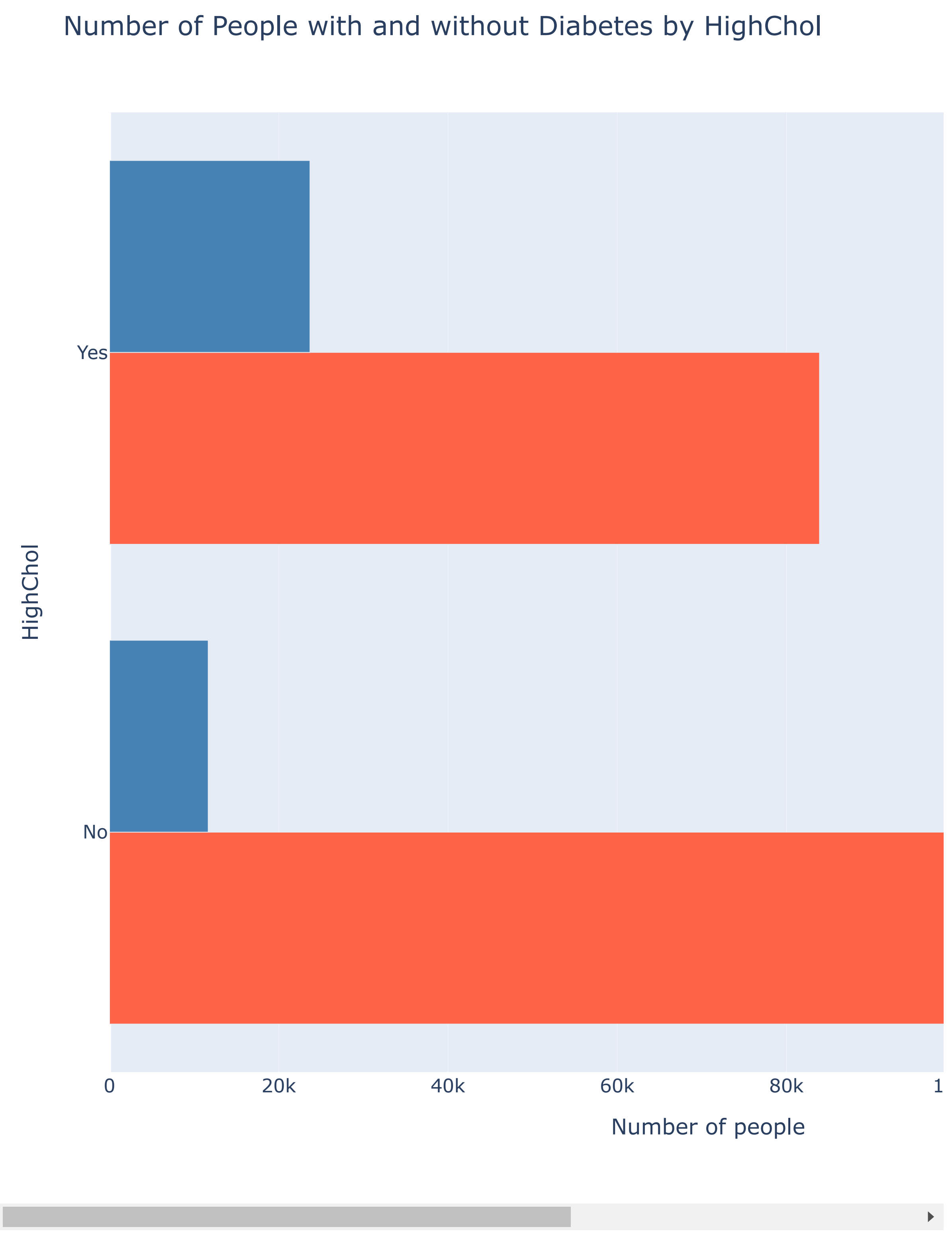
```
# Make a copy of the DataFrame
df_copy = df.copy()

# Set the 'HighBP' column as the index
df_copy.set_index('HighBP', inplace=True)

# Select the 'Diabetes_binary' column and convert it to numeric if needed
df_copy['Diabetes_binary'] = pd.to_numeric(df_copy['Diabetes_binary'], errors='coerce')

# Group the data by 'Diabetes_binary' and 'HighBP' columns and calculate counts
grouped = df_copy.groupby(['HighBP', 'Diabetes_binary']).size().unstack().fillna(0)

# Create the figure using plotly.graph_objects
fig = go.Figure()

# Add the 'No Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[0],
    orientation='h',
    name='No Diabetes',
    marker=dict(color='tomato')
))

# Add the 'Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[1],
    orientation='h',
    name='Diabetes',
    marker=dict(color='steelblue')
))

# Update the layout
fig.update_layout(
    title='Number of People with and without Diabetes by HighBP',
    xaxis_title='Number of people',
    yaxis=dict(
        title='HighBP',
        tickmode='array',
        tickvals=[0, 1],
        ticktext=['No', 'Yes']
    ),
    legend_title='Diabetes',
    width=1000,
    height=800
)


# Show the plot using plotly.offline
fig.show()
```


Number of People with and without Diabetes by HighBP
HighBP
Yes
No
0
20k
40k
60k
80k
10
Number of people

3.2.3) Visualization for HighChol

In [9]:

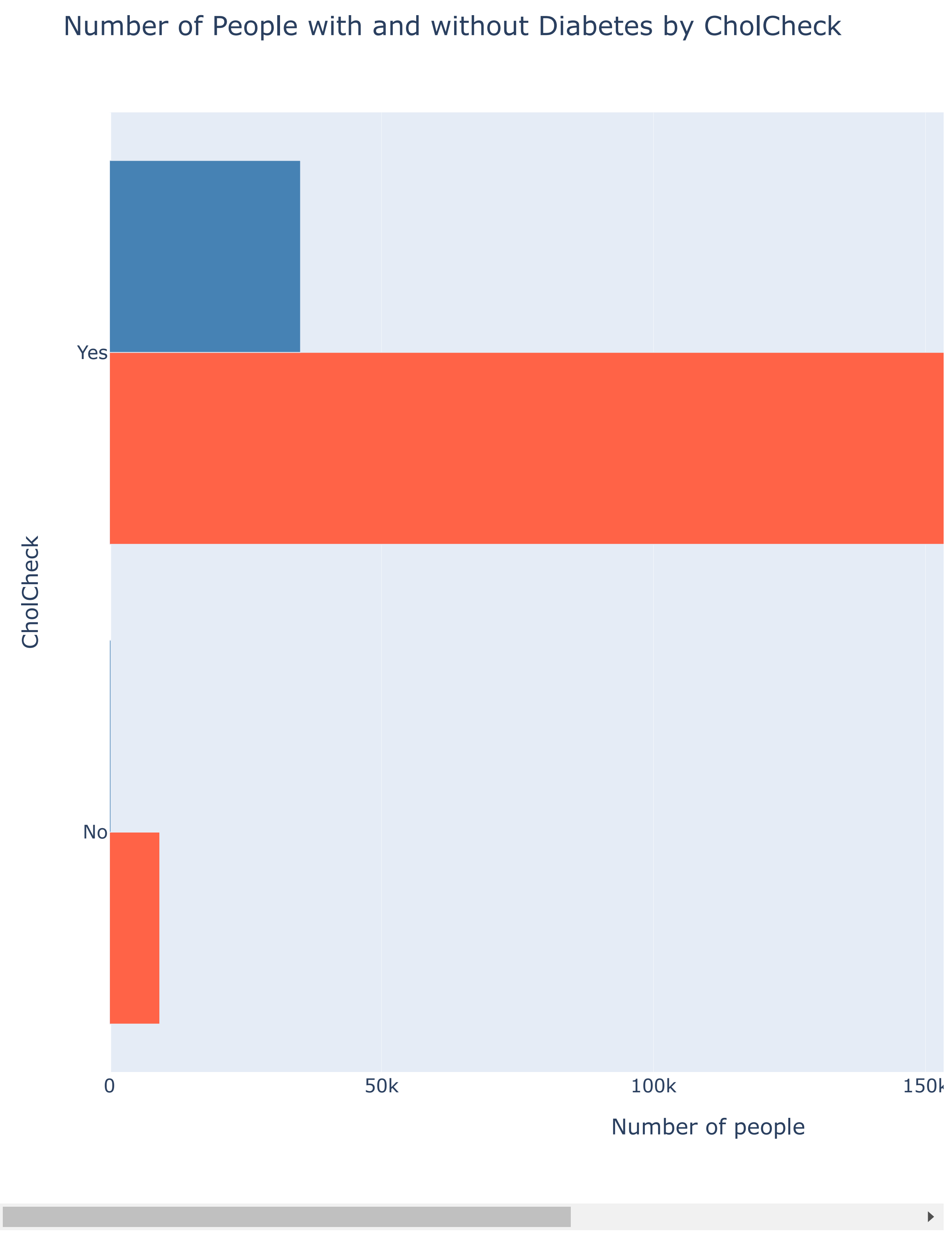
```
# Make a copy of the DataFrame
df_copy = df.copy()

# Set the 'HighChol' column as the index
df_copy.set_index('HighChol', inplace=True)

# Select the 'Diabetes_binary' column and convert it to numeric if needed
df_copy['Diabetes_binary'] = pd.to_numeric(df['Diabetes_binary'], errors='coerce')

# Group the data by 'Diabetes_binary' and 'HighChol' columns and calculate counts
grouped = df.groupby(['HighChol', 'Diabetes_binary']).size().unstack().fillna(0)

# Create the figure using plotly.graph_objects
fig = go.Figure()

# Add the 'No Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[0],
    orientation='h',
    name='No Diabetes',
    marker=dict(color='tomato')
))

# Add the 'Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[1],
    orientation='h',
    name='Diabetes',
    marker=dict(color='steelblue')
))

# Update the layout
fig.update_layout(
    title='Number of People with and without Diabetes by HighChol',
    xaxis_title='Number of people',
    yaxis=dict(
        title='HighChol',
        tickmode='array',
        tickvals=[0, 1],
        ticktext=['No', 'Yes']
    ),
    legend_title='Diabetes',
    width=1000,
    height=800
)


# Show the plot using plotly.offline
fig.show()
```


Number of People with and without Diabetes by HighChol
Yes
No
HighChol
0
20k
40k
60k
80k
1
Number of people

3.2.4) Visualization for CholCheck

In [10]:

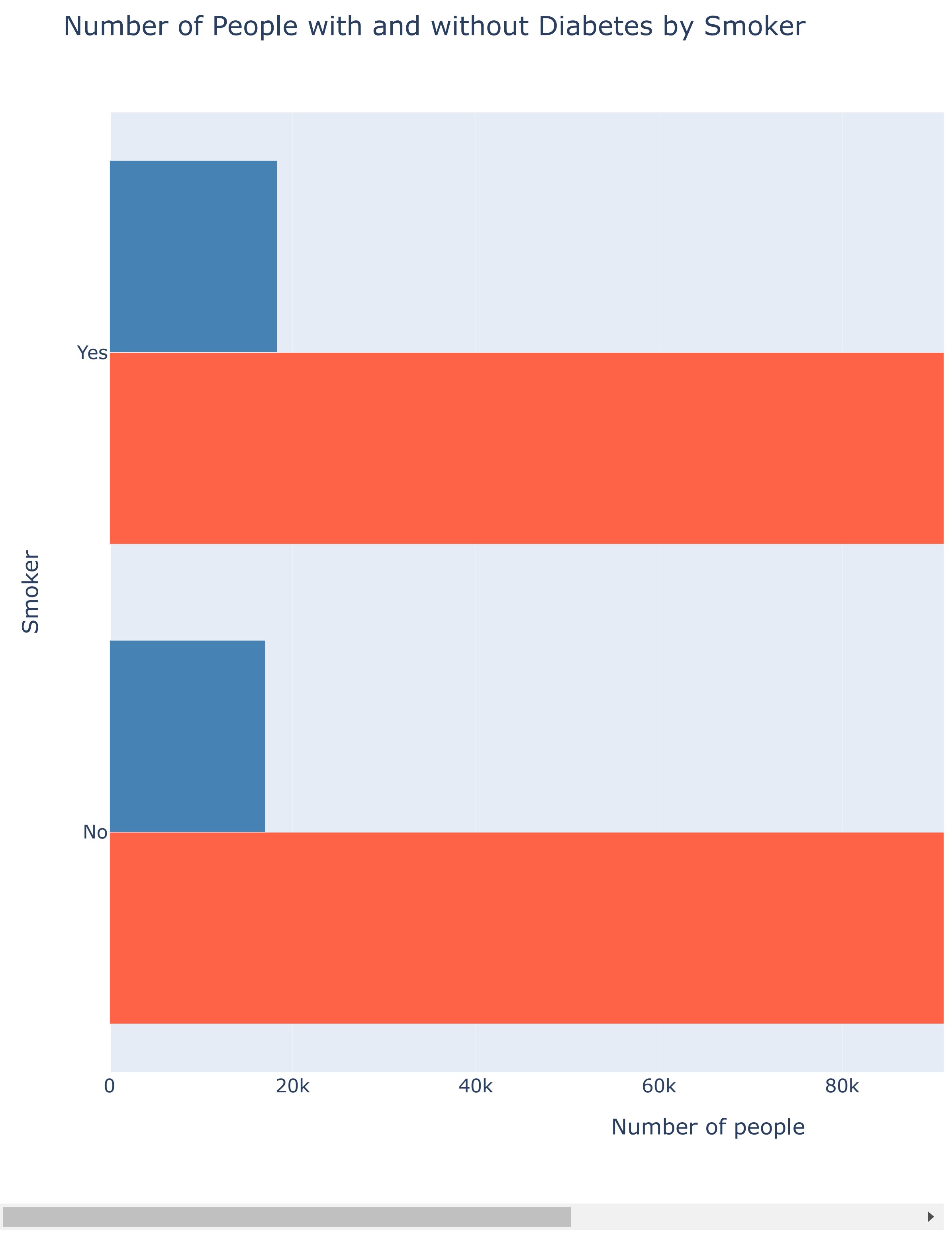
```
# Make a copy of the DataFrame
df_copy = df.copy()

# Set the 'CholCheck' column as the index
df_copy.set_index('CholCheck', inplace=True)

# Select the 'Diabetes_binary' column and convert it to numeric if needed
df_copy['Diabetes_binary'] = pd.to_numeric(df['Diabetes_binary'], errors='coerce')

# Group the data by 'Diabetes_binary' and 'CholCheck' columns and calculate counts
grouped = df.groupby(['CholCheck', 'Diabetes_binary']).size().unstack().fillna(0)

# Create the figure using plotly.graph_objects
fig = go.Figure()

# Add the 'No Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[0],
    orientation='h',
    name='No Diabetes',
    marker=dict(color='tomato')
))

# Add the 'Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[1],
    orientation='h',
    name='Diabetes',
    marker=dict(color='steelblue')
))

# Update the layout
fig.update_layout(
    title='Number of People with and without Diabetes by CholCheck',
    xaxis_title='Number of people',
    yaxis=dict(
        title='CholCheck',
        tickmode='array',
        tickvals=[0, 1],
        ticktext=['No', 'Yes']
    ),
    legend_title='Diabetes',
    width=1000,
    height=800
)


# Show the plot using plotly.offline
fig.show()
```


Number of People with and without Diabetes by CholCheck
CholCheck
Yes
No
0
50k
100k
150k
Number of people

3.2.5) Visualization for Smoker

In [11]:

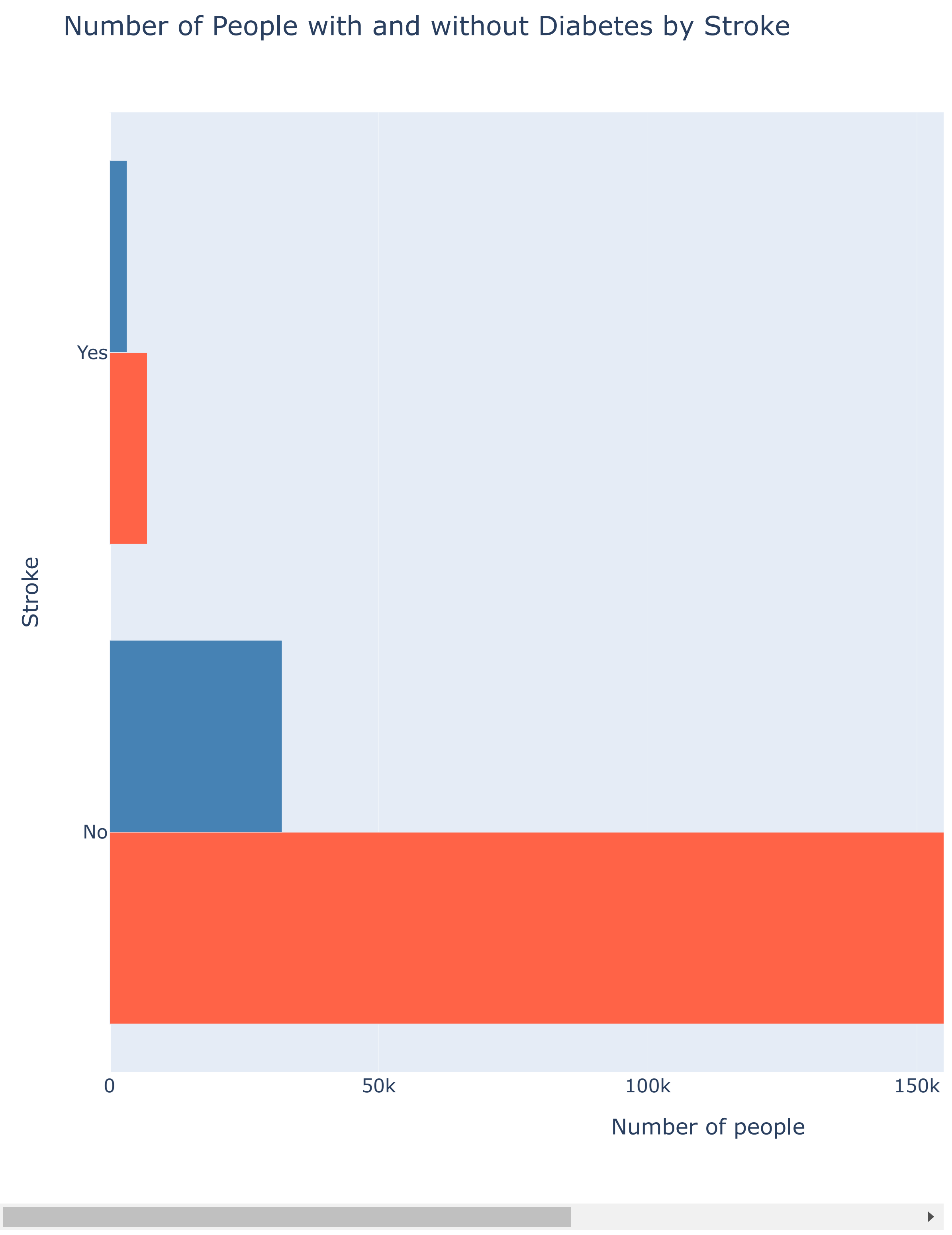
```
# Make a copy of the DataFrame
df_copy = df.copy()

# Select the 'Diabetes_binary' column and convert it to numeric if needed
df_copy['Diabetes_binary'] = pd.to_numeric(df_copy['Diabetes_binary'], errors='coerce')

# Group the data by 'Diabetes_binary' and 'Smoker' columns and calculate counts
grouped = df.groupby(['Smoker', 'Diabetes_binary']).size().unstack().fillna(0)

# Create the figure using plotly.graph_objects
fig = go.Figure()

# Add the 'No Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[0],
    orientation='h',
    name='No Diabetes',
    marker=dict(color='tomato')
))

# Add the 'Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[1],
    orientation='h',
    name='Diabetes',
    marker=dict(color='steelblue')
))

# Update the layout
fig.update_layout(
    title='Number of People with and without Diabetes by Smoker',
    xaxis_title='Number of people',
    yaxis=dict(
        title='Smoker',
        tickmode='array',
        tickvals=[0, 1],
        ticktext=['No', 'Yes']
    ),
    legend_title='Diabetes',
    width=1000,
    height=800
)


# Show the plot using plotly.offline
fig.show()
```


Number of People with and without Diabetes by Smoker
Smoker
Yes
No
0
20k
40k
60k
80k
Number of people

3.2.6) Visualization for Stroke

In [12]:

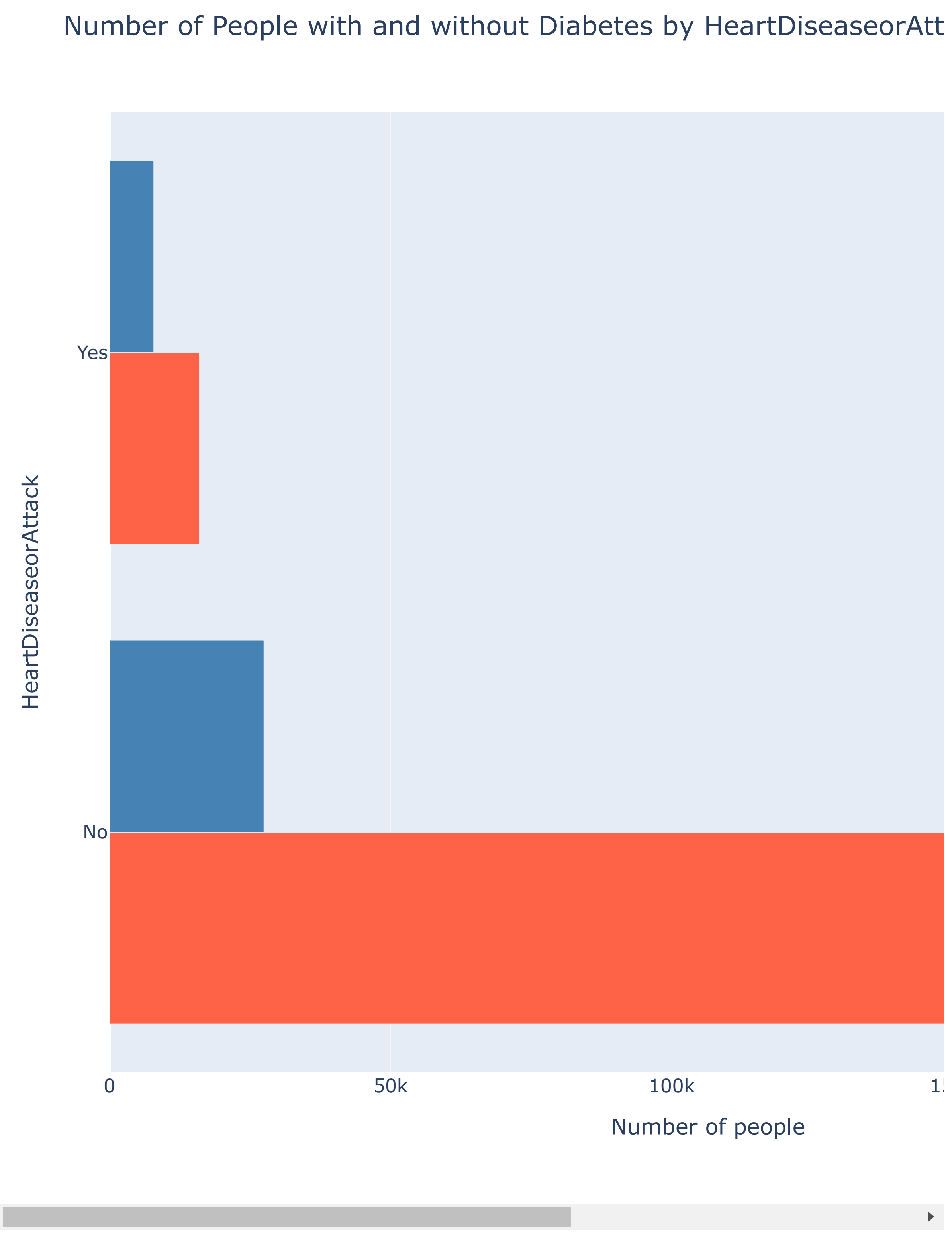
```
# Make a copy of the DataFrame
df_copy = df.copy()

# Set the 'Stroke' column as the index
df_copy.set_index('Stroke', inplace=True)

# Select the 'Diabetes_binary' column and convert it to numeric if needed
df_copy['Diabetes_binary'] = pd.to_numeric(df_copy['Diabetes_binary'], errors='coerce')

# Group the data by 'Diabetes_binary' and 'Stroke' columns and calculate counts
grouped = df.groupby(['Stroke', 'Diabetes_binary']).size().unstack().fillna(0)

# Create the figure using plotly.graph_objects
fig = go.Figure()

# Add the 'No Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[0],
    orientation='h',
    name='No Diabetes',
    marker=dict(color='tomato')
))

# Add the 'Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[1],
    orientation='h',
    name='Diabetes',
    marker=dict(color='steelblue')
))

# Update the layout
fig.update_layout(
    title='Number of People with and without Diabetes by Stroke',
    xaxis_title='Number of people',
    yaxis=dict(
        title='Stroke',
        tickmode='array',
        tickvals=[0, 1],
        ticktext=['No', 'Yes']
    ),
    legend_title='Diabetes',
    width=1000,
    height=800
)


# Show the plot using plotly.offline
fig.show()
```


Number of People with and without Diabetes by Stroke
Stroke
Yes
No
0
50k
100k
150k
Number of people

3.2.7) Visualization for HeartDiseaseorAttack

In [13]:

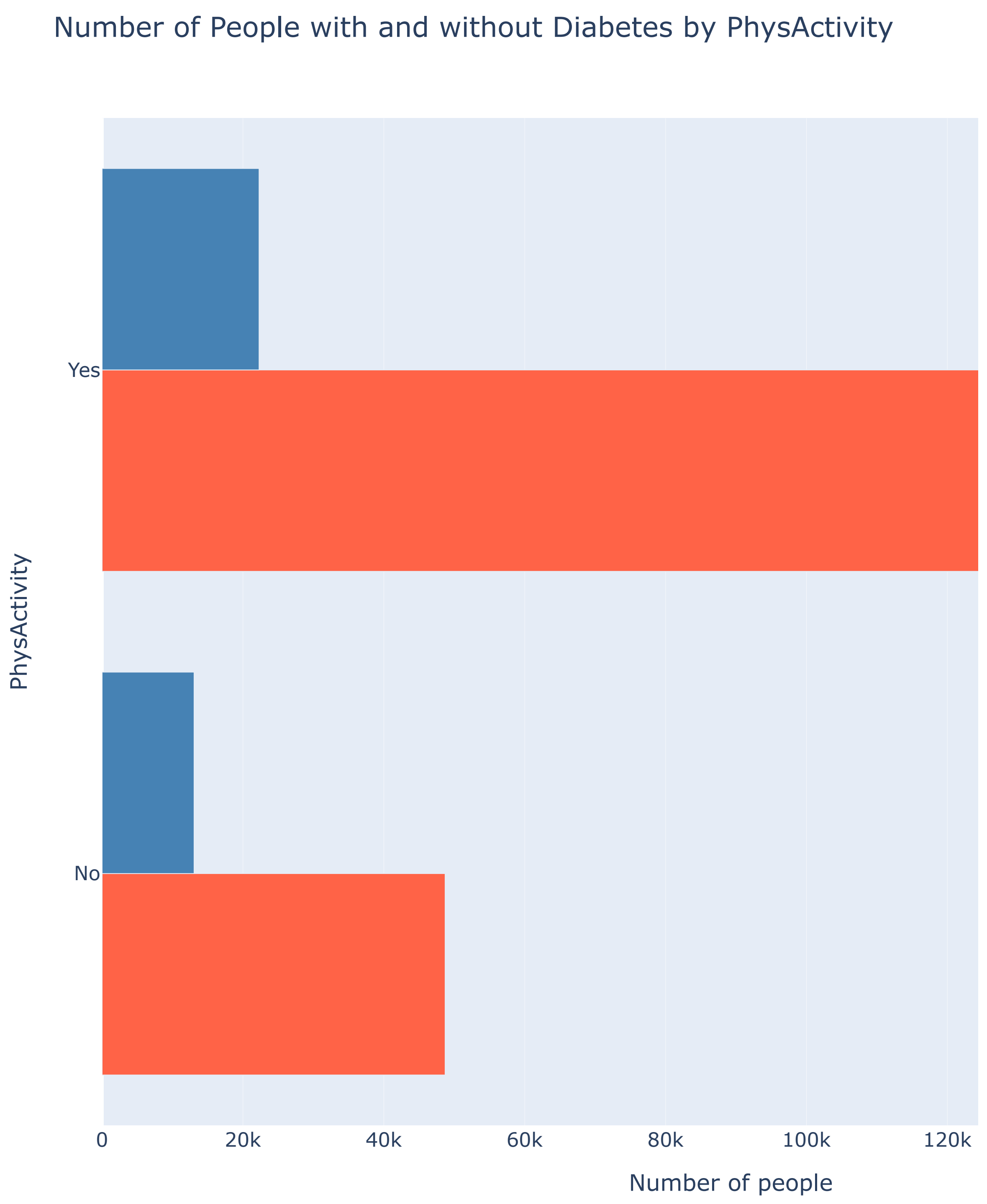
```
# Make a copy of the DataFrame
df_copy = df.copy()

# Set the 'HeartDiseaseorAttack' column as the index
df_copy.set_index('HeartDiseaseorAttack', inplace=True)

# Select the 'Diabetes_binary' column and convert it to numeric if needed
df_copy['Diabetes_binary'] = pd.to_numeric(df_copy['Diabetes_binary'], errors='coerce')

# Group the data by 'Diabetes_binary' and 'HeartDiseaseorAttack' columns and calculate counts
grouped = df.groupby(['HeartDiseaseorAttack', 'Diabetes_binary']).size().unstack().fillna(0)

# Create the figure using plotly.graph_objects
fig = go.Figure()

# Add the 'No Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[0],
    orientation='h',
    name='No Diabetes',
    marker=dict(color='tomato')
))

# Add the 'Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[1],
    orientation='h',
    name='Diabetes',
    marker=dict(color='steelblue')
))

# Update the layout
fig.update_layout(
    title='Number of People with and without Diabetes by HeartDiseaseorAttack',
    xaxis_title='Number of people',
    yaxis=dict(
        title='HeartDiseaseorAttack',
        tickmode='array',
        tickvals=[0, 1],
        ticktext=['No', 'Yes']
    ),
    legend_title='Diabetes',
    width=1000,
    height=800
)


# Show the plot using plotly.offline
fig.show()
```


Number of People with and without Diabetes by HeartDiseaseorAtt
HeartDiseaseorAttack
Yes
No
0
50k
100k
1
Number of people

3.2.8) Visualization for PhysActivity

In [14]:

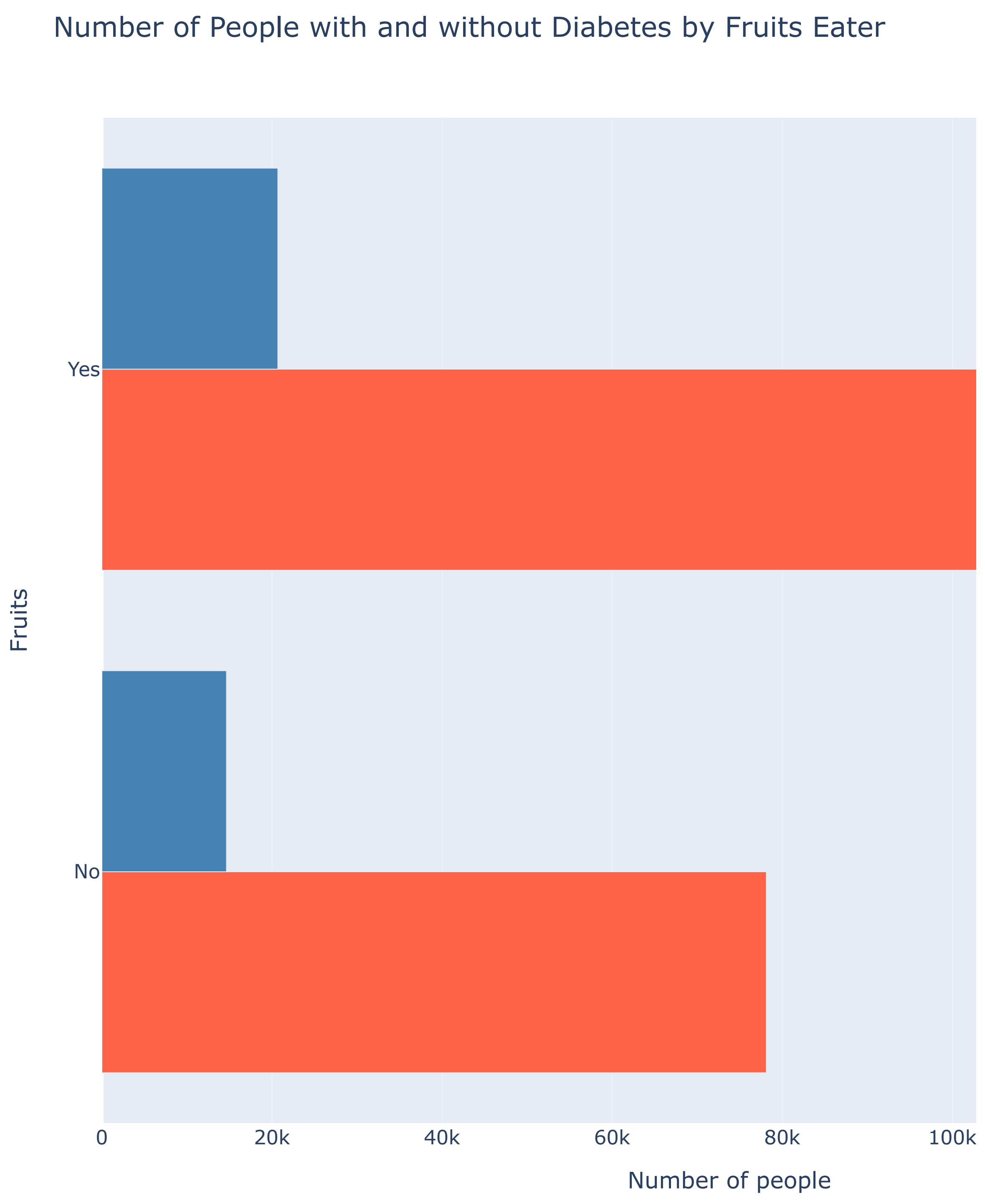
```
# Make a copy of the DataFrame
df_copy = df.copy()

# Set the 'PhysActivity' column as the index
df_copy.set_index('PhysActivity', inplace=True)

# Select the 'Diabetes_binary' column and convert it to numeric if needed
df_copy['Diabetes_binary'] = pd.to_numeric(df_copy['Diabetes_binary'], errors='coerce')

# Group the data by 'Diabetes_binary' and 'PhysActivity' columns and calculate counts
grouped = df.groupby(['PhysActivity', 'Diabetes_binary']).size().unstack().fillna(0)

# Create the figure using plotly.graph_objects
fig = go.Figure()

# Add the 'No Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[0],
    orientation='h',
    name='No Diabetes',
    marker=dict(color='tomato')
))

# Add the 'Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[1],
    orientation='h',
    name='Diabetes',
    marker=dict(color='steelblue')
))

# Update the layout
fig.update_layout(
    title='Number of People with and without Diabetes by PhysActivity',
    xaxis_title='Number of people',
    yaxis=dict(
        title='PhysActivity',
        tickmode='array',
        tickvals=[0, 1],
        ticktext=['No', 'Yes']
    ),
    legend_title='Diabetes',
    width=1000,
    height=800
)


# Show the plot using plotly.offline
fig.show()
```


Number of People with and without Diabetes by PhysActivity
PhysActivity
Yes
No
0
20k
40k
60k
80k
100k
120k
Number of people

3.2.9) Visualization for Fruits

In [15]:

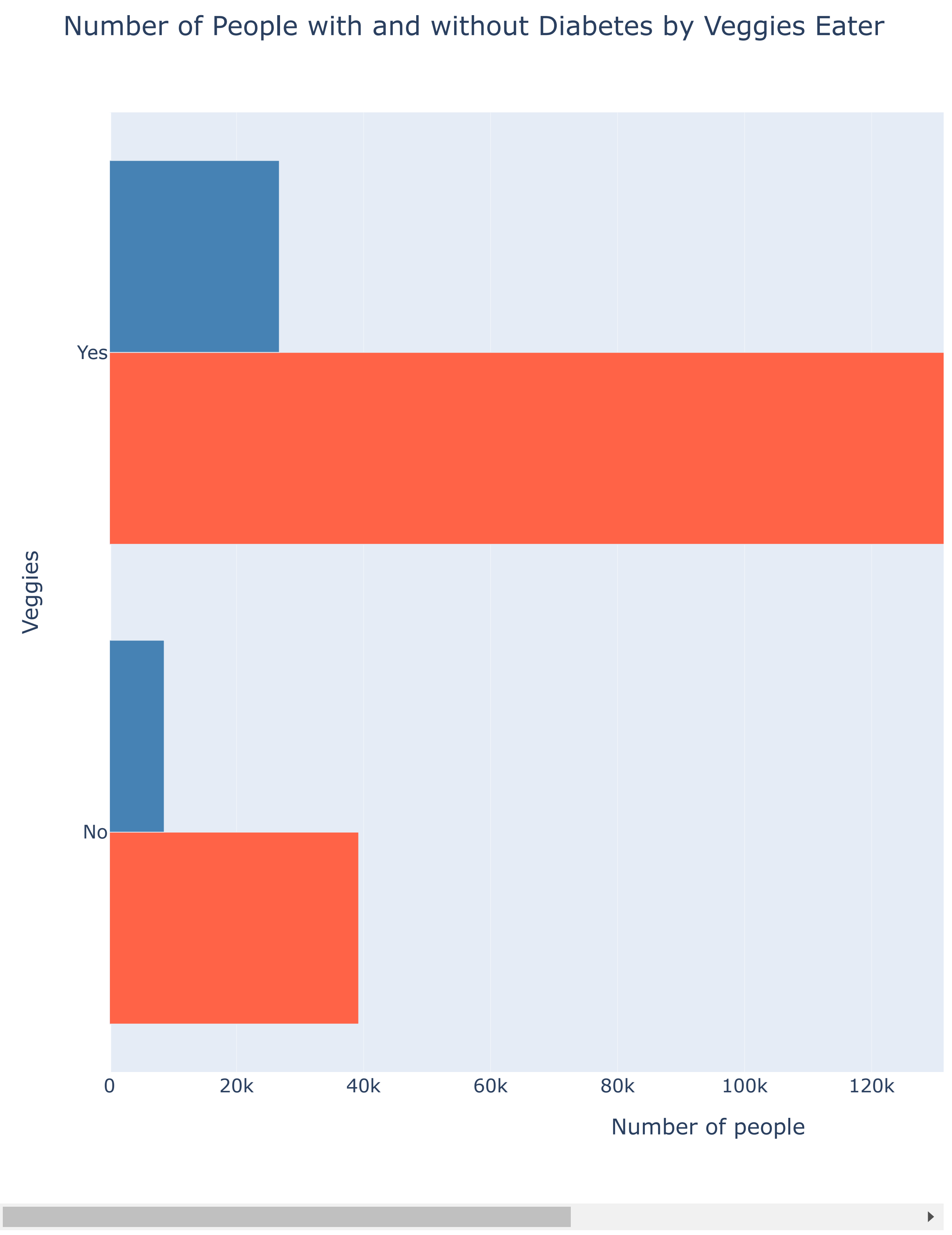
```
# Make a copy of the DataFrame
df_copy = df.copy()

# Set the 'Fruits' column as the index
df_copy.set_index('Fruits', inplace=True)

# Select the 'Diabetes_binary' column and convert it to numeric if needed
df_copy['Diabetes_binary'] = pd.to_numeric(df_copy['Diabetes_binary'], errors='coerce')

# Group the data by 'Diabetes_binary' and 'Fruits' columns and calculate counts
grouped = df.groupby(['Fruits', 'Diabetes_binary']).size().unstack().fillna(0)

# Create the figure using plotly.graph_objects
fig = go.Figure()

# Add the 'No Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[0],
    orientation='h',
    name='No Diabetes',
    marker=dict(color='tomato')
))

# Add the 'Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[1],
    orientation='h',
    name='Diabetes',
    marker=dict(color='steelblue')
))

# Update the layout
fig.update_layout(
    title='Number of People with and without Diabetes by Fruits Eater',
    xaxis_title='Number of people',
    yaxis=dict(
        title='Fruits',
        tickmode='array',
        tickvals=[0, 1],
        ticktext=['No', 'Yes']
    ),
    legend_title='Diabetes',
    width=1000,
    height=800
)


# Show the plot using plotly.offline
fig.show()
```


Number of People with and without Diabetes by Fruits Eater
Fruits
Yes
No
0
20k
40k
60k
80k
100k
Number of people

3.2.10) Visualization for Veggies

In [16]:

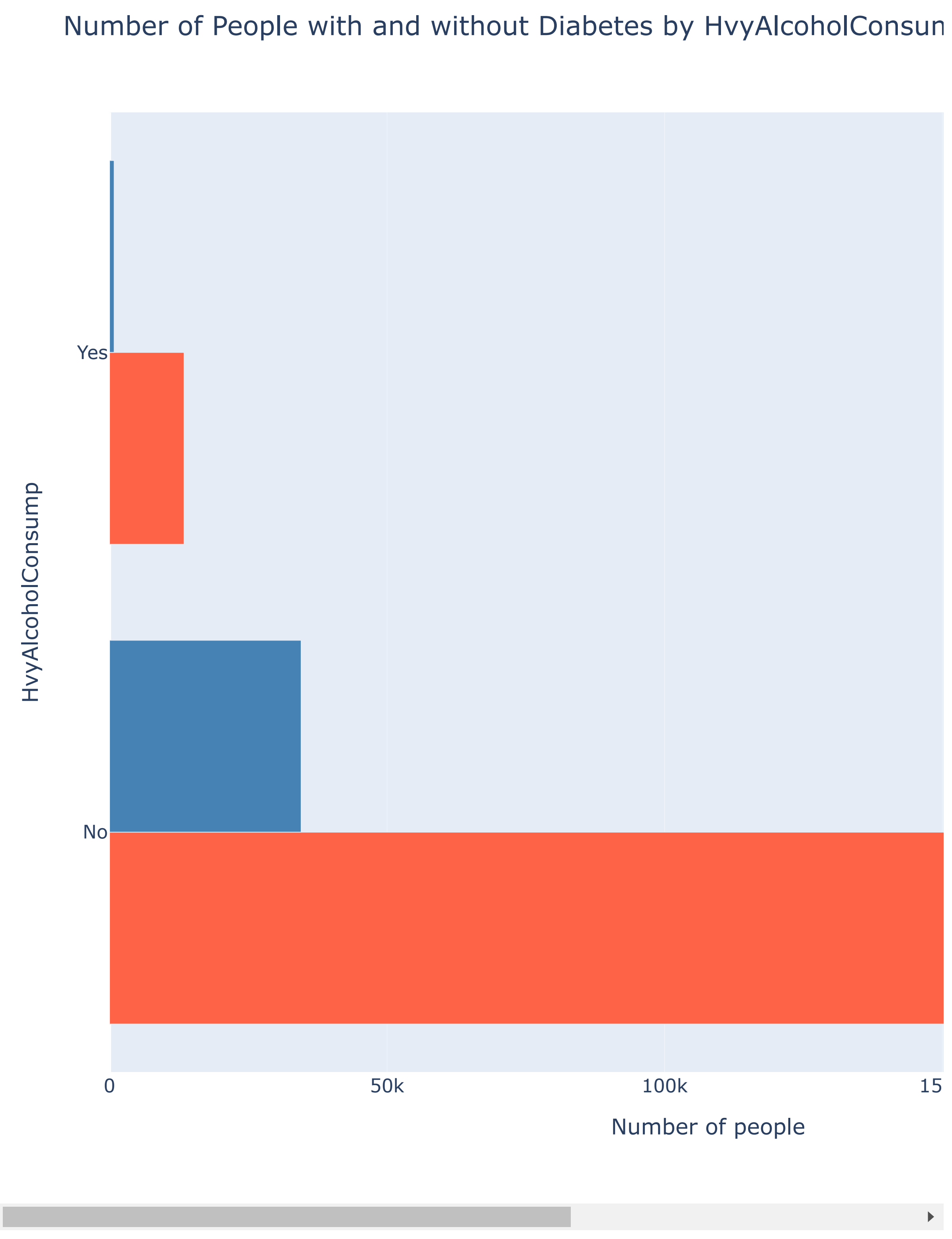
```
# Make a copy of the DataFrame
df_copy = df.copy()

# Set the 'Veggies' column as the index
df_copy.set_index('Veggies', inplace=True)

# Select the 'Diabetes_binary' column and convert it to numeric if needed
df_copy['Diabetes_binary'] = pd.to_numeric(df_copy['Diabetes_binary'], errors='coerce')

# Group the data by 'Diabetes_binary' and 'Veggies' columns and calculate counts
grouped = df.groupby(['Veggies', 'Diabetes_binary']).size().unstack().fillna(0)

# Create the figure using plotly.graph_objects
fig = go.Figure()

# Add the 'No Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[0],
    orientation='h',
    name='No Diabetes',
    marker=dict(color='tomato')
))

# Add the 'Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[1],
    orientation='h',
    name='Diabetes',
    marker=dict(color='steelblue')
))

# Update the layout
fig.update_layout(
    title='Number of People with and without Diabetes by Veggies Eater',
    xaxis_title='Number of people',
    yaxis=dict(
        title='Veggies',
        tickmode='array',
        tickvals=[0, 1],
        ticktext=['No', 'Yes']
    ),
    legend_title='Diabetes',
    width=1000,
    height=800
)


# Show the plot using plotly.offline
fig.show()
```


Number of People with and without Diabetes by Veggies Eater
Veggies
Yes
No
0
20k
40k
60k
80k
100k
120k
Number of people

3.2.11) Visualization for HvyAlcoholConsump

In [17]:

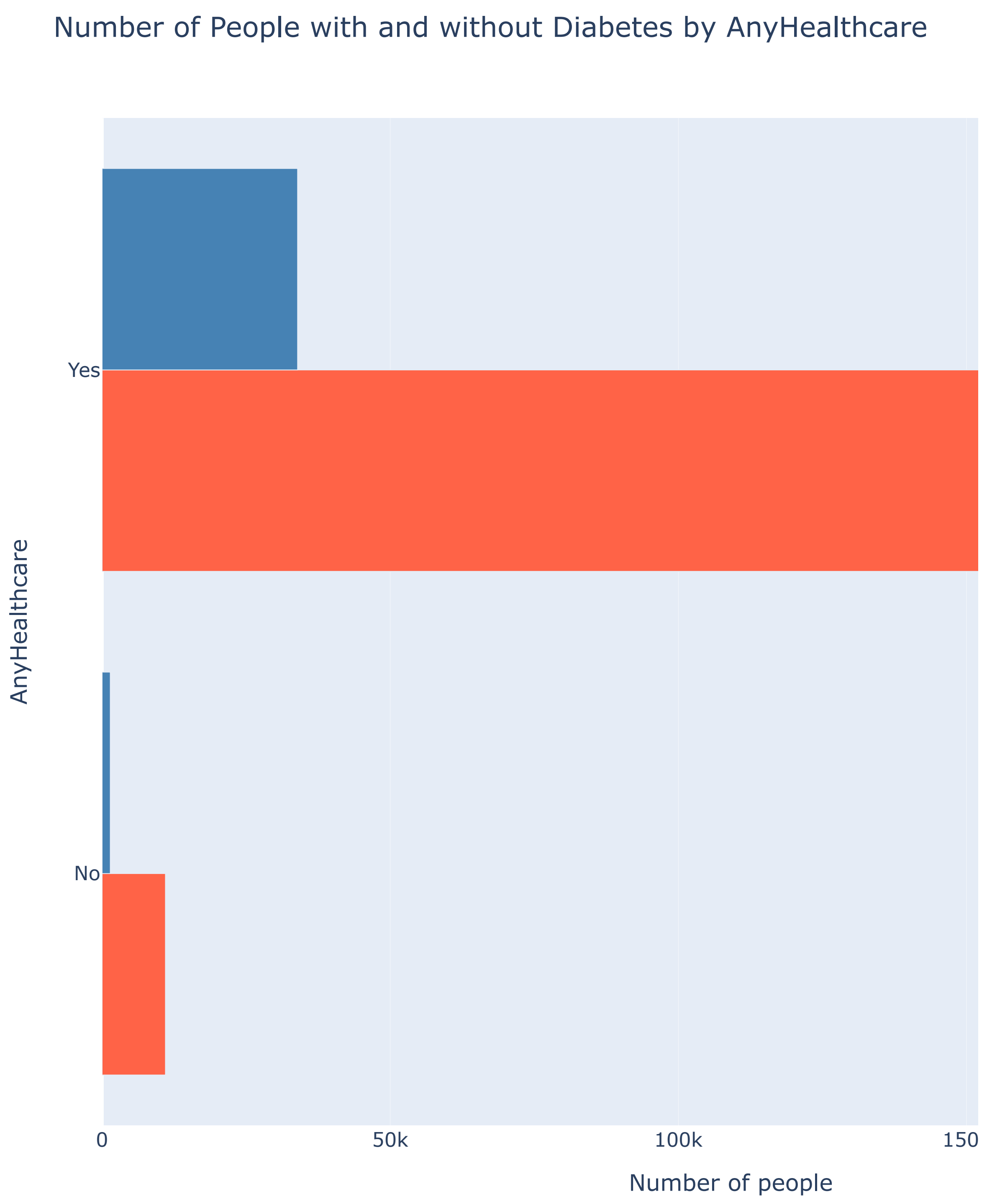
```
# Make a copy of the DataFrame
df_copy = df.copy()

# Set the 'HvyAlcoholConsump' column as the index
df_copy.set_index('HvyAlcoholConsump', inplace=True)

# Select the 'Diabetes_binary' column and convert it to numeric if needed
df_copy['Diabetes_binary'] = pd.to_numeric(df_copy['Diabetes_binary'], errors='coerce')

# Group the data by 'Diabetes_binary' and 'HvyAlcoholConsump' columns and calculate counts
grouped = df.groupby(['HvyAlcoholConsump', 'Diabetes_binary']).size().unstack().fillna(0)

# Create the figure using plotly.graph_objects
fig = go.Figure()

# Add the 'No Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[0],
    orientation='h',
    name='No Diabetes',
    marker=dict(color='tomato')
))

# Add the 'Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[1],
    orientation='h',
    name='Diabetes',
    marker=dict(color='steelblue')
))

# Update the layout
fig.update_layout(
    title='Number of People with and without Diabetes by HvyAlcoholConsump',
    xaxis_title='Number of people',
    yaxis=dict(
        title='HvyAlcoholConsump',
        tickmode='array',
        tickvals=[0, 1],
        ticktext=['No', 'Yes']
    ),
    legend_title='Diabetes',
    width=1000,
    height=800
)


# Show the plot using plotly.offline
fig.show()
```


Number of People with and without Diabetes by HvyAlcoholConsun
HvyAlcoholConsump
Yes
No
0
50k
100k
15
Number of people

3.2.12) Visualization for AnyHealthcare

In [18]:

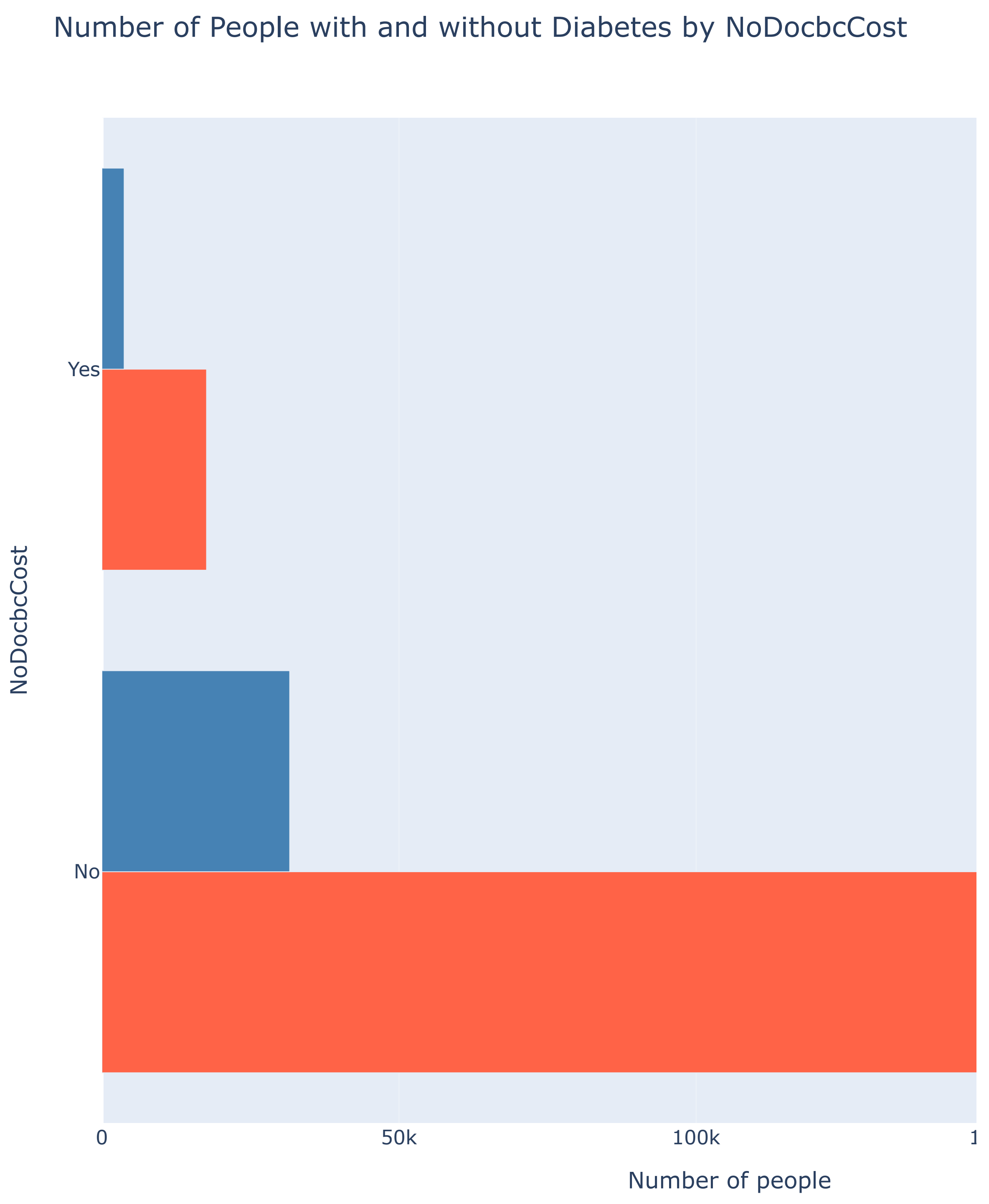
```
# Make a copy of the DataFrame
df_copy = df.copy()

# Set the 'AnyHealthcare' column as the index
df_copy.set_index('AnyHealthcare', inplace=True)

# Select the 'Diabetes_binary' column and convert it to numeric if needed
df_copy['Diabetes_binary'] = pd.to_numeric(df_copy['Diabetes_binary'], errors='coerce')

# Group the data by 'Diabetes_binary' and 'AnyHealthcare' columns and calculate counts
grouped = df.groupby(['AnyHealthcare', 'Diabetes_binary']).size().unstack().fillna(0)

# Create the figure using plotly.graph_objects
fig = go.Figure()

# Add the 'No Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[0],
    orientation='h',
    name='No Diabetes',
    marker=dict(color='tomato')
))

# Add the 'Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[1],
    orientation='h',
    name='Diabetes',
    marker=dict(color='steelblue')
))

# Update the layout
fig.update_layout(
    title='Number of People with and without Diabetes by AnyHealthcare',
    xaxis_title='Number of people',
    yaxis=dict(
        title='AnyHealthcare',
        tickmode='array',
        tickvals=[0, 1],
        ticktext=['No', 'Yes']
    ),
    legend_title='Diabetes',
    width=1000,
    height=800
)


# Show the plot using plotly.offline
fig.show()
```


Number of People with and without Diabetes by AnyHealthcare
AnyHealthcare
Yes
No
0
50k
100k
150
Number of people

3.2.13) Visualization for NoDocbcCost

In [19]:

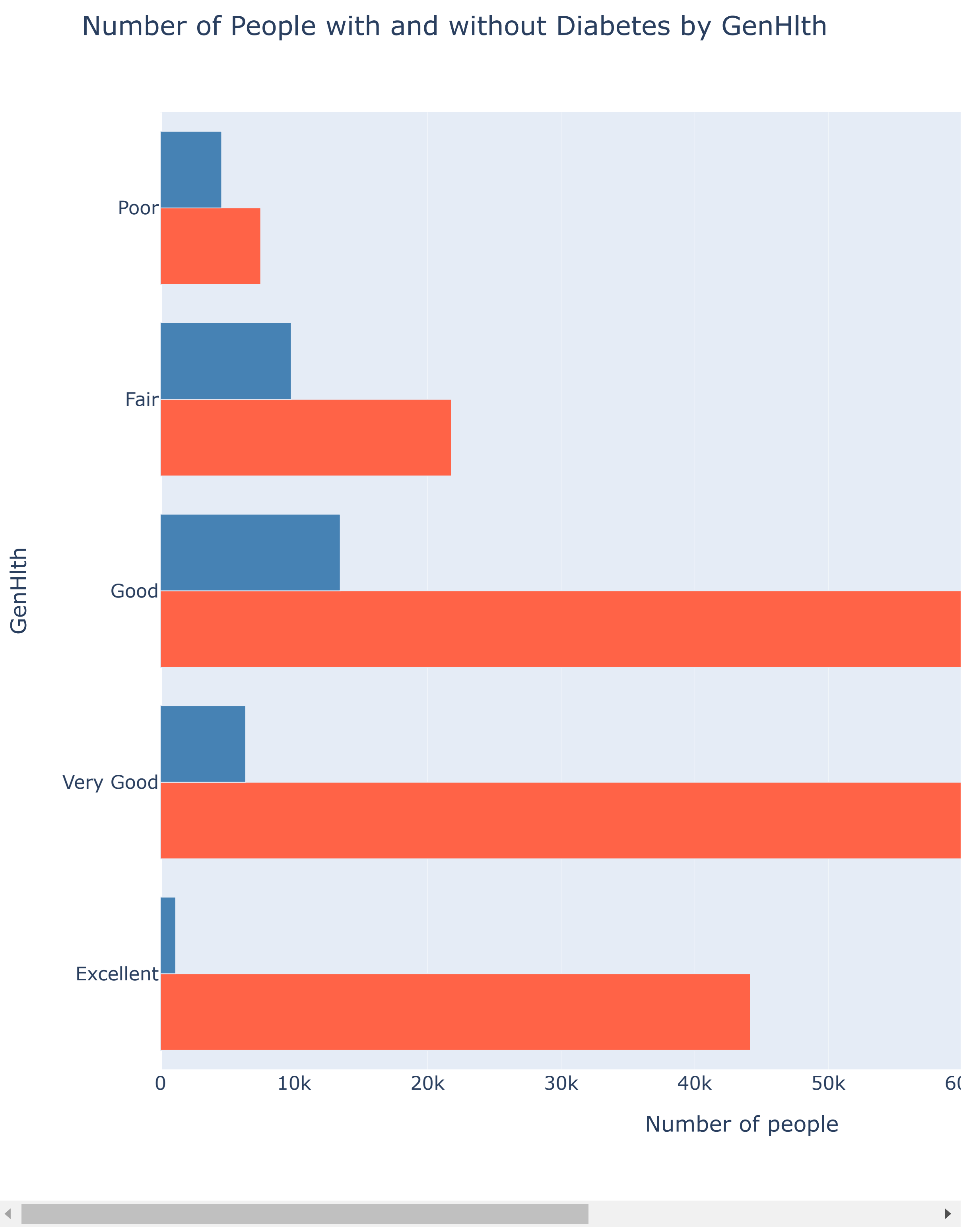
```
# Make a copy of the DataFrame
df_copy = df.copy()

# Set the 'AnyHealthcare' column as the index
df_copy.set_index('NoDocbcCost', inplace=True)

# Select the 'Diabetes_binary' column and convert it to numeric if needed
df_copy['Diabetes_binary'] = pd.to_numeric(df_copy['Diabetes_binary'], errors='coerce')

# Group the data by 'Diabetes_binary' and 'NoDocbcCost' columns and calculate counts
grouped = df.groupby(['NoDocbcCost', 'Diabetes_binary']).size().unstack().fillna(0)

# Create the figure using plotly.graph_objects
fig = go.Figure()

# Add the 'No Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[0],
    orientation='h',
    name='No Diabetes',
    marker=dict(color='tomato')
))

# Add the 'Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[1],
    orientation='h',
    name='Diabetes',
    marker=dict(color='steelblue')
))

# Update the layout
fig.update_layout(
    title='Number of People with and without Diabetes by NoDocbcCost',
    xaxis_title='Number of people',
    yaxis=dict(
        title='NoDocbcCost',
        tickmode='array',
        tickvals=[0, 1],
        ticktext=['No', 'Yes']
    ),
    legend_title='Diabetes',
    width=1000,
    height=800
)


# Show the plot using plotly.offline
fig.show()
```


Number of People with and without Diabetes by NoDocbcCost
NoDocbcCost
Yes
No
0
50k
100k
1
Number of people

3.2.14) Visualization for GenHlth

In [20]:

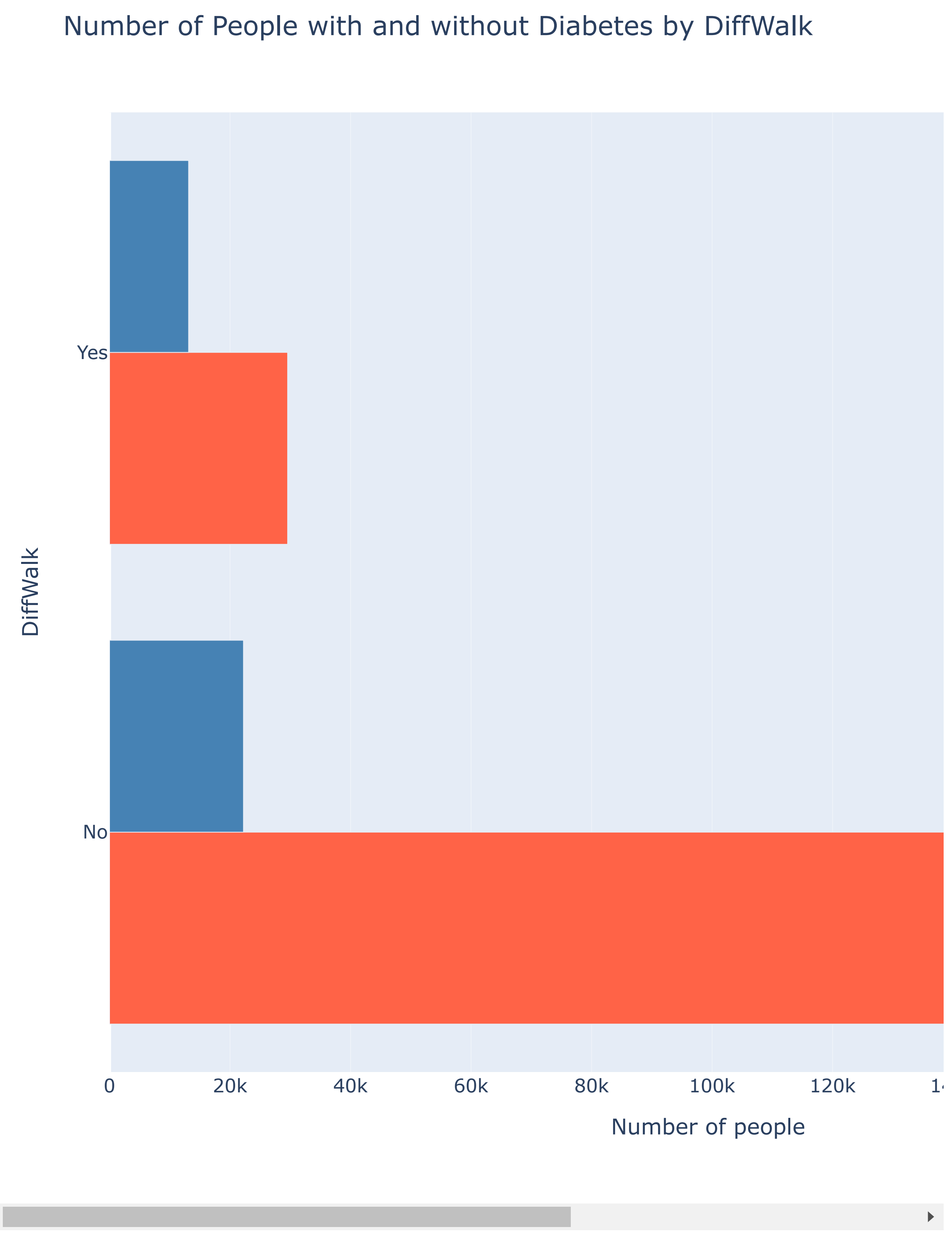
```
# Make a copy of the DataFrame
df_copy = df.copy()

# Set the 'GenHlth' column as the index
df_copy.set_index('GenHlth', inplace=True)

# Select the 'Diabetes_binary' column and convert it to numeric if needed
df_copy['Diabetes_binary'] = pd.to_numeric(df_copy['Diabetes_binary'], errors='coerce')

# Group the data by 'Diabetes_binary' and 'GenHlth' columns and calculate counts
grouped = df.groupby(['GenHlth', 'Diabetes_binary']).size().unstack().fillna(0)

# Create the figure using plotly.graph_objects
fig = go.Figure()

# Add the 'No Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[0],
    orientation='h',
    name='No Diabetes',
    marker=dict(color='tomato')
))

# Add the 'Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[1],
    orientation='h',
    name='Diabetes',
    marker=dict(color='steelblue')
))

# Update the layout
fig.update_layout(
    title='Number of People with and without Diabetes by GenHlth',
    xaxis_title='Number of people',
    yaxis=dict(
        title='GenHlth',
        tickmode='array',
        tickvals=[1, 2, 3, 4, 5],
        ticktext=['Excellent', 'Very Good', 'Good', 'Fair', 'Poor']
    ),
    legend_title='Diabetes',
    width=1000,
    height=800
)


# Show the plot using plotly.offline
fig.show()
```


Number of People with and without Diabetes by GenHlth
Poor
Fair
Good
Very Good
Excellent
GenHlth
0
10k
20k
30k
40k
50k
6(
Number of people

3.2.15) Visualization for DiffWalk

In [21]:

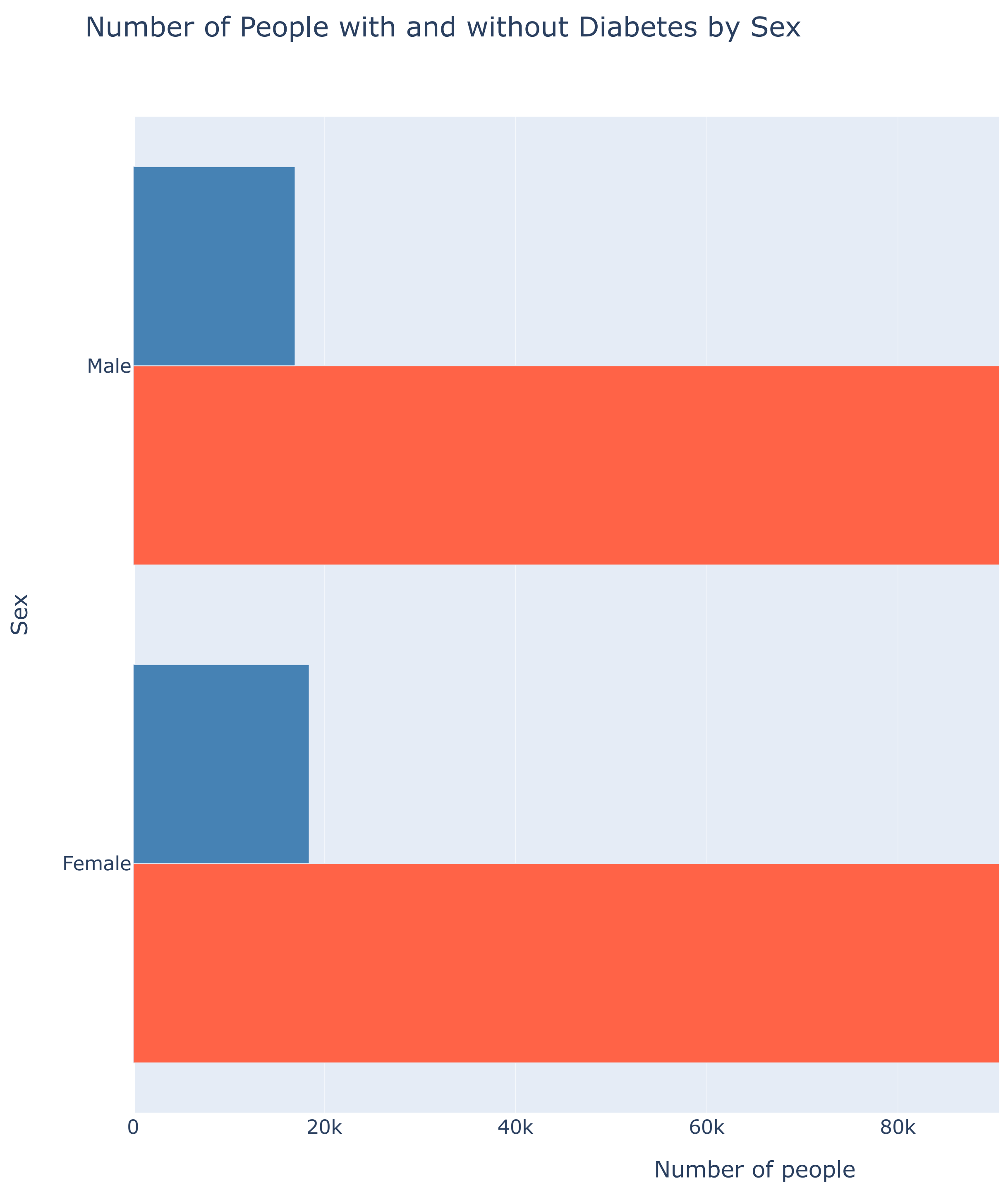
```
# Make a copy of the DataFrame
df_copy = df.copy()

# Set the 'DiffWalk' column as the index
df_copy.set_index('DiffWalk', inplace=True)

# Select the 'Diabetes_binary' column and convert it to numeric if needed
df_copy['Diabetes_binary'] = pd.to_numeric(df_copy['Diabetes_binary'], errors='coerce')

# Group the data by 'Diabetes_binary' and 'DiffWalk' columns and calculate counts
grouped = df.groupby(['DiffWalk', 'Diabetes_binary']).size().unstack().fillna(0)

# Create the figure using plotly.graph_objects
fig = go.Figure()

# Add the 'No Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[0],
    orientation='h',
    name='No Diabetes',
    marker=dict(color='tomato')
))

# Add the 'Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[1],
    orientation='h',
    name='Diabetes',
    marker=dict(color='steelblue')
))

# Update the layout
fig.update_layout(
    title='Number of People with and without Diabetes by DiffWalk',
    xaxis_title='Number of people',
    yaxis=dict(
        title='DiffWalk',
        tickmode='array',
        tickvals=[0, 1],
        ticktext=['No', 'Yes']
    ),
    legend_title='Diabetes',
    width=1000,
    height=800
)


# Show the plot using plotly.offline
fig.show()
```


Number of People with and without Diabetes by DiffWalk
DiffWalk
Yes
No
0
20k
40k
60k
80k
100k
120k
Number of people

3.2.16) Visualization for Sex

In [22]:

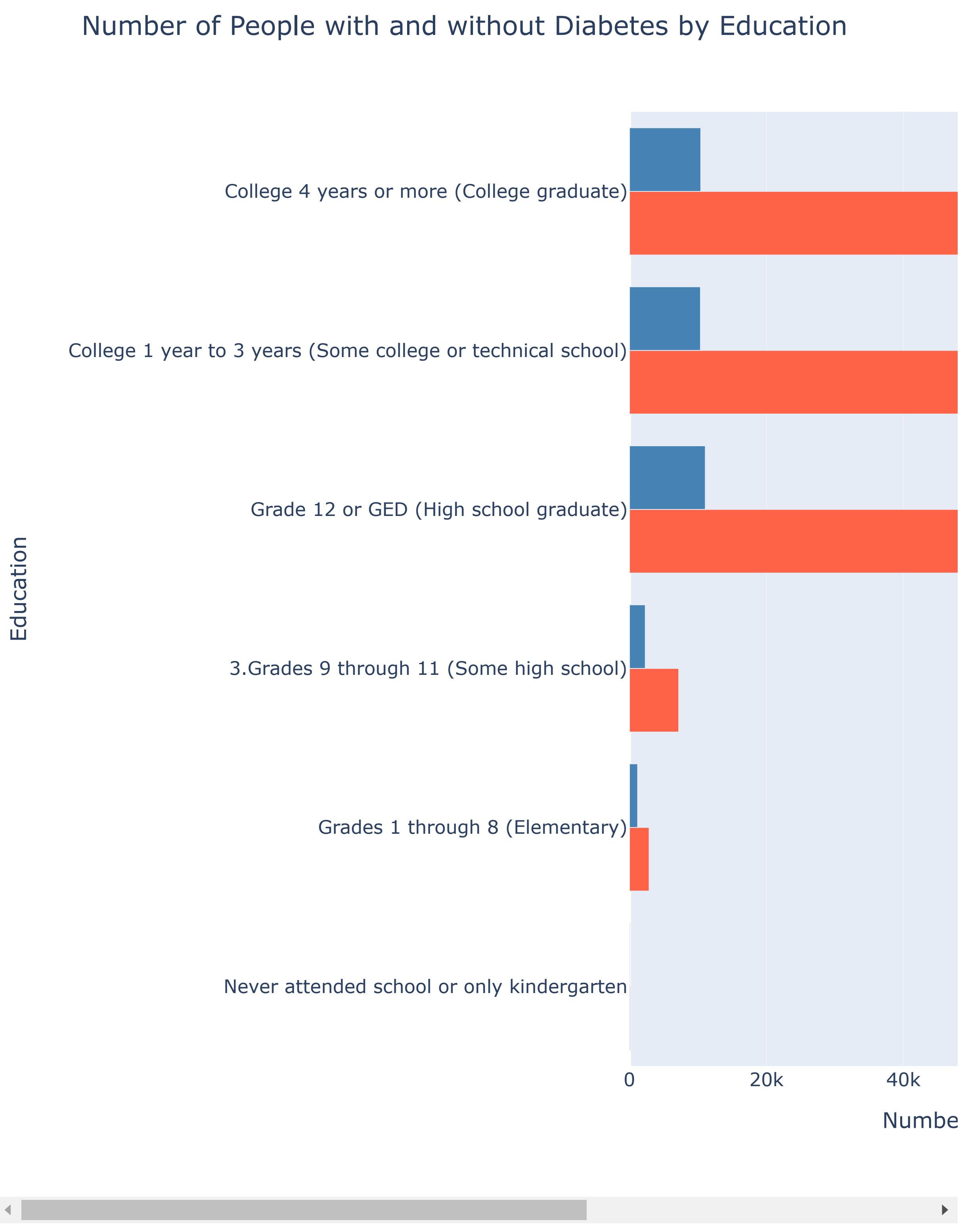
```
# Make a copy of the DataFrame
df_copy = df.copy()

# Set the 'Sex' column as the index
df_copy.set_index('Sex', inplace=True)

# Select the 'Diabetes_binary' column and convert it to numeric if needed
df_copy['Diabetes_binary'] = pd.to_numeric(df_copy['Diabetes_binary'], errors='coerce')

# Group the data by 'Diabetes_binary' and 'Sex' columns and calculate counts
grouped = df.groupby(['Sex', 'Diabetes_binary']).size().unstack().fillna(0)

# Create the figure using plotly.graph_objects
fig = go.Figure()

# Add the 'No Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[0],
    orientation='h',
    name='No Diabetes',
    marker=dict(color='tomato')
))

# Add the 'Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[1],
    orientation='h',
    name='Diabetes',
    marker=dict(color='steelblue')
))

# Update the layout
fig.update_layout(
    title='Number of People with and without Diabetes by Sex',
    xaxis_title='Number of people',
    yaxis=dict(
        title='Sex',
        tickmode='array',
        tickvals=[0, 1],
        ticktext=['Female', 'Male']
    ),
    legend_title='Diabetes',
    width=1000,
    height=800
)

# Show the plot using plotly.offline
fig.show()
```


Number of People with and without Diabetes by Sex
Sex
Male
Female
0
20k
40k
60k
80k
Number of people

3.2.17) Visualization for Education

In [23]:

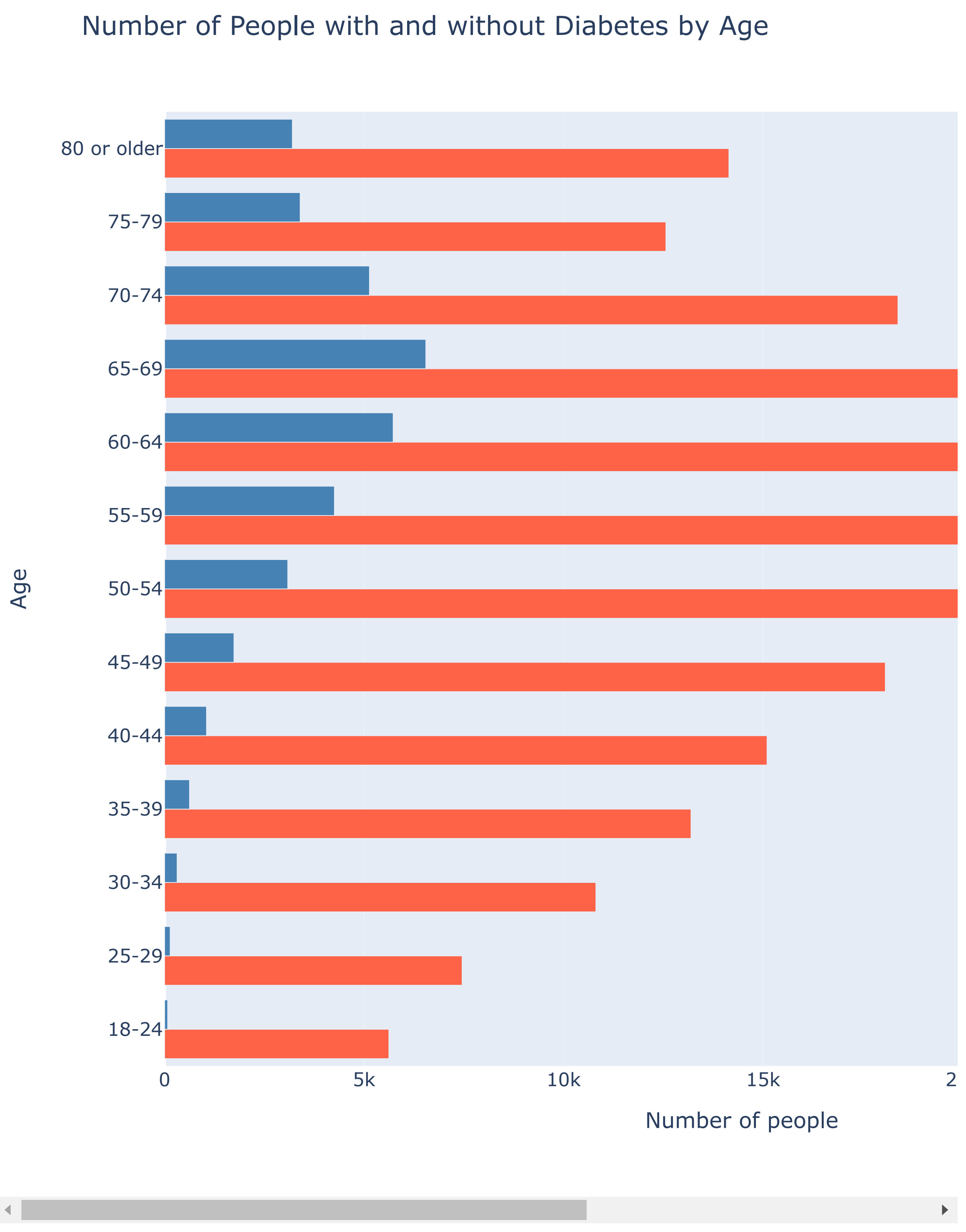
```
# Make a copy of the DataFrame
df_copy = df.copy()

# Set the 'Education' column as the index
df_copy.set_index('Education', inplace=True)

# Select the 'Diabetes_binary' column and convert it to numeric if needed
df_copy['Diabetes_binary'] = pd.to_numeric(df_copy['Diabetes_binary'], errors='coerce')

# Group the data by 'Diabetes_binary' and 'Education' columns and calculate counts
grouped = df.groupby(['Education', 'Diabetes_binary']).size().unstack().fillna(0)

# Create the figure using plotly.graph_objects
fig = go.Figure()

# Add the 'No Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[0],
    orientation='h',
    name='No Diabetes',
    marker=dict(color='tomato')
))

# Add the 'Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[1],
    orientation='h',
    name='Diabetes',
    marker=dict(color='steelblue')
))

# Update the layout
fig.update_layout(
    title='Number of People with and without Diabetes by Education',
    xaxis_title='Number of people',
    yaxis=dict(
        title='Education',
        tickmode='array',
        tickvals=[1, 2, 3, 4, 5, 6],
        ticktext=['Never attended school or only kindergarten', 'Grades 1 through 8 (Elementar
                  'Grade 12 or GED (High school graduate)' ,' College 1 year to 3 years (Some
                  'College 4 years or more (College graduate)']
    ),
    legend_title='Diabetes',
    width=1000,
    height=800
)


# Show the plot using plotly.offline
fig.show()
```


Number of People with and without Diabetes by Education
Education
College 4 years or more (College graduate)
College 1 year to 3 years (Some college or technical school)
Grade 12 or GED (High school graduate)
3.Grades 9 through 11 (Some high school)
Grades 1 through 8 (Elementary)
Never attended school or only kindergarten
0
20k
40k
Numbe

3.2.18) Visualization for Age

In [24]:

```
# Make a copy of the DataFrame
df_copy = df.copy()

# Set the 'Education' column as the index
df_copy.set_index('Age', inplace=True)

# Select the 'Diabetes_binary' column and convert it to numeric if needed
df_copy['Diabetes_binary'] = pd.to_numeric(df_copy['Diabetes_binary'], errors='coerce')

# Group the data by 'Diabetes_binary' and 'Education' columns and calculate counts
grouped = df.groupby(['Age', 'Diabetes_binary']).size().unstack().fillna(0)

# Create the figure using plotly.graph_objects
fig = go.Figure()

# Add the 'No Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[0],
    orientation='h',
    name='No Diabetes',
    marker=dict(color='tomato')
))

# Add the 'Diabetes' bar trace
fig.add_trace(go.Bar(
    y=grouped.index,
    x=grouped[1],
    orientation='h',
    name='Diabetes',
    marker=dict(color='steelblue')
))

# Update the layout
fig.update_layout(
    title='Number of People with and without Diabetes by Age',
    xaxis_title='Number of people',
    yaxis=dict(
        title='Age',
        tickmode='array',
        tickvals=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13],
        ticktext=['18-24', '25-29', '30-34', '35-39', '40-44','45-49','50-54','55-59','60-64',
    ),
    legend_title='Diabetes',
    width=1000,
    height=800
)


# Show the plot using plotly.offline
fig.show()
```



3.3) Data Visualization (Continuous Data)

3.3.1) Visualization for BMI

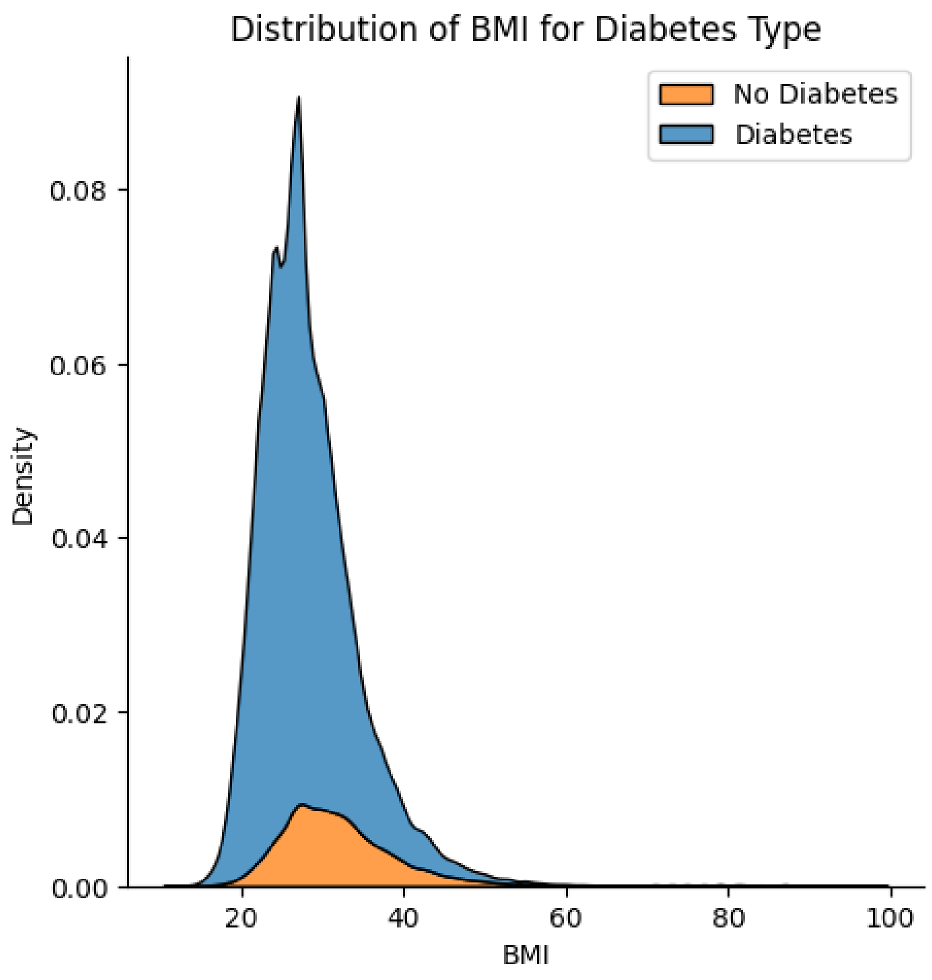
Explanation

In this part, we are going to visualize the continuous data using density graph. The code below creates a density plot using Seaborn to visualize the distribution of target column for different diabetes types (No Diabetes and Diabetes).

The plot displays stacked distributions, with each category represented by a different color. It allows for a quick comparison with the target columns between the two diabetes groups, helping to identify potential differences or patterns in the target columns.

In [25]:

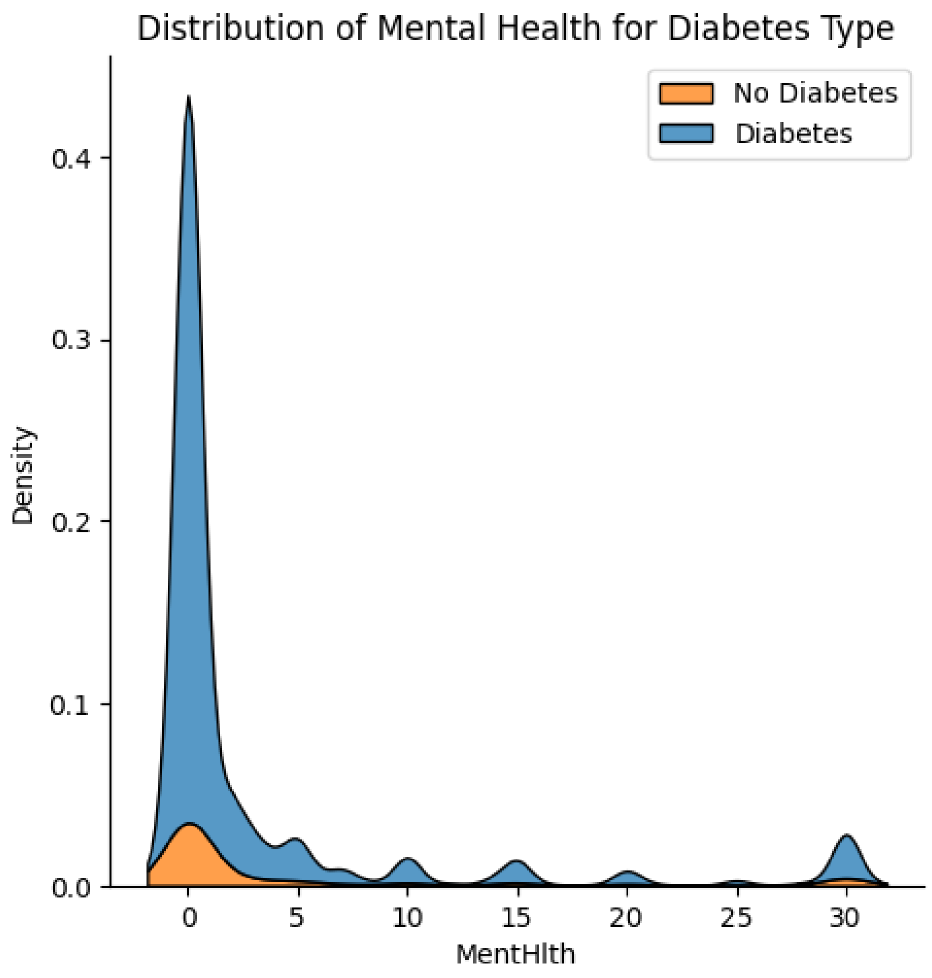
```
# Set the custom legend labels
legend_labels = ['No Diabetes', 'Diabetes']

# Create the density plot with stacked distributions
sns.displot(df, x="BMI", hue="Diabetes_binary", kind="kde", multiple="stack", legend=False)

# Set the custom legend labels
plt.legend(legend_labels)

# Set the plot title
plt.title('Distribution of BMI for Diabetes Type')

# Show the plot
plt.show()
```



3.3.2) Visualization for MentHlth

In [26]:

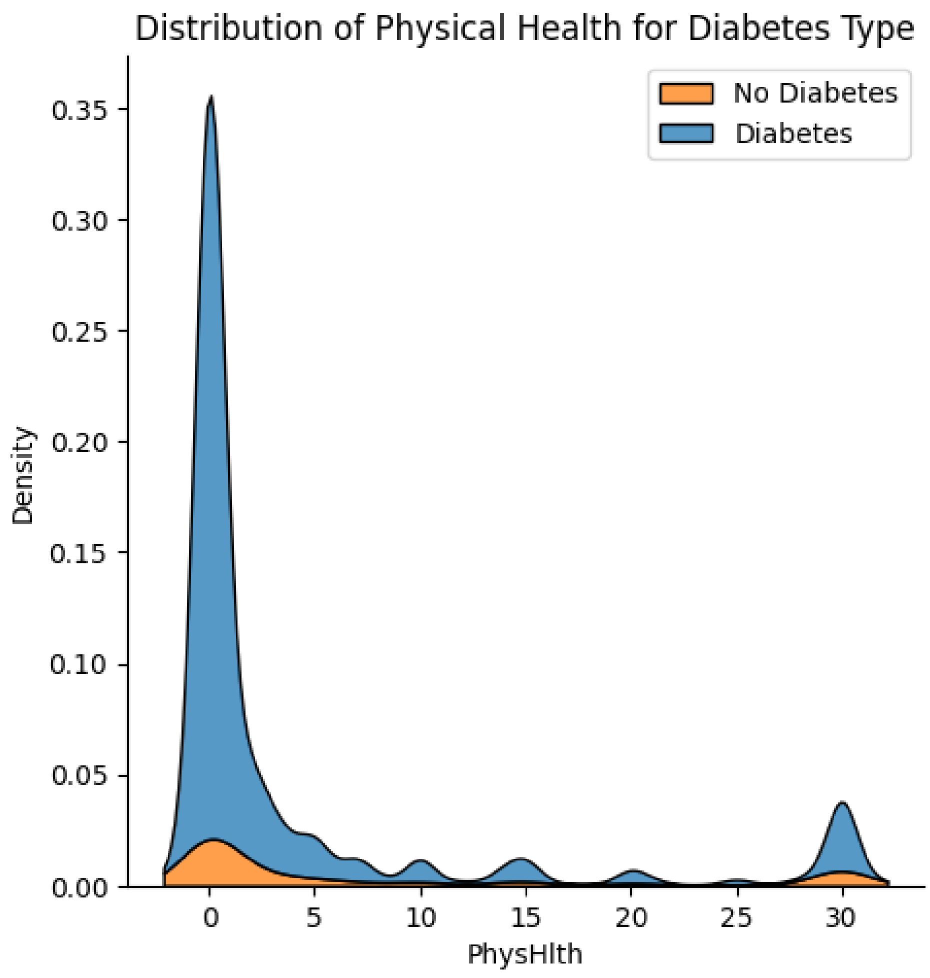
```
# Set the custom legend labels
legend_labels = ['No Diabetes', 'Diabetes']

# Create the density plot with stacked distributions
sns.displot(df, x="MentHlth", hue="Diabetes_binary", kind="kde", multiple="stack", legend=Fals

# Set the custom legend labels
plt.legend(legend_labels)

# Set the plot title
plt.title('Distribution of Mental Health for Diabetes Type')

# Show the plot
plt.show()
```



3.3.3) Visualization for PhysHlth

In [27]:

```
# Set the custom legend labels
legend_labels = ['No Diabetes', 'Diabetes']

# Create the density plot with stacked distributions
sns.displot(df, x="PhysHlth", hue="Diabetes_binary", kind="kde", multiple="stack", legend=Fals

# Set the custom legend labels
plt.legend(legend_labels)

# Set the plot title
plt.title('Distribution of Physical Health for Diabetes Type')

# Show the plot
plt.show()
```



4) Data Prepocessing

4.1) Replacing Missing Values

In [28]:

```
# creating bool series True for NaN values
bool_series = pd.isnull(df["BMI"])

# filtering data
# displaying data only with BMI = NaN
df[bool_series]
```

Out[28]:

	Diabetes_binary	HighBP	HighChol	CholCheck	BMI	Smoker	Stroke	HeartDiseaseorAttack	Pl
9	0	0	0	1	NaN	0.0	0	0	
10	1	0	0	1	NaN	1.0	0	0	
11	0	1	1	1	NaN	1.0	0	0	
12	0	0	0	1	NaN	1.0	0	0	
13	1	1	1	1	NaN	0.0	0	0	
14	0	0	1	1	NaN	1.0	1	0	
15	0	1	0	1	NaN	0.0	0	0	
16	0	1	1	1	NaN	0.0	0	0	
17	1	0	0	1	NaN	1.0	0	0	
18	0	0	0	0	NaN	0.0	0	0	
19	0	0	1	1	NaN	0.0	0	0	
20	0	1	1	1	NaN	0.0	1	1	
21	0	1	1	1	NaN	1.0	0	0	
22	0	0	0	1	NaN	1.0	0	0	
23	1	1	0	1	NaN	0.0	0	0	
24	0	1	1	1	NaN	1.0	0	0	
25	0	0	0	1	NaN	0.0	0	0	
26	1	1	1	1	NaN	1.0	1	1	
27	1	1	1	1	NaN	1.0	0	1	
28	1	1	1	1	NaN	1.0	0	0	
29	0	0	1	1	NaN	1.0	0	0	
30	1	1	1	1	NaN	1.0	1	0	
31	0	1	0	1	NaN	1.0	0	0	
32	0	0	0	1	NaN	0.0	0	0	
33	0	1	0	1	NaN	0.0	0	0	
34	1	1	1	1	NaN	1.0	0	0	

26 rows × 22 columns

Explanation

This code creates a boolean series named "bool_series" where each element is True if the corresponding value in the "BMI" column of the DataFrame "df" is NaN (missing value), and False otherwise.

Then, it filters the data by displaying only the rows where the "BMI" column has missing values (NaN). This helps to identify the specific rows in the DataFrame where the BMI values are missing.

In [29]:

```
# creating bool series True for NaN values
bool_series_smoker = pd.isnull(df["Smoker"])

# filtering data
# displaying data only with Smoker = NaN
df[bool_series_smoker]
```

Out[29]:

	Diabetes_binary	HighBP	HighChol	CholCheck	BMI	Smoker	Stroke	HeartDiseaseorAttack
60	1	1	0	1	27.0	NaN	0	0
61	0	1	0	1	27.0	NaN	0	0
62	0	1	0	1	34.0	NaN	0	0
63	0	1	1	1	30.0	NaN	0	0
64	0	1	1	1	27.0	NaN	0	0
...	...	...	...	...	...	...	...	...
138	0	0	1	1	27.0	NaN	0	0
139	0	1	1	1	31.0	NaN	0	0
140	0	0	0	1	23.0	NaN	0	0
141	0	1	1	1	28.0	NaN	1	0
142	0	0	0	1	26.0	NaN	0	0

83 rows × 22 columns

Explanation

This code creates a boolean series named "bool_series_smoker" where each element is True if the corresponding value in the "Smoker" column of the DataFrame "df" is NaN (missing value), and False otherwise.

Then, it filters the data by displaying only the rows where the "Smoker" column has missing values (NaN). This helps to identify the specific rows in the DataFrame where the "Smoker" values are missing.

In [30]:

```
# creating bool series True for NaN values
bool_series_age = pd.isnull(df["Age"])

# filtering data
# displaying data only with Smoker = NaN
df[bool_series_age]
```

Out[30]:

	Diabetes_binary	HighBP	HighChol	CholCheck	BMI	Smoker	Stroke	HeartDiseaseorAttack
196	0	0	0	1	29.0	0.0	0	0
197	1	1	1	1	29.0	1.0	1	1
198	0	0	1	1	27.0	0.0	0	0
199	0	0	0	1	21.0	0.0	0	0
200	0	0	0	1	28.0	0.0	0	0
...	...	...	...	...	...	...	...	...
386	0	0	0	1	26.0	0.0	0	0
387	0	0	1	1	27.0	1.0	0	0
388	1	1	0	1	34.0	1.0	1	0
389	0	1	0	1	43.0	0.0	0	0
390	0	1	1	1	30.0	0.0	0	0

195 rows × 22 columns

Explanation

This code creates a boolean series named "bool_series_age" where each element is True if the corresponding value in the "Age" column of the DataFrame "df" is NaN (missing value), and False otherwise.

Then, it filters the data by displaying only the rows where the "Age" column has missing values (NaN). This helps to identify the specific rows in the DataFrame where the "Age" values are missing.

In [31]:

```
# Calculate median for BMI and Age
median_bmi = df['BMI'].median()
median_age = df['Age'].median()

# Calculate mode for Smoker
mode_smoker = df['Smoker'].mode().iloc[0]

# Calculate mode for Age
mode_age = df['Age'].mode().iloc[0]

# Replace missing values with median and mode
df['BMI'].fillna(median_bmi, inplace=True)
df['Age'].fillna(mode_age, inplace=True)
df['Smoker'].fillna(mode_smoker, inplace=True)

# Save the updated DataFrame back to the CSV file
# df.to_csv('diabetes_2_missing_values.csv', index=False)

print("Median BMI: ",median_bmi)
print("Median Age: ",mode_age)
print("Mode Smoker: ",median_bmi)
```

```
Median BMI:  27.0
Median Age:  9.0
Mode Smoker:  27.0
```

Explanation

This code calculates the median for the 'BMI' and 'Age' columns in the DataFrame 'df' using the 'median()' method. It also calculates the mode for the 'Smoker' and 'Age' columns using the 'mode()' method.

Next, the code replaces the missing values (NaN) in the 'BMI', 'Age', and 'Smoker' columns with their respective calculated median and mode values using the 'fillna()' method. This process helps to impute the missing data with representative values based on the distribution of the existing data in each column.

In [32]:

```
df.isnull().sum().sort_values(ascending=False)
```

Out[32]:

```
Diabetes_binary         0
HighBP                  0
Education               0
Age                     0
Sex                     0
DiffWalk                0
PhysHlth                0
MentHlth                0
GenHlth                 0
NoDocbcCost             0
AnyHealthcare           0
HvyAlcoholConsump       0
Veggies                 0
Fruits                  0
PhysActivity            0
HeartDiseaseorAttack    0
Stroke                  0
Smoker                  0
BMI                     0
CholCheck               0
HighChol                0
Income                  0
dtype: int64
```

Explanation

Above code uses the 'isnull()' method to identify missing values (NaN) in each column of the DataFrame 'df'. The 'sum()' method is then applied to calculate the total number of missing values in each column. The result counts are sorted in descending order using the 'sort_values()' method to display columns with the highest number of missing values at the top.

4.2) Remove Dupplicate

In [33]:

```
duplicate_rows = df[df.duplicated(keep=False)]

# Display the duplicate rows
print("Duplicate Rows:")
print(duplicate_rows)
```

```
Duplicate Rows:
        Diabetes_binary  HighBP  HighChol  CholCheck   BMI  Smoker  Stroke  \
5                     0       1         1          1  25.0     1.0       0   
25                    0       0         0          1  27.0     0.0       0   
44                    0       0         1          1  31.0     1.0       0   
52                    1       1         1          1  27.0     1.0       0   
53                    0       0         0          1  31.0     0.0       0   
...                 ...     ...       ...        ...   ...     ...     ...   
253492                1       1         1          1  33.0     0.0       0   
253550                0       0         0          1  25.0     0.0       0   
253563                0       0         1          1  24.0     1.0       0   
253597                0       0         0          1  24.0     0.0       0   
253638                0       0         0          1  24.0     0.0       0   

        HeartDiseaseorAttack  PhysActivity  Fruits  ...  AnyHealthcare  \
5                          0             1       1  ...              1   
25                         0             1       1  ...              1   
44                         0             0       1  ...              1   
52                         0             0       0  ...              1   
53                         0             1       0  ...              1   
...                      ...           ...     ...  ...            ...   
253492                     0             1       1  ...              1   
253550                     0             1       1  ...              1   
253563                     0             1       1  ...              1   
253597                     0             1       1  ...              1   
253638                     0             1       1  ...              1   

        NoDocbcCost  GenHlth  MentHlth  PhysHlth  DiffWalk  Sex   Age  \
5                 0        2         0         2         0    1  10.0   
25                0        2         0         0         0    0   5.0   
44                0        2         0         0         0    0   8.0   
52                0        5         0        30         1    0  10.0   
53                0        2         0         0         0    0  10.0   
...             ...      ...       ...       ...       ...  ...   ...   
253492            0        3         0         0         0    1   9.0   
253550            0        1         0         0         0    0   7.0   
253563            0        2         0         0         0    1   8.0   
253597            0        2         0         0         0    0   5.0   
253638            0        2         0         0         0    1   1.0   

        Education  Income  
5               6       8  
25              6       8  
44              5       8  
52              4       5  
53              5       6  
...           ...     ...  
253492          6       6  
253550          6       8  
253563          6       8  
253597          6       8  
253638          4       6  

[35578 rows x 22 columns]
```

Explanation

The code above uses the 'duplicated()' method on the DataFrame 'df' with the parameter 'keep=False'. This will identify all the duplicate rows in the DataFrame, keeping all instances of the duplicated rows.

In [34]:

```
# Drop duplicate rows in place
df.drop_duplicates(inplace=True)
```

Explanation

The code uses the 'drop_duplicates()' method on the DataFrame 'df' with the parameter 'inplace=True'. This method removes duplicate rows from the DataFrame in place, meaning that it modifies the DataFrame directly without creating a new copy.

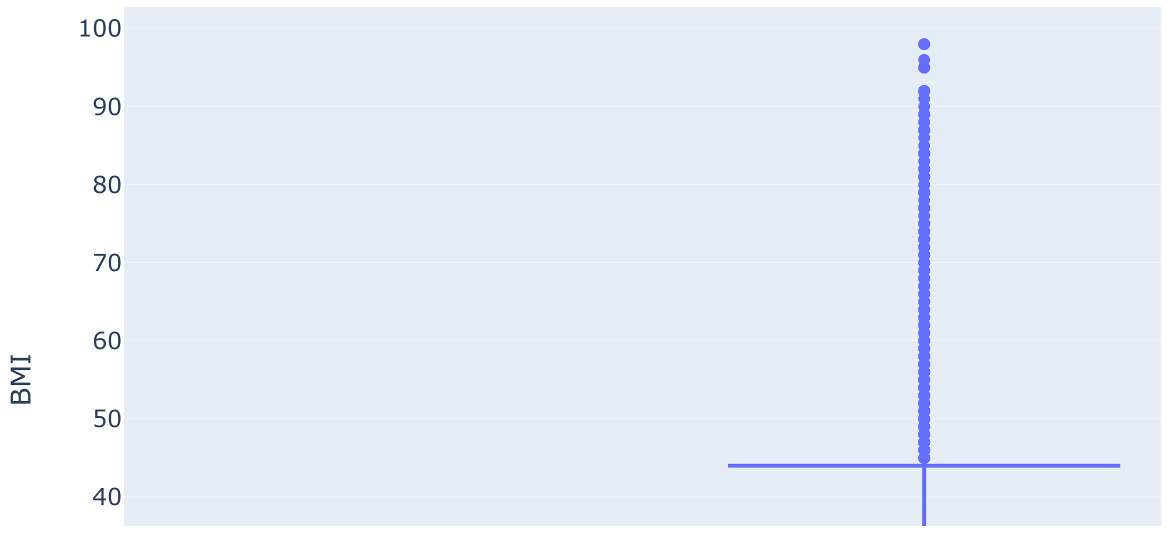
After executing this code, the DataFrame 'df' will have any duplicate rows removed, and it will be updated with the modified version containing only unique rows.

4.3) Remove Outliers

4.3.1) Check and remove outlier for BMI

In [35]:

```
df = df.reset_index(drop=True)
fig = px.box(df, y="BMI")
fig.show()
```



Explanation

The code resets the DataFrame index, and then it creates and displays a box plot for the 'BMI' column using Plotly Express. The box plot shows the distribution of 'BMI' values, including median, quartiles, and potential outliers.

In [36]:

```
# Calculate the IQR for the 'BMI' column
Q1 = df['BMI'].quantile(0.25)
Q3 = df['BMI'].quantile(0.75)
IQR = Q3 - Q1

# Define the IQR threshold for outlier detection
threshold = 1.5

# Remove rows with 'BMI' values outside the IQR range
df = df[~((df['BMI'] < (Q1 - threshold * IQR)) | (df['BMI'] > (Q3 + threshold * IQR)))]
```

Explantion

The code resets the DataFrame index, and then it creates and displays a box plot for the 'BMI' column using Plotly Express. The box plot shows the distribution of 'BMI' values, including median, quartiles, and potential outliers.

4.4) Balance Dataset

In [37]:

```
print(df['Diabetes_binary'].value_counts())
```

```
Diabetes_binary
0    190734
1     33100
Name: count, dtype: int64
```

Explanation

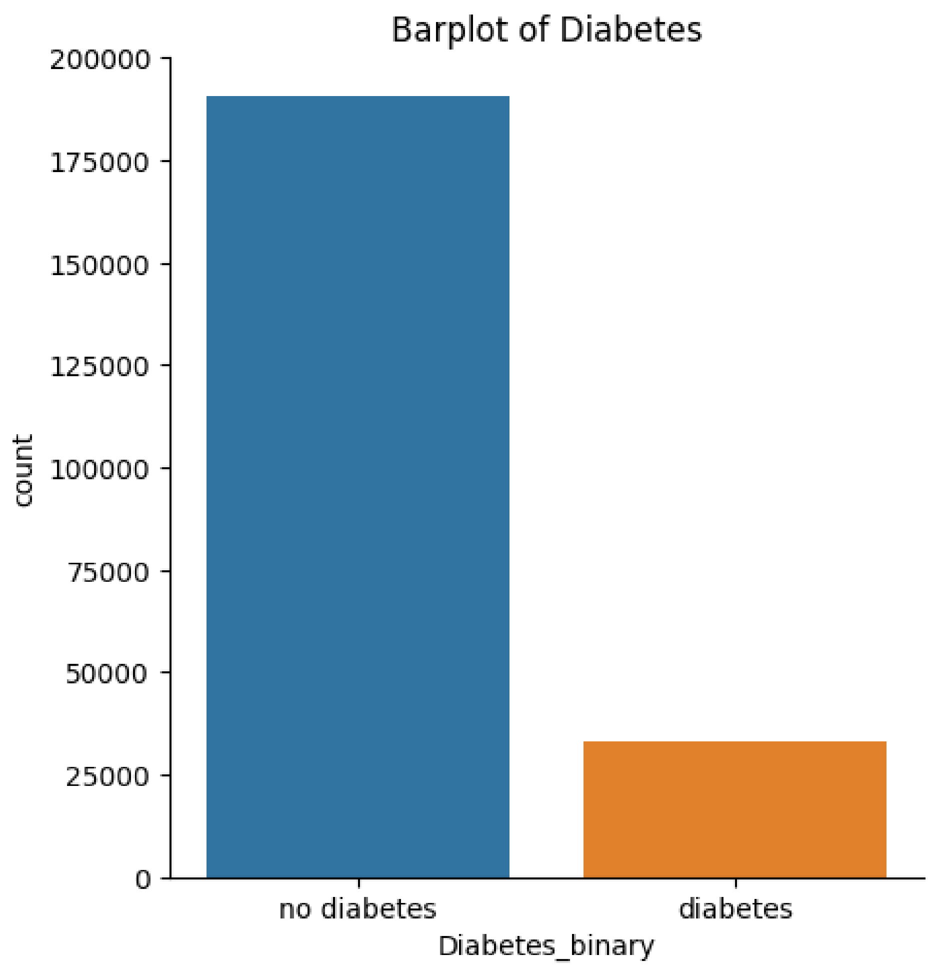
This code prints the count of occurrences of each unique value in the 'Diabetes_binary' column of the DataFrame 'df'. It provides a quick summary of how many instances have a value of '0' (indicating no diabetes) and how many instances have a value of '1' (indicating diabetes).

In [38]:

```
sns.catplot(x="Diabetes_binary", kind='count', data=df)
plt.xticks([0,1],['no diabetes','diabetes'])
plt.title('Barplot of Diabetes')
```

Out[38]:

```
Text(0.5, 1.0, 'Barplot of Diabetes')
```



Explanation

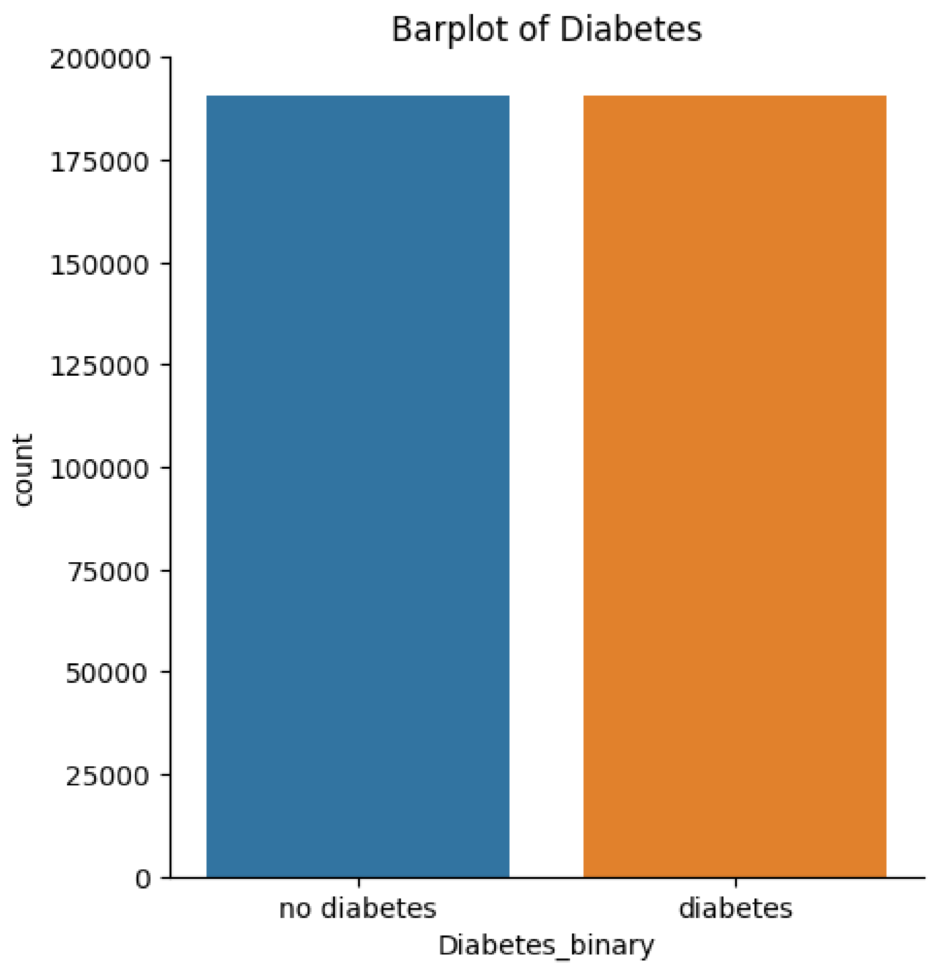
This code creates a categorical plot using Seaborn's catplot. It visualizes the count of instances for each category ('0' for no diabetes and '1' for diabetes) in the 'Diabetes_binary' column. The plot is displayed as a bar chart with custom labels for the x-axis ticks ('no diabetes' and 'diabetes'). The title of the plot is set as "Barplot of Diabetes."

In [39]:

```
# Separate majority and minority classes
df_majority = df[df.Diabetes_binary== 0]
df_minority = df[df.Diabetes_binary== 1]

# Upsample minority class
df_minority_upsampled = resample(df_minority,
                                 replace=True,     # sample with replacement
                                 n_samples=len(df_majority),    # to match majority class
                                 random_state=123) # reproducible results

# Combine majority class with upsampled minority class
df_upsampled = pd.concat([df_majority, df_minority_upsampled])

# Display new class counts
sns.catplot(x="Diabetes_binary", kind='count', data=df_upsampled)
plt.xticks([0,1],['no diabetes','diabetes'])
plt.title('Barplot of Diabetes')
```

Out[39]:

```
Text(0.5, 1.0, 'Barplot of Diabetes')
```



Explanation

The code performs the upsampling of the minority class in the 'Diabetes_binary' column to address class imbalance.

df_majority and df_minority: The DataFrame is split into two DataFrames based on the values in the 'Diabetes_binary' column. df_majority contains instances where 'Diabetes_binary' is 0 (no diabetes), and df_minority contains instances where 'Diabetes_binary' is 1 (diabetes).

df_minority_upsampled: The minority class is upsampled using the resample function from scikit-learn. Upsampling involves randomly duplicating instances from the minority class to match the number of instances in the majority class. The n_samples parameter is set to the length of the majority class to balance the class distribution.

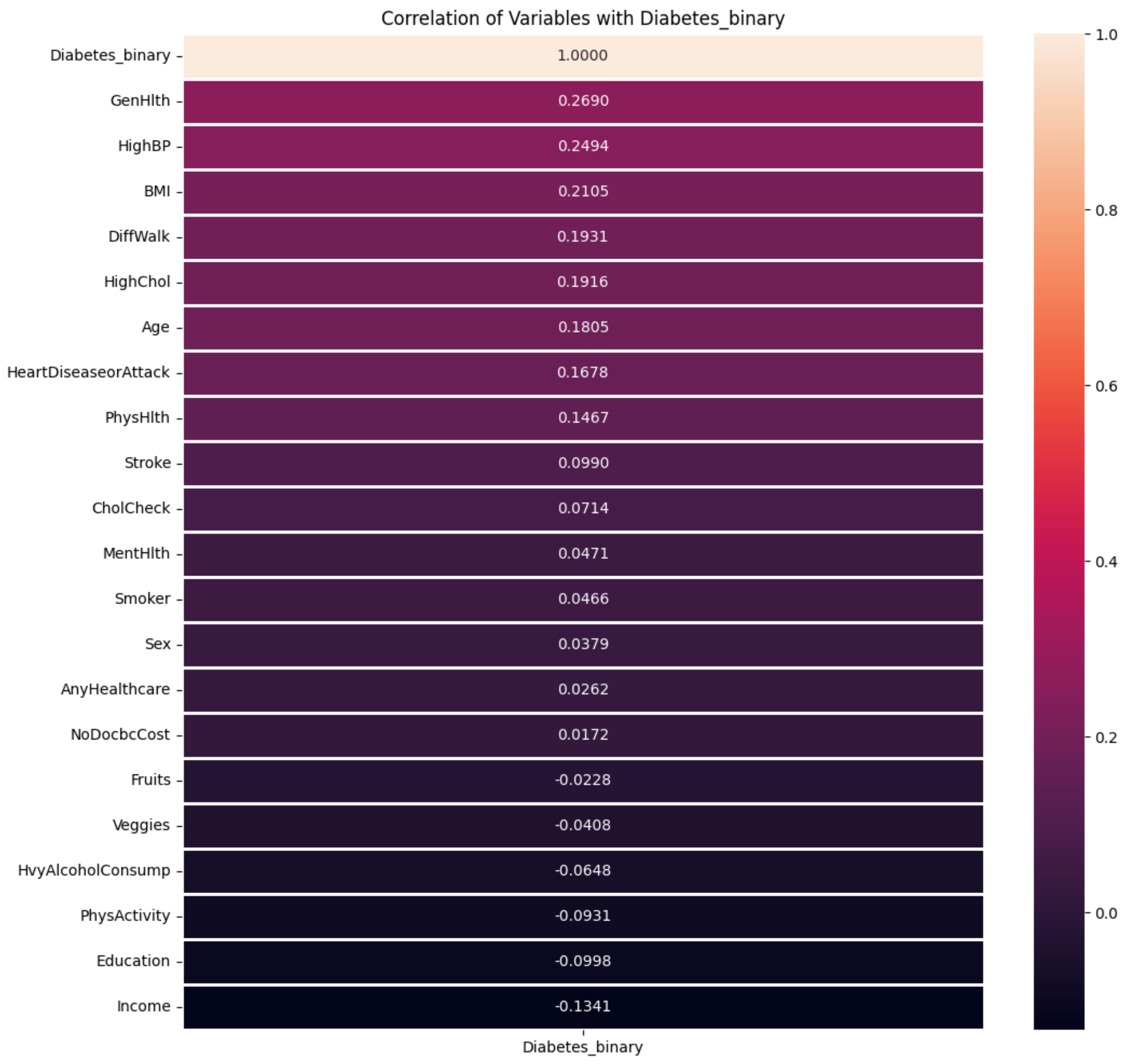
df_upsampled: The upsampled minority class is combined with the original majority class using pd.concat to create a new DataFrame called df_upsampled. Now, the classes are balanced in this new DataFrame.

The justification for upsampling the minority class is to prevent class imbalance, which can negatively impact the performance of machine learning models, especially those sensitive to class proportions. By creating a balanced dataset, the model can learn from both classes equally, potentially leading to better generalization and predictive

4.5) Remove useless attribute

In [40]:

```
# Set the 'event_type' column as the target variable
target_variable = 'Diabetes_binary'

# set the plot size
plt.figure(figsize=(12, 12))

matrix = df.corr()[[target_variable]].round(4)

# sorted the correlation value
sorted_matrix = matrix.sort_values(by=target_variable, ascending=False)
sns.heatmap(sorted_matrix, annot=True, fmt=".4f", linewidths=2)

# set the heatmap title
plt.title(f"Correlation of Variables with {target_variable}")

plt.show()
```



Explanation

The code generates a correlation matrix heatmap to visualize the correlations between different variables and the target variable, which is 'Diabetes_binary.' Each cell in the heatmap represents the correlation coefficient between the target variable and the corresponding variable from the dataset.

The values are rounded to four decimal places and displayed using color intensity, where brighter colors indicate

In [41]:

```
# Calculate the correlation matrix
correlation_matrix = df.corr()

# Filter the columns based on the correlation coefficient
selected_columns = correlation_matrix[correlation_matrix[target_variable].round(4) > 0.04].inc

# Remove the target variable from the selected columns
selected_columns = selected_columns.drop(target_variable)

# Add the target variable to the selected columns
selected_columns = selected_columns.insert(0, target_variable)

# Create a new dataframe with the selected columns
df_filtered = df[selected_columns]

# Overwrite the existing dataframe with the filtered dataframe
df = df_filtered

# Display the updated dataframe
print(df_filtered)
```

```
        Diabetes_binary  HighBP  HighChol  CholCheck   BMI  Smoker  Stroke  \
0                     0       1         1          1  40.0     1.0       0
1                     0       0         0          0  25.0     1.0       0
2                     0       1         1          1  28.0     0.0       0
3                     0       1         0          1  27.0     0.0       0
4                     0       1         1          1  24.0     0.0       0
...                 ...     ...       ...        ...   ...     ...     ...
229466                0       0         0          1  27.0     0.0       0
229468                1       1         1          1  18.0     0.0       0
229469                0       0         0          1  28.0     0.0       0
229470                0       1         0          1  23.0     0.0       0
229471                1       1         1          1  25.0     0.0       0

        HeartDiseaseorAttack  GenHlth  MentHlth  PhysHlth  DiffWalk   Age
0                          0        5        18        15         1   9.0
1                          0        3         0         0         0   7.0
2                          0        5        30        30         1   9.0
3                          0        2         0         0         0  11.0
4                          0        2         3         0         0  11.0
...                      ...      ...       ...       ...       ...   ...
229466                     0        1         0         0         0   3.0
229468                     0        4         0         0         1  11.0
229469                     0        1         0         0         0   2.0
229470                     0        3         0         0         0   7.0
229471                     1        2         0         0         0   9.0

[223834 rows x 13 columns]
```

Explanation

The code calculates the correlation matrix for the DataFrame 'df' using the 'corr()' function. It then filters the columns based on the correlation coefficient with the target variable 'Diabetes_binary.' Only columns with a correlation coefficient greater than 0.04 are selected.

The target variable is added to the list of selected columns at the beginning. The DataFrame 'df_filtered' is created using these selected columns, and then 'df' is updated with the new filtered DataFrame. The updated DataFrame 'df' contains only the columns that have a significant correlation with the target variable, which can be useful for building a predictive model.

5) Machine Learning Model

Explanation

In this diabetes prediction project, we aim to utilize three classification algorithms - Decision Tree (DT), Naive Bayes (NB), and K-Nearest Neighbors (KNN) - to predict whether individuals have diabetes or not based on their health-related attributes.

The classification objective is to create accurate and efficient models that can effectively classify patients into two distinct categories, "diabetes" and "non-diabetes," using historical health data. The Decision Tree algorithm is chosen for its ability to create interpretable and non-linear models, while Naive Bayes is selected for its simplicity and capability to handle various feature types.

Lastly, K-Nearest Neighbors is included for its non-parametric nature and flexibility in capturing patterns in complex and noisy medical datasets. These models aim to provide valuable insights and support medical professionals in making well-informed decisions for patient care.

5.1) KNN Model

In [42]:

```
# storing the input values in the X variable
X = df.iloc[:,1:].values
# storing all the ouputs in y variable
y = df.iloc[:,0].values

# Split in training and testing
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)

# applying standard scale method
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()

# scaling training and testing data set
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

# importing KNN algorithm
from sklearn.neighbors import KNeighborsClassifier

# K value set to be 5
KNN_classifier = KNeighborsClassifier(n_neighbors=5)

# model training
KNN_classifier.fit(X_train,y_train)

# testing the model
y_pred= KNN_classifier.predict(X_test)

# importing accuracy_score
from sklearn.metrics import accuracy_score

# printing accuracy
knn_accuracy = accuracy_score(y_test,y_pred)
print(accuracy_score(y_test,y_pred))
```

```
0.8368304269482212
```

Explanation

The code builds a K-Nearest Neighbors (KNN) classifier to predict the 'Diabetes_binary' target variable based on the input features. It first separates the input features ('X') and the target variable ('y'). The data is then split into training and testing sets, where 30% of the data is used for testing. The input features are scaled using the StandardScaler to ensure consistent and meaningful comparisons.

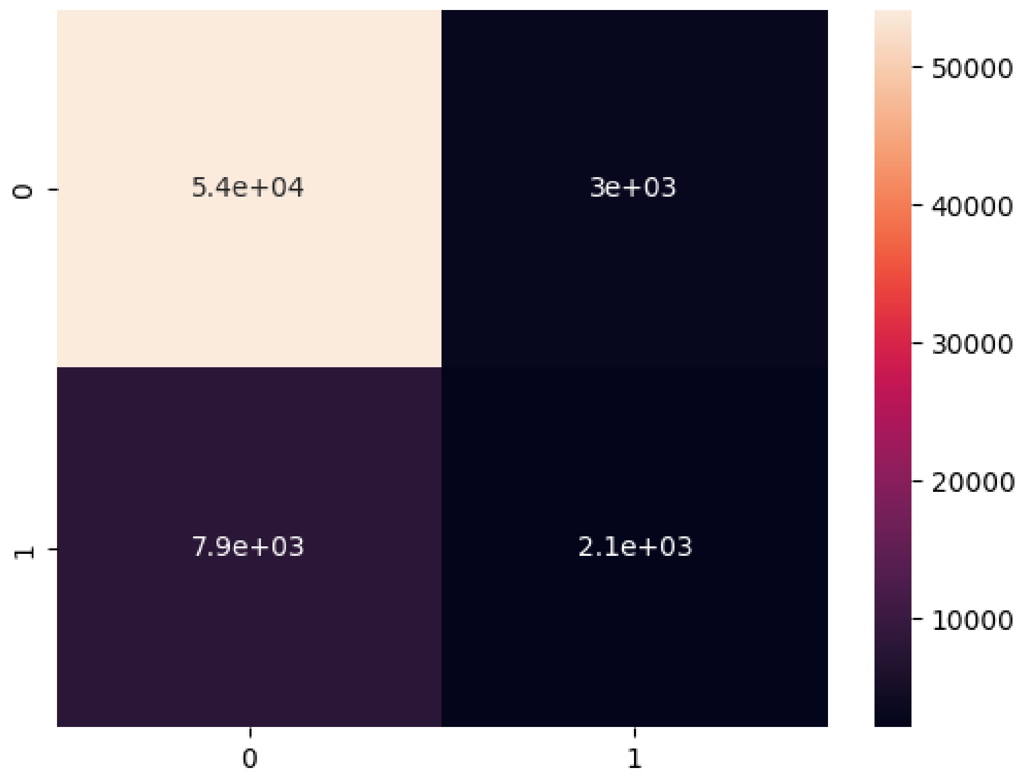
Next, the KNN classifier is instantiated with 'n_neighbors' set to 5 and trained on the scaled training data. Once the model is trained, it is used to predict the target variable on the scaled testing data ('X_test'), and the predicted values are stored in 'y_pred'.

Finally, the accuracy of the KNN model is evaluated using the accuracy_score function, which calculates the percentage of correctly predicted target values in the testing set. The accuracy score is then printed to the console, representing the performance of the KNN classifier in predicting the 'Diabetes_binary' values.

In [43]:

```
# Making the Confusion Matrix
from sklearn.metrics import confusion_matrix
# providing actual and predicted values
cm = confusion_matrix(y_test, y_pred)
# If True, write the data value in each cell
sns.heatmap(cm,annot=True)

# saving confusion matrix in png form
#plt.savefig('confusion_Matrix.png')

print(cm)
```

```
[[54133  3027]
 [ 7930  2061]]
```



Explanation

The code creates a confusion matrix using the actual ('y_test') and predicted ('y_pred') values from the KNN classifier. It displays the number of true positive (TP), true negative (TN), false positive (FP), and false negative (FN) predictions made by the model.

The heatmap is created using Seaborn's 'heatmap' function, with the confusion matrix 'cm' as the data. The 'annot=True' argument allows the actual values to be displayed in each cell of the heatmap for better visualization.

The confusion matrix shows the following layout:
[[TN FP]
[FN TP]]

- TN (True Negative) represents the number of correctly predicted negative instances (correctly predicted as 'no diabetes').
- FP (False Positive) represents the number of incorrect predictions of positive instances (incorrectly predicted as 'diabetes').

- FN (False Negative) represents the number of incorrect predictions of negative instances (incorrectly predicted as 'no diabetes').
- TP (True Positive) represents the number of correctly predicted positive instances (correctly predicted as

In [44]:

```
# finding the whole report
from sklearn.metrics import classification_report

# Print the classification report
report = classification_report(y_test, y_pred, digits=3)
print(report)

# Split the report by newlines
lines = report.split('\n')

# Find the line containing the overall metrics
overall_metrics_line = lines[-2]

# Split the line by spaces
metrics = overall_metrics_line.split()

# Extract the precision, recall, and F1-score
knn_precision = float(metrics[2])
knn_recall = float(metrics[3])
knn_f1_score = float(metrics[4])

# Print the overall metrics
print("Overall Precision:", knn_precision)
print("Overall Recall:", knn_recall)
print("Overall F1-score:", knn_f1_score)
```

```
              precision    recall  f1-score   support

           0      0.872     0.947     0.908     57160
           1      0.405     0.206     0.273      9991

    accuracy                          0.837     67151
   macro avg      0.639     0.577     0.591     67151
weighted avg      0.803     0.837     0.814     67151

Overall Precision: 0.803
Overall Recall: 0.837
Overall F1-score: 0.814
```

Explanation

The code calculates the classification report for the KNN classifier's performance on the test data (y_test and y_pred). The classification report provides detailed metrics for each class (diabetes and no diabetes) along with their weighted average, which represents overall metrics.

After obtaining the classification report, the code extracts the precision, recall, and F1-score values from the overall metrics line. These metrics give a summary of the classifier's performance across both classes.

- Precision (Overall Precision): The ability of the model to correctly predict positive instances (diabetes) out of all instances predicted as positive.
- Recall (Overall Recall): The ability of the model to correctly predict positive instances (diabetes) out of all actual positive instances in the test data.
- F1-score (Overall F1-score): The harmonic mean of precision and recall, providing a balanced measure of the classifier's performance.

5.2) Naive Bayes Model

In [45]:

```
#training and testing data
from sklearn.model_selection import train_test_split

#assign test data size 80%(testing)
x_train, x_test, y_train, y_test = train_test_split(X,y, test_size = 0.3, random_state = 0)

#importing standard scaler
from sklearn.preprocessing import StandardScaler

#scalling the input data
sc_x = StandardScaler()
x_train = sc_x.fit_transform(x_train)
x_test = sc_x.fit_transform(x_test)

#training the model using BERNOULLI NAIVE BAYES CLASSIFIER
#import classifier
from sklearn.naive_bayes import BernoulliNB

#initializating the NB
nb_classifier = BernoulliNB()

#training the model
nb_classifier.fit (x_train,y_train)

#testing the model
y_pred = nb_classifier.predict(x_test)

#importing accuracy score
from sklearn.metrics import accuracy_score

#printing the accuracy of the model
nb_accuracy = accuracy_score(y_pred, y_test)
print(nb_accuracy)
```

```
0.812854611249274
```

Explanation

The code above show the process of training and testing a Bernoulli Naive Bayes classifier on the dataset. The data is split into training and testing sets using the train_test_split function, with 70% used for training and 30% for testing.

Before training the model, the input features (x_train and x_test) are standardized using the StandardScaler to bring them to a similar scale. This preprocessing step ensures that each feature contributes equally to the model's performance.

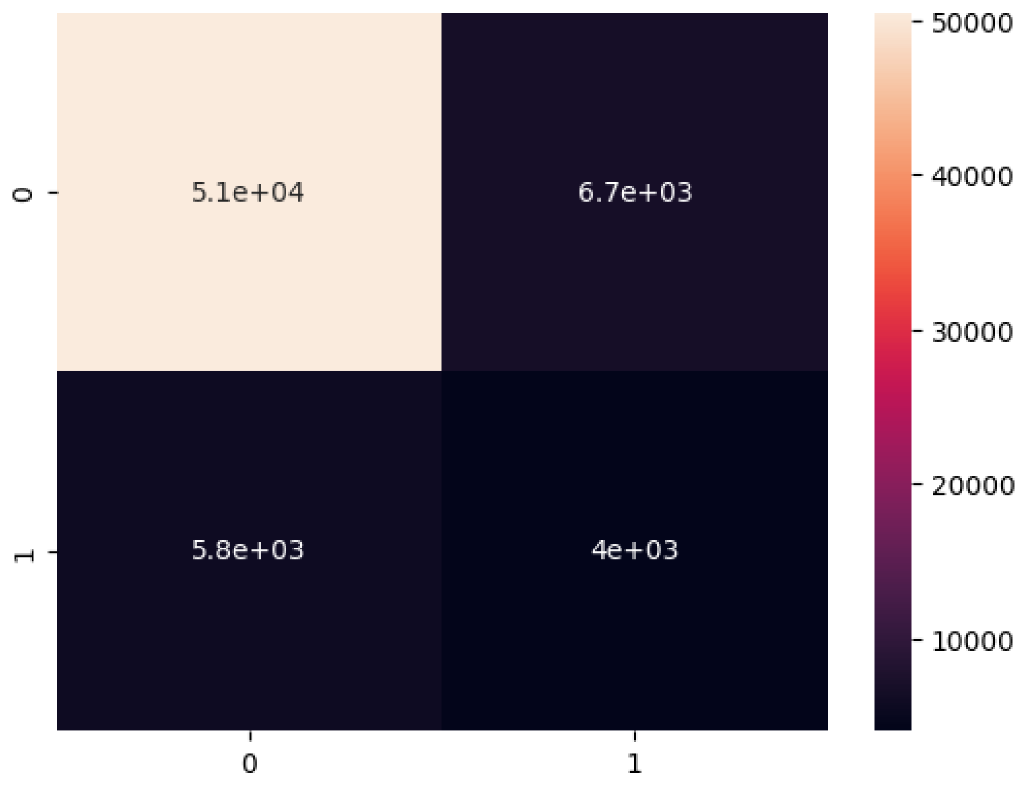
Next, the Bernoulli Naive Bayes classifier is imported from sklearn.naive_bayes. An instance of the classifier is created, and the model is trained using the fit method with the training data (x_train and y_train).

After training, the model is tested on the testing data using the predict method, which returns the predicted output values (y_pred). The accuracy of the model is calculated using accuracy_score from sklearn.metrics, comparing the predicted values with the actual target values (y_test). The accuracy score represents the percentage of correctly predicted instances in the testing data.

In [46]:

```
#evaluation of BERNOULLI NAIVE BAYES CLASSIFIER

#importing the require modules
import seaborn as sns
from sklearn.metrics import confusion_matrix

#passing actual and predicted values
cm = confusion_matrix(y_test, y_pred)

#true write data values in each cell of the matrix
sns.heatmap(cm, annot = True)

print(cm)
```

```
[[50541  6725]
 [ 5842  4043]]
```



In [47]:

```
# finding the whole report
from sklearn.metrics import classification_report

# Print the classification report
report = classification_report(y_test, y_pred, digits=3)
print(report)

# Split the report by newlines
lines = report.split('\n')

# Find the line containing the overall metrics
overall_metrics_line = lines[-2]

# Split the line by spaces
metrics = overall_metrics_line.split()

# Extract the precision, recall, and F1-score
nb_precision = float(metrics[2])
nb_recall = float(metrics[3])
nb_f1_score = float(metrics[4])

# Print the overall metrics
print("Overall Precision:", nb_precision)
print("Overall Recall:", nb_recall)
print("Overall F1-score:", nb_f1_score)
```

```
              precision    recall  f1-score   support

           0      0.896     0.883     0.889     57266
           1      0.375     0.409     0.392      9885

    accuracy                          0.813     67151
   macro avg      0.636     0.646     0.640     67151
weighted avg      0.820     0.813     0.816     67151

Overall Precision: 0.82
Overall Recall: 0.813
Overall F1-score: 0.816
```

5.3) Decision Tree Model

In [48]:

```
# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,random_state=1) # 70%

# Create Decision Tree classifer object
from sklearn.tree import DecisionTreeClassifier # Import Decision Tree Classifier
dt_classifier = DecisionTreeClassifier()

#scalling the input data
sc_x = StandardScaler()
x_train = sc_x.fit_transform(x_train)
x_test = sc_x.fit_transform(x_test)

# Train Decision Tree Classifer
dt_classifier = dt_classifier.fit(X_train,y_train)

#Predict the response for test dataset
y_pred = dt_classifier.predict(X_test)

# importing accuracy_score
from sklearn.metrics import accuracy_score

# printing accuracy
dt_accuracy = accuracy_score(y_test,y_pred)
print(accuracy_score(y_test,y_pred))
```

```
0.8116185909368439
```

Explanation

The code uses the train_test_split function from sklearn.model_selection to split the dataset into training set (70%) and test set (30%). The X variable contains the input features, and the y variable contains the target variable (Diabetes_binary).

Then, the code creates an instance of the DecisionTreeClassifier class from sklearn.tree. It then calls the fit method to train the classifier using the training data (X_train and y_train).

Next, the StandardScaler from sklearn.preprocessing is used to standardize the features in the training and test datasets. This step ensures that all features have the same scale and prevents any feature from dominating the learning algorithm.

The code uses the trained Decision Tree Classifier (dt_classifier) to predict the target values (y_pred) for the test data (X_test).

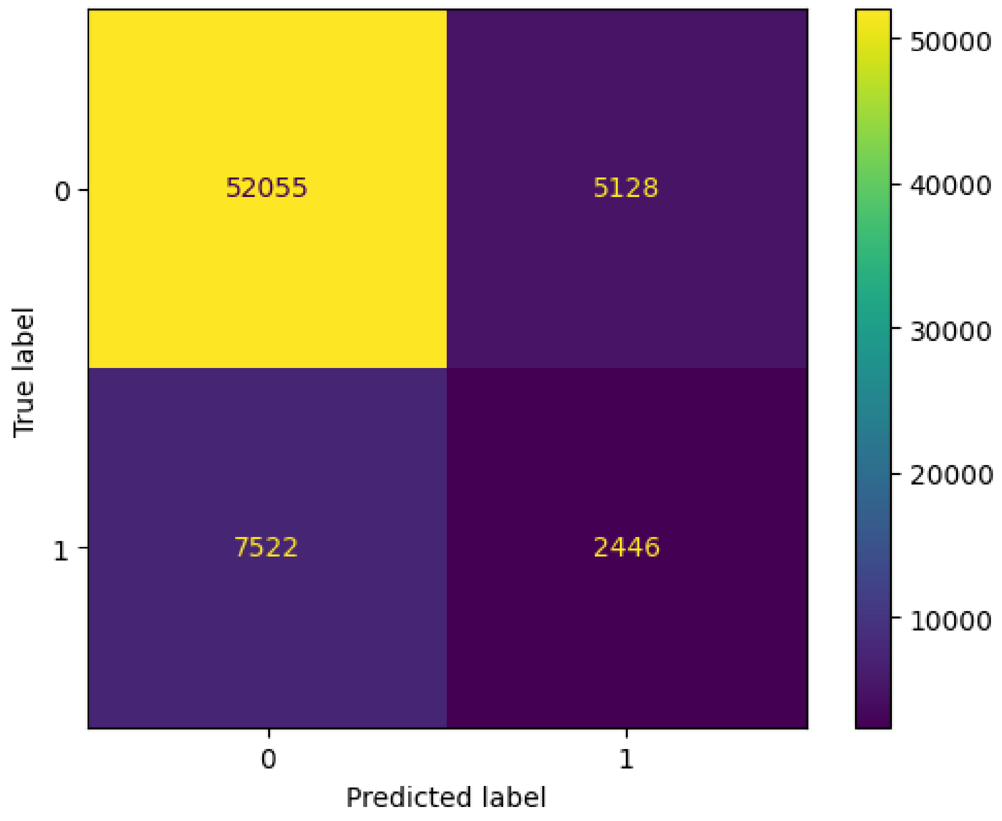
Lastly, accuracy_score function from sklearn.metrics is used to calculate the accuracy of the model by comparing the predicted values (y_pred) with the actual target values (y_test).

In [49]:

```
# Calculate the confusion matrix
cm = confusion_matrix(y_test, y_pred,labels=dt_classifier.classes_)

# Plot the confusion matrix with labels
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=dt_classifier.classes_)
disp.plot()
```

Out[49]:

```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x20f495b09a0
>
```



In [50]:

```
# finding the whole report
from sklearn.metrics import classification_report

# Print the classification report
report = classification_report(y_test, y_pred, digits=4)
print(report)

# Split the report by newlines
lines = report.split('\n')

# Find the line containing the overall metrics
overall_metrics_line = lines[-2]

# Split the line by spaces
metrics = overall_metrics_line.split()

# Extract the precision, recall, and F1-score
dt_precision = float(metrics[2])
dt_recall = float(metrics[3])
dt_f1_score = float(metrics[4])

# Print the overall metrics
print("Overall Precision:", dt_precision)
print("Overall Recall:", dt_recall)
print("Overall F1-score:", dt_f1_score)
```

```
              precision    recall  f1-score   support

           0     0.8737    0.9103    0.8917     57183
           1     0.3229    0.2454    0.2789      9968

    accuracy                         0.8116     67151
   macro avg     0.5983    0.5779    0.5853     67151
weighted avg     0.7920    0.8116    0.8007     67151

Overall Precision: 0.792
Overall Recall: 0.8116
Overall F1-score: 0.8007
```

6) Deploy Model

In [51]:

```
import pickle

# Dictionary to store model results
model_results = {}

if dt_accuracy > nb_accuracy and dt_accuracy > knn_accuracy:
    # Save the Decision model
    model_name = 'Decision Tree'
    model = dt_classifier
    accuracy = dt_accuracy
    precision = dt_precision
    f1score = dt_f1_score
    recall = dt_recall
    print("Decision model saved with accuracy:", decision_accuracy)
elif nb_accuracy > dt_accuracy and nb_accuracy > knn_accuracy:
    # Save the Naive Bayes model
    model_name = 'Naive Bayes'
    model = nb_classifier
    accuracy = nb_accuracy
    precision = nb_precision
    f1score = nb_f1_score
    recall = nb_recall
    print("Naive Bayes model saved with accuracy:", nb_accuracy)
else:
    # Save the KNN model
    model_name = 'KNN'
    model = KNN_classifier
    accuracy = knn_accuracy
    precision = knn_precision
    f1score = knn_f1_score
    recall = knn_recall
    print("KNN model saved with accuracy:", knn_accuracy)

# Save the model
filename = 'model.sav'
pickle.dump(model, open(filename, 'wb'))

# Store model results in dictionary
model_results = {
    'accuracy': accuracy,
    'precision': precision,
    'f1score': f1score,
    'recall': recall,
    'model_name': model_name,
    'knn': {
        'accuracy': knn_accuracy,
        'precision': knn_precision,
        'f1score': knn_f1_score,
        'recall': knn_recall,
    },
    'dt': {
        'accuracy': dt_accuracy,
        'precision': dt_precision,
        'f1score': dt_f1_score,
        'recall': dt_recall,
    },
    'nb': {
        'accuracy': nb_accuracy,
        'precision': nb_precision,
        'f1score': nb_f1_score,
        'recall': nb_recall,
    }
}

# Save accuracy and precision separately
pickle.dump(model_results, open('model_results.sav', 'wb'))

# Save individual models
pickle.dump(nb_classifier, open('NB.sav', 'wb'))
pickle.dump(dt_classifier, open('DT.sav', 'wb'))
pickle.dump(KNN_classifier, open('KNN.sav', 'wb'))
```

```
KNN model saved with accuracy: 0.8368304269482212
```

Explanation

The code above is to save the best-performing model among the Decision Tree Classifier, Naive Bayes Classifier, and K-Nearest Neighbors (KNN) Classifier, along with their respective accuracy, precision, F1 score, and recall metrics.

First, the script compares the accuracy of the three models and determines which one has the highest accuracy. Based on this comparison, it selects the best model and saves it as a binary file using the pickle.dump() function.

The script also creates a dictionary called model_results to store the performance metrics of all three models. It saves the accuracy, precision, F1 score, and recall for each model in this dictionary.

After saving the best model and model results, it separately saves each individual model (Naive Bayes, Decision Tree, and KNN) in binary files using pickle.dump().

The saved model and metrics can be loaded later for further analysis, comparison, or deployment in other applications.

In [ ]:

```

```