VA Diabetes Overview

Problem:

Veterans who receive care in Veterans Health Administration experience and suffer from a higher burden of obesity and diabetes compared to non-Veteran populations. The human and economic costs are staggering as reflected in the recent 2020 Obesity VA/DoD Clinical Practice Guideline Management of Adult Overweight and Obesity and the newly revised 2023 VA/DoD Clinical Practice Guideline Management of Type 2 Diabetes Mellitus. Most patients with diabetes mellitus are asymptomatic, which leads to delayed and more complex treatment.

Research Question:

Will empowering the veteran with knowledge and simple assessment promote healthier lifestyle choices that address and mitigate risk factors associated with T2DM?

Research Demographic - Veterans with BMI >= 25

Outcomes we hope to achieve over a 6 - 12 month follow up:

1. Weight loss: Evaluate percent of patients that achieved 5%; Evaluate percent of patients that achieved 10%
2. HbA1c reduction: Evaluate percent of patients that achieved HbA1c less than 6.5%; Evaluate percent of patients that reduced HbA1c by more than 1%

Objective:

The prevalence of Type 2 Diabetes Mellitus (T2DM) among veterans is a multifaceted issue. Environmental, genetic and lifestyle interactions have not been fully explored as predeterminants of T2DM. Glycated Hemoglobin (HbA1c) is one of the primary tools to diagnose T2DM. However, often times this is only monitored effectively after the onset of the disease. This research will develop a classification model to proactively identify veterans at high risk for T2DM, providing risk scores that enable earlier interventions and potentially prevent or delay disease development.

Imports

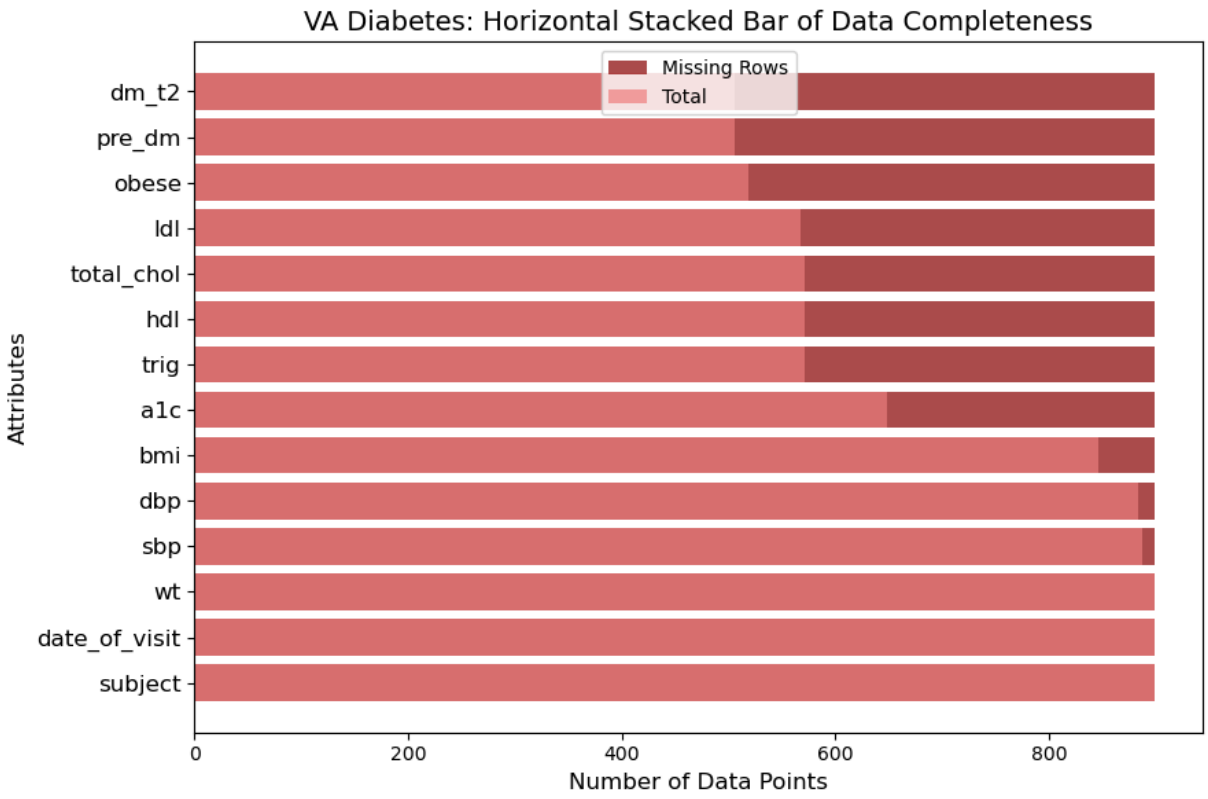
In [ ]:
```
# Package imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import re
```

```
%matplotlib inline
import warnings
warnings.filterwarnings('ignore')
```

In [ ]:
```
import os

# Retrieve the 'Projects' environment variable
base_dir = os.environ.get('Projects', '/Users/ben/Projects')

# Construct the path to the Excel file
file_path = os.path.join(base_dir, 'VA_diab_review', 'DB_raw.xlsx')
```

In [ ]:
```
# Loading xlsx file into a dataframe
df = (
    pd.read_excel(file_path, sheet_name = 'va_diabetes')
    # Dropping columns with 0 non-null count
    .dropna(axis=1, how='all')
    # Lowercasing and replacing spaces with underscores and remove leading s
    .rename(columns=lambda x: x.strip().lower().replace(' ', '_').replace('-
)
```

In [ ]:
```
# Calculate non-missing and missing data points
non_missing_data_na = df.notnull().sum()
missing_data_counts_na = df.isnull().sum()

# Update the total number of entries after dropping rows
total_entries_na_updated = len(df)

# Sorting the columns in ascending order
sorted_columns = missing_data_counts_na.sort_values(ascending=True).index

# Create the horizontal stacked bar graph
plt.figure(figsize=(9, 6))
plt.barh(sorted_columns, total_entries_na_updated, label='Missing Rows', col
plt.barh(sorted_columns, non_missing_data_na[sorted_columns], label='Total',

plt.ylabel('Attributes', fontsize=12)
plt.xlabel('Number of Data Points', fontsize=12)
plt.title('VA Diabetes: Horizontal Stacked Bar of Data Completeness', fontsi
plt.legend(loc='upper center')
plt.tick_params(axis='y', labelsize=12)
plt.tight_layout()
plt.show()
```



In [ ]:
```
df.info()
df.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 899 entries, 0 to 898
Data columns (total 14 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   subject        899 non-null    int64
 1   date_of_visit  899 non-null    datetime64[ns]
 2   wt             899 non-null    int64
 3   sbp            887 non-null    float64
 4   dbp            884 non-null    float64
 5   bmi            846 non-null    float64
 6   obese          518 non-null    object
 7   pre_dm         506 non-null    object
 8   dm_t2          506 non-null    object
 9   a1c            649 non-null    object
 10  trig           571 non-null    float64
 11  hdl            571 non-null    object
 12  total_chol     571 non-null    float64
 13  ldl            568 non-null    object
dtypes: datetime64[ns](1), float64(5), int64(2), object(6)
memory usage: 98.5+ KB
```

Out[ ]:

	subject	date_of_visit	wt	sbp	dbp	bmi	obese	pre_dm	dm_t2	a1c	trig
0	58	2023-04-21	256	150.0	101.0	36.8	YES	NO	NO	4.4	134.0
1	213	2023-12-22	245	136.0	68.0	29.9	NaN	NaN	NaN	4.5	231.0
2	369	2023-08-31	316	NaN	NaN	43.0	N	NaN	NaN	4.5	285.0
3	465	2023-11-13	321	147.0	78.0	43.6	Y	N	N	4.5	285.0
4	102	2023-05-04	212	127.0	87.0	32.3	YES	NO	NO	4.6	140.0

Cleaning & Transformation

Research Audit

```
1. drop any nulls subsetting for ['date_of_visit', 'wt',
'a1c']
2. create visit category for initial and followup to compare
(address research questions)
3. non-numeric character check necessary for num dtypes that
imported as object
4. laboratory nulls will not be imputed / outlier check
(missing bmi impute by creating ht)
5. a1c is gold standard for diagnoses; create function to
label encode for binary, multi-class / pre_dm and dm_t2 can
be dropped
6. create blood_pressure feature as ordinal EDA / potential
drop
7. create body_type instead of obese from bmi for EDA /
potential drop

 HbA1c Target Labels

    Healthy: Below 5.7%
    Prediabetes: 5.7% to 6.4%
    Diabetes: 6.5% or higher

blood_pressure

    Normal: SBP is less than 120 and DBP is less than 80.
    Elevated: SBP is between 120 and 129 and DBP is less than
80.
    Hypertension Stage 1: SBP is between 130 and 139 or DBP
is between 80 and 89.
    Hypertension Stage 2: SBP is 140 or higher or DBP is 90
or higher.
    Hypertensive Crisis: SBP is above 180 or DBP is above
120.
    UNDETERMINED: Any blood pressure reading that does not
fit the above categories
```

body_type
   Underweight: BMI less than 18.5
   Normal weight: BMI 18.5 to 24.9
   Overweight: BMI 25 to 29.9
   Obesity: BMI 30 and over

Pre-process and Model

8. date_of_visit, subject can be dropped before modeling
9. split dataframe into train/test
10. standarize and select classification models for eval

In [ ]:
```
df = df.dropna(subset=['date_of_visit', 'wt', 'a1c'])  # Drop rows with any
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 649 entries, 0 to 648
Data columns (total 14 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   subject        649 non-null    int64
 1   date_of_visit  649 non-null    datetime64[ns]
 2   wt             649 non-null    int64
 3   sbp            644 non-null    float64
 4   dbp            643 non-null    float64
 5   bmi            632 non-null    float64
 6   obese          502 non-null    object
 7   pre_dm         498 non-null    object
 8   dm_t2          498 non-null    object
 9   a1c            649 non-null    object
 10  trig           541 non-null    float64
 11  hdl            541 non-null    object
 12  total_chol     541 non-null    float64
 13  ldl            538 non-null    object
dtypes: datetime64[ns](1), float64(5), int64(2), object(6)
memory usage: 76.1+ KB
```

In [ ]:
```
# Create new column for visit type (initial or followup)

df_sorted = df.sort_values(by=['subject', 'date_of_visit'])

# Function to label visits as 'initial' or 'follow_up', and drop middle visi
def label_visits(group):
    if len(group) > 2:  # If there are more than 2 visits, drop the middle c
        return pd.concat([group.head(1).assign(visit='initial'), group.tail(
    elif len(group) == 2:  # If there are exactly 2 visits
        return pd.concat([group.head(1).assign(visit='initial'), group.tail(
    else:  # If there is only 1 visit
        return group.assign(visit='initial')

# Apply the function to each group and re-assign to df
df = df_sorted.groupby('subject', as_index=False).apply(label_visits).reset_
```

In [ ]:
```
# Address data types of a1c, hdl, and ldl by removing non-numeric characters
columns_of_interest = ['a1c', 'hdl', 'ldl']
unique_values = {col: df[col].unique() for col in columns_of_interest}

unique_values
```

Out[ ]:
```
{'a1c': array([6.4, 6.3, 6.7, 5.6, 9, 6.5, 5.7, 6.9, 5.5, 5, 6.1, 5.3,
        '5.7(2021)', 6, 5.4, 5.8, 7.2, 5.1, 8.5, 6.8, 5.9, 5.2, 9.2, 6.6,
        4.7, 8.6, 7.7, 4.4, 9.4, 4.9, 7.3, 7.5, 4.8, 7.1, 8.8, 6.2, 9.5,
        10.6, 4.6, 8.3, 9.1, 8.1, 7.8, 8, 7, '7.1 (dex)', 4.5, 7.6,
        '5.6 (5.8)', 9.7, '5.6 (5.7)', '5.4 (5.7)', 9.8, 8.4, 10.7,
        '5.5 (5.7)', '8.5 (CGM 7.2)', '6.6 (CGM)', '5.5(5.9)', '5.3(8.1)',
        '5.3(5.7)', 10.4, '5.3 (5.7)', 10.9, '5.6(5.8)', 13.5, 11.9],
       dtype=object),
 'hdl': array([54, nan, 49, 33, 23, 28, 46, 27, 40, 36, 31, 52, 34, 20, 50,
53,
        44, 56, 68, 58, 45, 29, 51, 35, 38, 41, 42, 47, 39, 37, 'yy', 26,
        48, 43, 25, 32, 76, 59, 61, 60, 55, 57, 30, 22, 62, 78, 69, 66, 73,
        63, 77, 64, 75, 91, 71, 74, 18, 275, 101, 87, 24], dtype=object),
 'ldl': array([22.1, nan, 84.6, 120.3, 53, 25, 130.9, 19, 96, 69, 116, 95,
100,
        97, 104, 115, 119, 137, 35, 152, 121, 101, 66, 157, 94, 165, 153,
        80, 164, 125, 149, 128, 129, 'nc', 113, 93, 61, 114, 33, 102, 134,
        198, 175, 130, 64, 73, 213, 158, 172, 136, 44, 162, 150, 29, 21,
        87, 120, 109, 108, 78, 138, 91, 65, 161, 103, 144, 43, 51, 111,
        156, 63, 107, 126, 90, 83, 173, 187, 123, 74, 147, 140, 88, 191,
        151, 79, 76, 39, 77, 176, 177, 99, 59, 31, 106, 131, 145, 81, 194,
        146, 82, 143, 141, 124, 122, 110, 159, 118, 174, 127, 155, 47, 56,
        75, 142, 112, 89, 0, 163, 62, 224, 105, 206, 86, 98, 168, 67, 132,
        68, 50, 41, 117, 170, 133, 42, 184, 72, 28, 221, 154, 135, 71, 186,
        265, 3, 92, 55, 7.2, 84, 54, 85, 180, 232, 139, 27, 160, 273, 60,
        208, 183, 202, 182, 167, 148, 169, 220, 'NC', 200, 233, 203, 32,
        179, 188], dtype=object)}
```

In [ ]:
```
# Define the function to clean and convert a1c, ldl, hdl dtypes
def clean_and_convert_columns(df, columns):
    for column in columns:
        # Handle text within parentheses and any textual values for hdl and
        df[column] = df[column].apply(lambda x: np.nan if isinstance(x, str)
        # Remove anything within parentheses and convert to float
        df[column] = df[column].astype(str).str.replace(r"\(.*\)", "", regex
    return df

# Apply the function to clean and convert the columns
columns_to_clean = ['a1c', 'hdl', 'ldl']
clean_and_convert_columns(df, columns_to_clean)
```

Out[ ]:

	subject	date_of_visit	wt	sbp	dbp	bmi	obese	pre_dm	dm_t2	a1c	t
0	1	2023-03-27	264	129.0	78.0	39.07	YES	NO	YES	6.4	18
1	1	2023-11-14	265	130.0	78.0	39.20	NaN	NaN	NaN	6.3	N
2	2	2023-03-28	233	130.0	89.0	31.67	YES	NO	YES	6.7	6
3	2	2023-12-22	220	138.0	71.0	29.90	NaN	NaN	NaN	5.6	N
4	3	2023-03-28	230	146.0	94.0	29.59	NO	NO	YES	9.0	33
...	...	...	...	...	...	...	...	...	...	...	
628	515	2024-01-30	266	124.0	91.0	34.20	NaN	NaN	NaN	5.5	N
629	516	2024-01-31	222	139.0	81.0	33.30	y	NaN	NaN	4.7	48
630	517	2024-01-31	376	132.0	84.0	57.20	y	n	n	5.1	10
631	518	2024-01-31	275	138.0	84.0	37.40	y	NaN	NaN	5.6	21
632	519	2024-02-01	242	121.0	77.0	38.60	y	y	n	5.9	6

633 rows × 15 columns

In [ ]:

```python
# Create height column from BMI and weight

def process_height_from_bmi(df):
    """
    Adds a height column to the DataFrame based on BMI and weight.
    - Converts weight from pounds to kilograms.
    - Calculates height in meters from BMI, then converts to centimeters.
    - Sorts DataFrame by 'subject' and 'date_of_visit'.
    - Forward fills the height for each subject.

    Parameters:
    - df: DataFrame with columns 'wt' for weight in pounds, 'bmi' for BMI,
          'subject', and 'date_of_visit'.

    Returns:
    - DataFrame with the new 'ht_cm' column and sorted by 'subject' and 'dat
    """
    # Convert weight from pounds to kilograms for the calculation
    df['wt_kg'] = df['wt'] * 0.453592

    # Calculate height in meters from BMI and convert to centimeters
    df['ht_cm'] = np.sqrt(df['wt_kg'] / df['bmi']) * 100

    # Sort DataFrame by 'subject' and 'date_of_visit' to ensure correct orde
    df.sort_values(by=['subject', 'date_of_visit'], inplace=True)

    # Forward fill the 'ht_cm' for each subject to apply the height to follo
    df['ht_cm'] = df.groupby('subject')['ht_cm'].ffill()

    return df
```

```
# Apply the function to create the 'ht_cm' column
df = process_height_from_bmi(df)
```

In [ ]:
```
# Locate where ht and wt are not null but BMI is missing

pre_filtered_df = df[df['ht_cm'].notnull() & df['wt'].notnull() & df['bmi'].
print("Shape of filtered DataFrame:", pre_filtered_df.shape)
pre_filtered_df.head(17)
```

```
Shape of filtered DataFrame: (9, 17)
```

Out[ ]:

	subject	date_of_visit	wt	sbp	dbp	bmi	obese	pre_dm	dm_t2	a1c	tri
6	4	2023-10-16	169	NaN	NaN	NaN	NaN	NaN	NaN	6.4	230.
48	39	2023-10-13	167	140.0	90.0	NaN	NaN	NaN	NaN	5.9	Na
54	43	2023-09-11	228	140.0	84.0	NaN	NaN	NaN	NaN	5.5	Na
325	248	2023-12-07	207	NaN	NaN	NaN	NaN	NaN	NaN	5.7	196.
378	289	2023-11-06	215	122.0	78.0	NaN	NaN	NaN	NaN	5.7	Na
389	297	2023-10-04	181	116.0	65.0	NaN	NaN	NaN	NaN	5.6	Na
462	354	2023-08-24	341	124.0	63.0	NaN	NaN	NaN	NaN	5.3	134.
511	401	2024-02-06	191	NaN	NaN	NaN	NaN	NaN	NaN	5.6	92.
561	448	2023-12-18	248	NaN	NaN	NaN	NaN	NaN	NaN	5.7	Na

In [ ]:
```
# Impute BMI for missing values

# Convert 'ht_cm' to meters by dividing by 100
pre_filtered_df['ht_m'] = pre_filtered_df['ht_cm'] / 100

# Calculate BMI for these rows
pre_filtered_df['bmi'] = pre_filtered_df['wt_kg'] / (pre_filtered_df['ht_m']

# Update the original DataFrame with the calculated BMI values
df.update(pre_filtered_df['bmi'])

# verify
print("Updated DataFrame with imputed BMI values:")
print(df[['subject', 'date_of_visit', 'wt', 'ht_cm', 'bmi']])
```

```
Updated DataFrame with imputed BMI values:
     subject date_of_visit   wt       ht_cm    bmi
0          1    2023-03-27  264  175.070491  39.07
1          1    2023-11-14  265  175.110665  39.20
2          2    2023-03-28  233  182.678157  31.67
3          2    2023-12-22  220  182.687335  29.90
4          3    2023-03-28  230  187.769102  29.59
..       ...           ...  ...         ...    ...
628      515    2024-01-30  266  187.828054  34.20
629      516    2024-01-31  222  173.894987  33.30
630      517    2024-01-31  376  172.674657  57.20
631      518    2024-01-31  275  182.626266  37.40
632      519    2024-02-01  242  168.634614  38.60

[633 rows x 5 columns]
```

In [ ]:
```
# verify filled BMI values
post_filtered_df = df[df['ht_cm'].notnull() & df['wt'].notnull() & df['bmi']
print("Shape of filtered DataFrame:", post_filtered_df.shape)
```

```
Shape of filtered DataFrame: (0, 17)
```

In [ ]:
```
# Function to create diabetes status
def categorize_diabetes_and_clean(df, a1c_column):
    """
    Categorizes HbA1c levels, adds binary and multi-class labels to the Data
    and drops 'db_stat' and 'status' columns if they exist. Categories are t

    Parameters:
    - df: DataFrame containing a1c data.
    - a1c: The column name in the DataFrame that contains a1c values.

    The function modifies the DataFrame in-place and adds two new categorica
    - db_bin: Binary categorization (0 for Healthy, 1 for Diabetes).
    - db_multi: Multi-class categorization (0 for Healthy, 1 for Pre-Diabete

    Drops 'db_stat' and 'status' columns if present.
    """
    # Binary categorization
    db_bin_cats = pd.Categorical(df[a1c_column].apply(lambda x: 0 if x < 6.5
    df['db_bin'] = db_bin_cats.rename_categories(['Non-Diabetes', 'Diabetes'

    # Multi-class categorization
    db_multi_cats = pd.Categorical(df[a1c_column].apply(lambda x: 0 if x < 5
    df['db_multi'] = db_multi_cats.rename_categories(['Healthy', 'Pre-Diabet

    # Drop 'db_stat' and 'status' columns if they exist
    df.drop(columns=['pre_dm', 'dm_t2'], errors='ignore', inplace=True)

# Apply the function to create a new column for binary and multi-class categ
categorize_diabetes_and_clean(df, 'a1c')
```

In [ ]:
```
# Function for blood pressure categorization
def categorize_blood_pressure_and_clean(df, sbp_column, dbp_column):
    """
    Categorizes blood pressure readings into multi-class labels in the DataF
```

```
    and removes any redundant or irrelevant columns. The categories are trea

    Parameters:
    - df: DataFrame containing blood pressure data.
    - sbp: The column name in the DataFrame that contains systolic blood pre
    - dbp: The column name in the DataFrame that contains diastolic blood pr

    The function modifies the DataFrame in-place and adds a new categorical
    - blood_pressure: Multi-class categorization (Normal, Elevated, Hyperter

    Optionally drops any columns named 'old_bp' and 'previous_bp' if present
    """
    # Define multi-class categorization logic
    def multi_class_bp(sbp, dbp):
        if sbp < 120 and dbp < 80:
            return 'Normal'
        elif 120 <= sbp <= 129 and dbp < 80:
            return 'Elevated'
        elif (130 <= sbp <= 139) or (80 <= dbp <= 89):
            return 'Hypertension Stage 1'
        elif sbp >= 140 or dbp >= 90:
            return 'Hypertension Stage 2'
        elif sbp > 180 or dbp > 120:
            return 'Hypertensive Crisis'
        else:
            return 'Undetermined'

    # Apply categorization
    df['blood_pressure'] = pd.Categorical(df.apply(lambda row: multi_class_b
                                          categories=['Normal', 'Elevated',
                                          ordered=True)

    # Drop columns if they exist
    df.drop(columns=['old_bp', 'previous_bp'], errors='ignore', inplace=True

categorize_blood_pressure_and_clean(df, 'sbp', 'dbp')
```

In [ ]:

```
# Function for BMI categorization
def categorize_bmi_and_clean(df, bmi_column):
    """
    Categorizes BMI values into multi-class labels in the DataFrame,
    and removes any redundant or irrelevant columns. The categories are trea

    Parameters:
    - df: DataFrame containing BMI data.
    - bmi_column: The column name in the DataFrame that contains BMI values.

    The function modifies the DataFrame in-place and adds a new categorical
    - body_type: Multi-class categorization (Underweight, Normal weight, Ove

    Optionally drops any columns named 'old_bmi' and 'previous_bmi' if prese
    """
    # Define multi-class categorization logic
    def multi_class_bmi(bmi):
        if bmi < 18.5:
            return 'Underweight'
```

```
        elif 18.5 <= bmi <= 24.9:
            return 'Normal weight'
        elif 25 <= bmi <= 29.9:
            return 'Overweight'
        else:  # BMI of 30 and over
            return 'Obesity'

    # Apply categorization
    df['body_type'] = pd.Categorical(df[bmi_column].apply(multi_class_bmi),
                                     categories=['Underweight', 'Normal weig
                                     ordered=True)

    # Drop columns if they exist
    df.drop(columns=['old_bmi', 'previous_bmi'], errors='ignore', inplace=Tr

categorize_bmi_and_clean(df, 'bmi')
```

In [ ]:
```
df.drop(columns=['obese','ht_cm', 'wt_kg'], inplace=True) # Drop columns tha
```

In [ ]:
```
# mapping categorical values as ordinal and interpretability

db_multi_mapping = {'Healthy': 0, 'Pre-Diabetes': 1, 'Diabetes': 2}

db_bin_mapping = {'Non-Diabetes': 0, 'Diabetes': 1}

blood_pressure_mapping = {
    'Normal': 0, 'Elevated': 1, 'Hypertension Stage 1': 2,
    'Hypertension Stage 2': 3, 'Hypertensive Crisis': 4, 'Undetermined': 5
}
body_type_mapping = {
    'Underweight': 0, 'Normal weight': 1, 'Overweight': 2, 'Obesity': 3
}

# Apply the mappings
df['blood_pressure_c'] = df['blood_pressure'].map(blood_pressure_mapping)
df['body_type_c'] = df['body_type'].map(body_type_mapping)
df['db_multi_c'] = df['db_multi'].map(db_multi_mapping)
df['db_bin_c'] = df['db_bin'].map(db_bin_mapping)

# Convert these columns to numeric to ensure they are treated as such
df['blood_pressure_c'] = pd.to_numeric(df['blood_pressure_c'], errors='coerc
df['body_type_c'] = pd.to_numeric(df['body_type_c'], errors='coerce')
df['db_multi_c'] = pd.to_numeric(df['db_multi_c'], errors='coerce')
df['db_bin_c'] = pd.to_numeric(df['db_bin_c'], errors='coerce')
```

In [ ]:
```
df_follow = df[df['visit'] == 'follow_up']
df_init = df[df['visit'] == 'initial']
df_init.subject.nunique(), df_follow.subject.nunique()
```

Out[ ]: (504, 129)

In [ ]:
```
# create function for research questions

def create_research_df(data):
    # Filter for initial visits with BMI >= 25
```

```
    initial_visits = data[(data['bmi'] >= 25) & (data['visit'] == 'initial')
    # Filter for follow_up visits
    follow_up_visits = data[data['visit'] == 'follow_up']
    # Combine the two dataframes
    research_df = pd.concat([initial_visits, follow_up_visits], ignore_index
    return research_df

# Apply the function to the original data
df_research = create_research_df(df)
```

In [ ]:
```
# ensure df_research contains subjects with inital with BMI >= 25
df_research = df[df['subject'].isin(df_follow['subject'])]
df_research.subject.nunique()
```

Out[ ]: 129

In [ ]:
```
# drop remaining nulls for df_init / df
df_init.dropna(inplace=True)
df.dropna(inplace=True)
```

Exploratory Data Analysis

1. df_init = all initial visits
2. df_research = research questions on pre-post
3. df = use for feature importance / cor matrix / preprocess for modeling

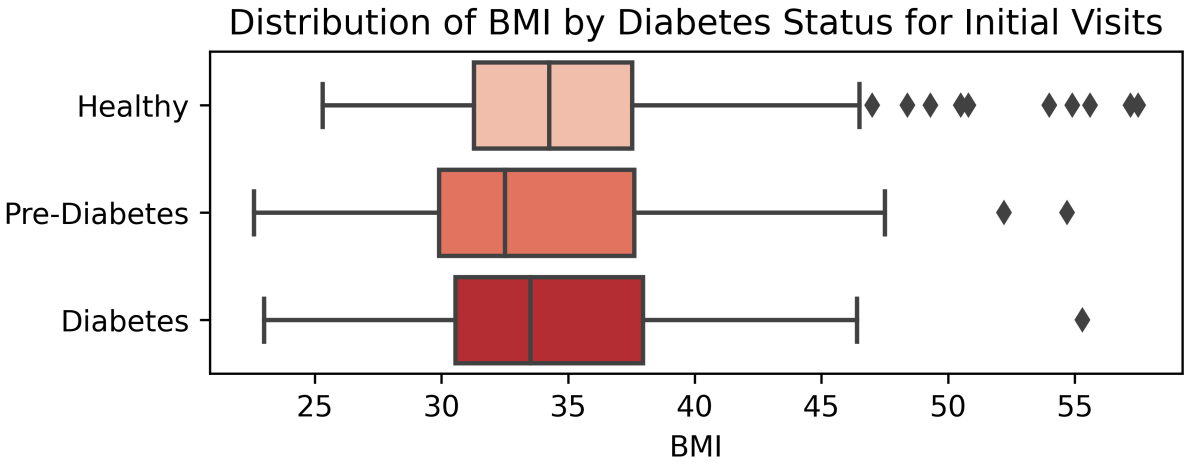
In [ ]:
```
df_init.to_csv('df_init.csv', index=False)
df_research.to_csv('df_research.csv', index=False)
df.to_csv('df.csv', index=False)
```

In [ ]:
```
df_init.drop('date_of_visit', axis=1).describe().T
```

Out[ ]:

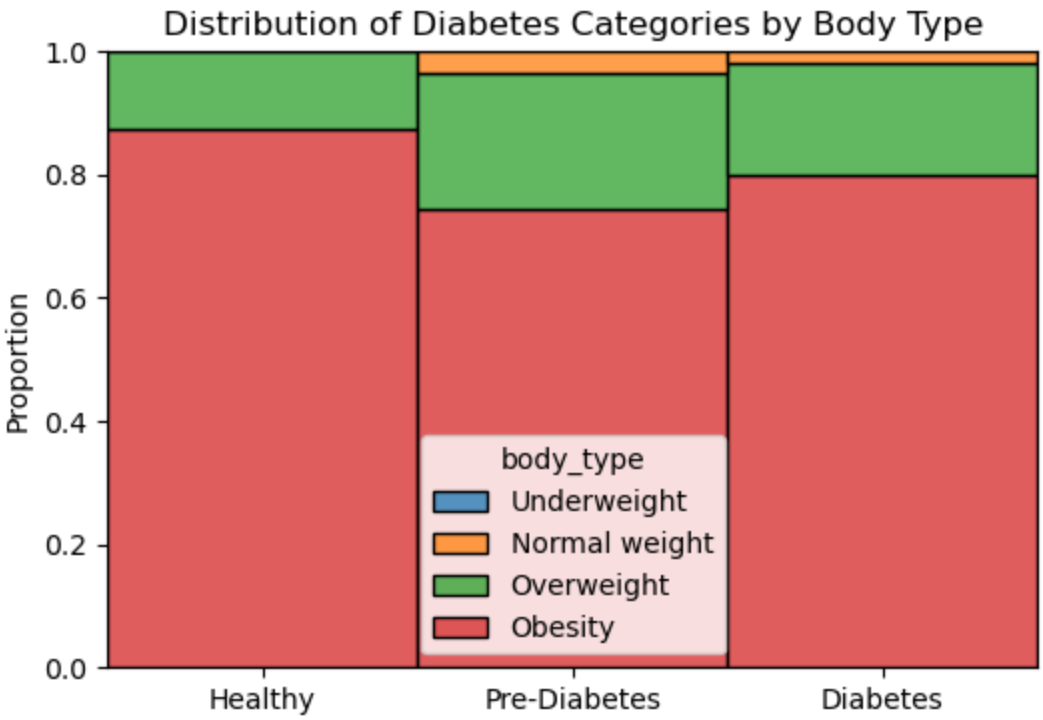
	count	mean	std	min	25%	50%	75%	ma
subject	432.0	248.770833	147.150036	1.0	123.75	244.5	374.500	519.
wt	432.0	243.509259	42.575767	153.0	216.00	240.0	266.250	397.
sbp	432.0	128.314815	13.349069	15.0	121.00	129.0	136.000	200.
dbp	432.0	83.861111	10.513682	13.0	78.00	84.0	89.000	138.
bmi	432.0	34.598704	5.819453	22.6	30.80	33.6	37.600	57.
a1c	432.0	5.810880	0.998489	4.4	5.30	5.6	5.925	13.
trig	432.0	165.824074	82.525573	25.0	103.75	150.0	213.500	512.
hdl	432.0	42.571759	10.705483	22.0	35.00	41.0	48.000	101.
total_chol	432.0	192.321759	42.354001	82.0	163.00	192.0	219.250	344.
ldl	432.0	115.740972	39.948840	7.2	90.00	114.0	141.000	273.
blood_pressure_c	432.0	1.800926	0.862844	0.0	2.00	2.0	2.000	3.
body_type_c	432.0	2.805556	0.429932	1.0	3.00	3.0	3.000	3.
db_multi_c	432.0	0.599537	0.704180	0.0	0.00	0.0	1.000	2.
db_bin_c	432.0	0.127315	0.333712	0.0	0.00	0.0	0.000	1.

In [ ]:
```
# Create the horizontal box plot using seaborn
plt.figure(figsize=(6, 2), dpi=600)
sns.boxplot(y="db_multi", x="bmi", data=df_init, orient="h", palette="Reds")

# Set the plot title and labels
plt.title("Distribution of BMI by Diabetes Status for Initial Visits")
plt.xlabel("BMI")
plt.ylabel("")
plt.show()
```

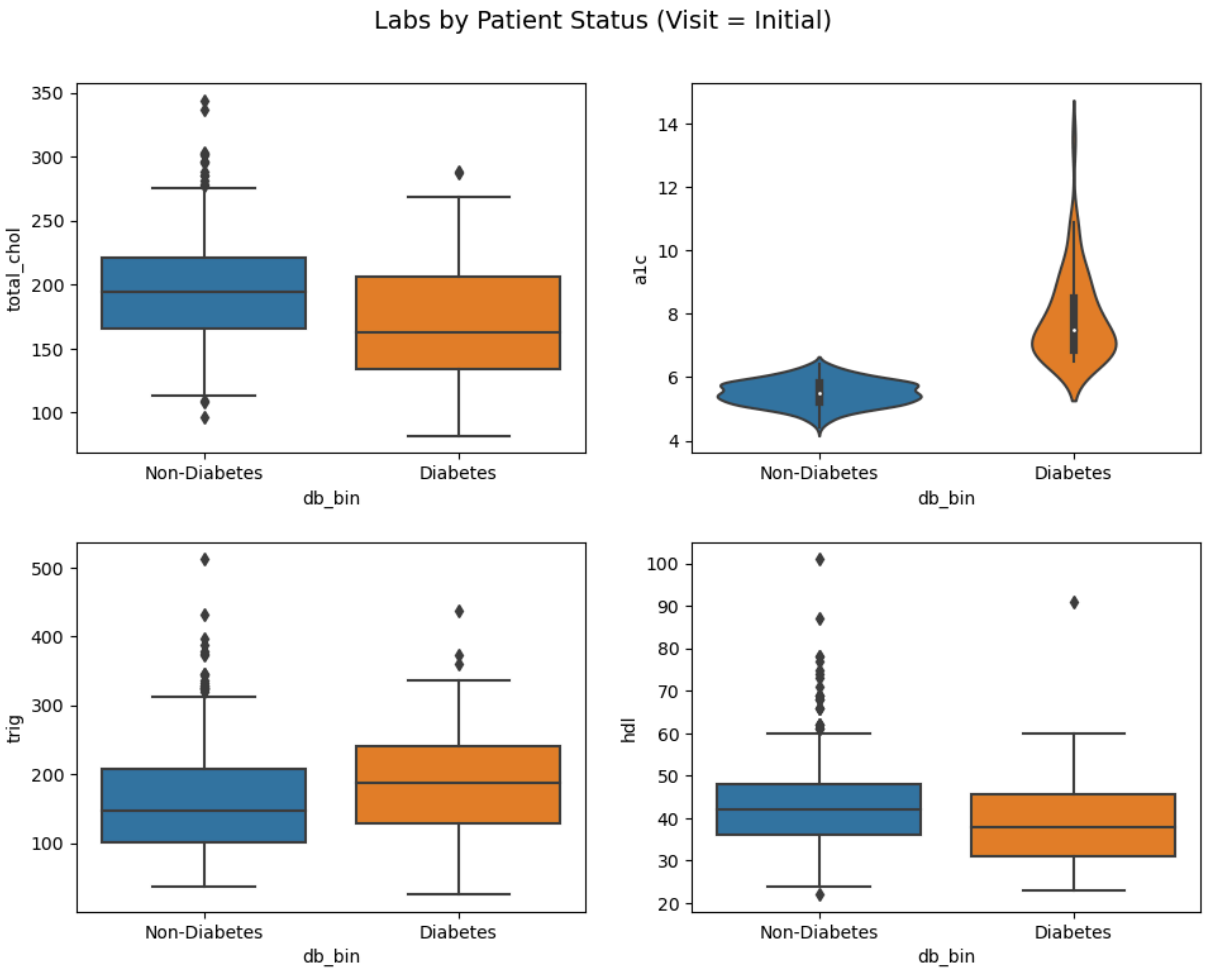


In [ ]:
```
# Create the histogram with frequencies
plt.figure(figsize=(6, 4))
```

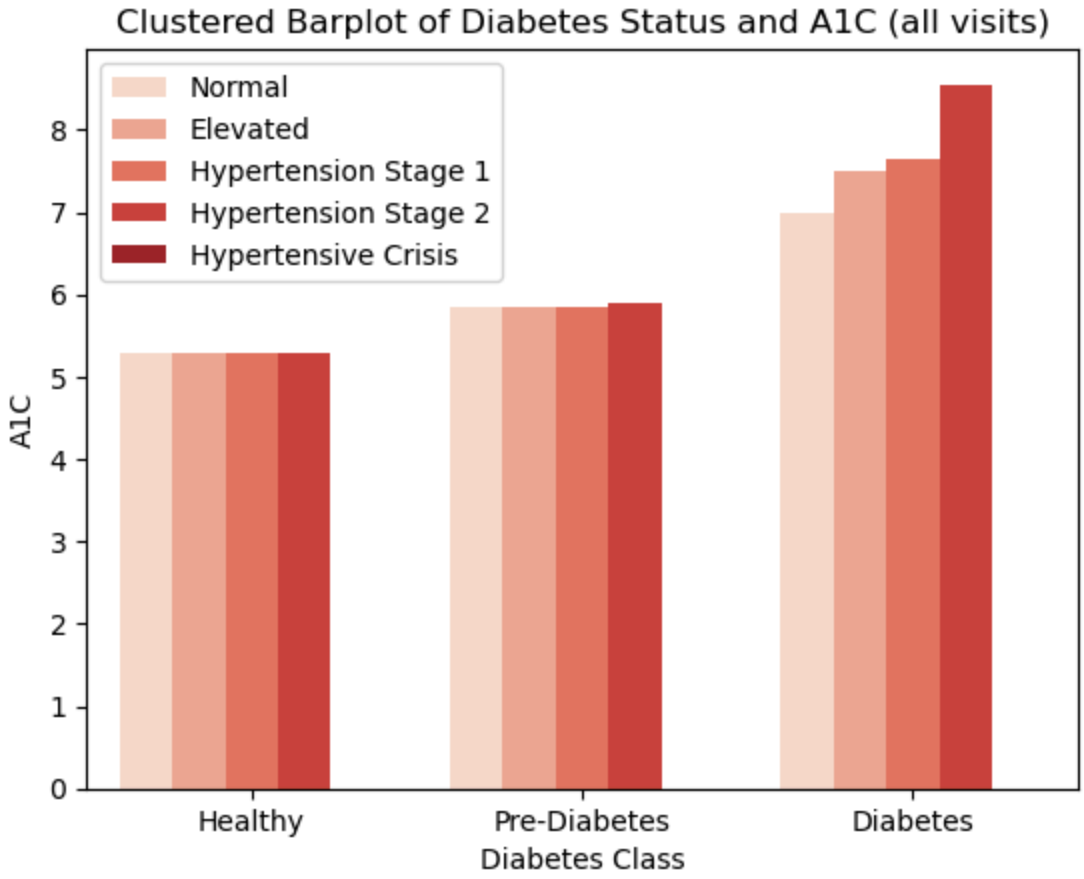
```
sns.histplot(data=df_init, x="db_multi", hue="body_type", multiple="fill", s

# Set the plot title and labels
plt.title("Distribution of Diabetes Categories by Body Type",fontsize=12)
plt.xlabel("")
plt.ylabel("Proportion")
plt.show()
```



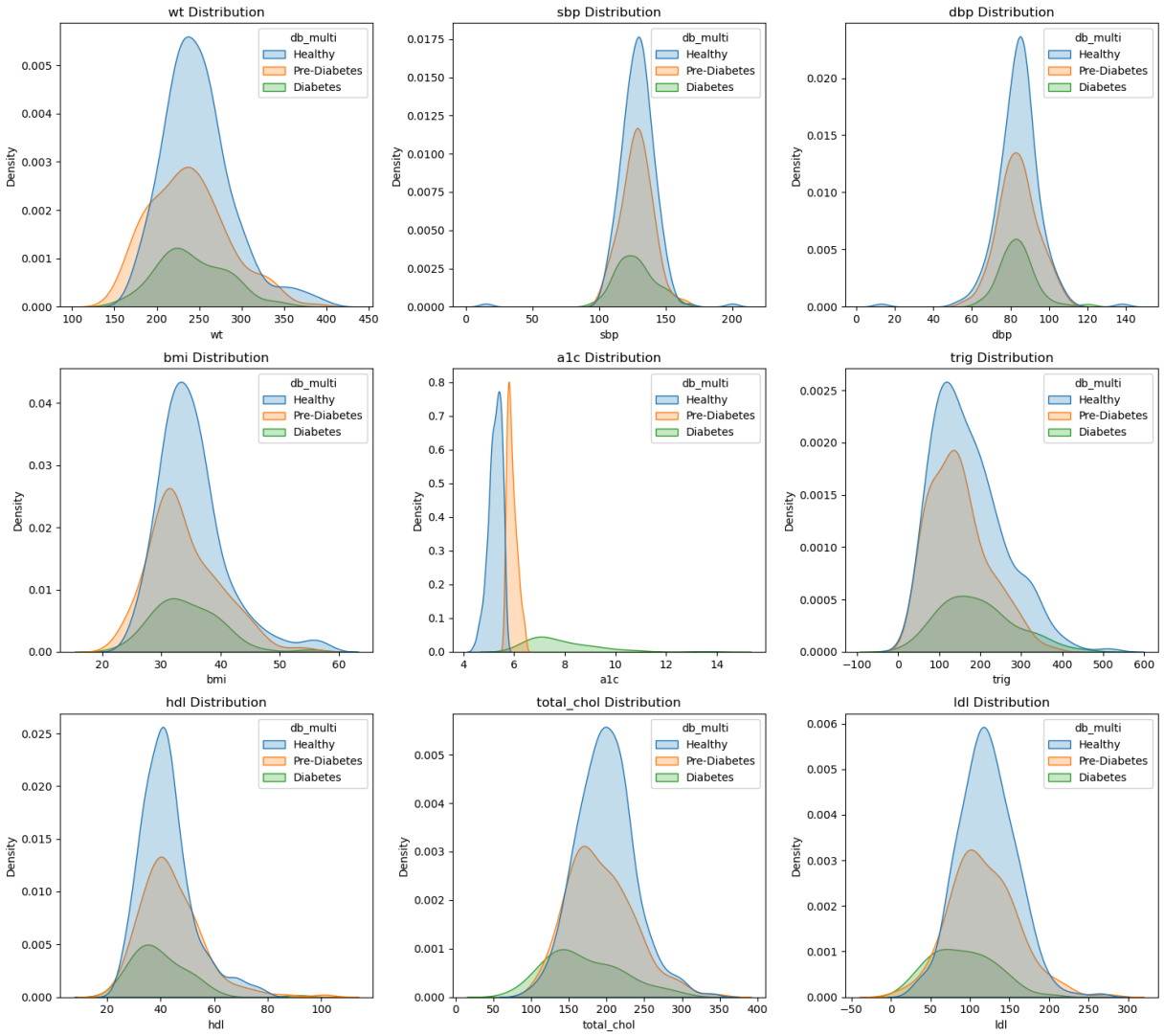
In [ ]:
```
# Inital Visit Overview

fig, axes = plt.subplots(2, 2, figsize= (10, 8))

fig.suptitle('Labs by Patient Status (Visit = Initial)', fontsize=14)
sns.boxplot(ax=axes[0, 0], data=df_init, x='db_bin', y='total_chol', whis=1)
sns.violinplot(ax=axes[0, 1], data=df_init, x='db_bin', y='a1c', whis=1)
sns.boxplot(ax=axes[1, 0], data=df_init, x='db_bin', y='trig', whis=1)
sns.boxplot(ax=axes[1, 1], data=df_init, x='db_bin', y='hdl', whis=1)


plt.tight_layout(pad=2)
plt.show()
```



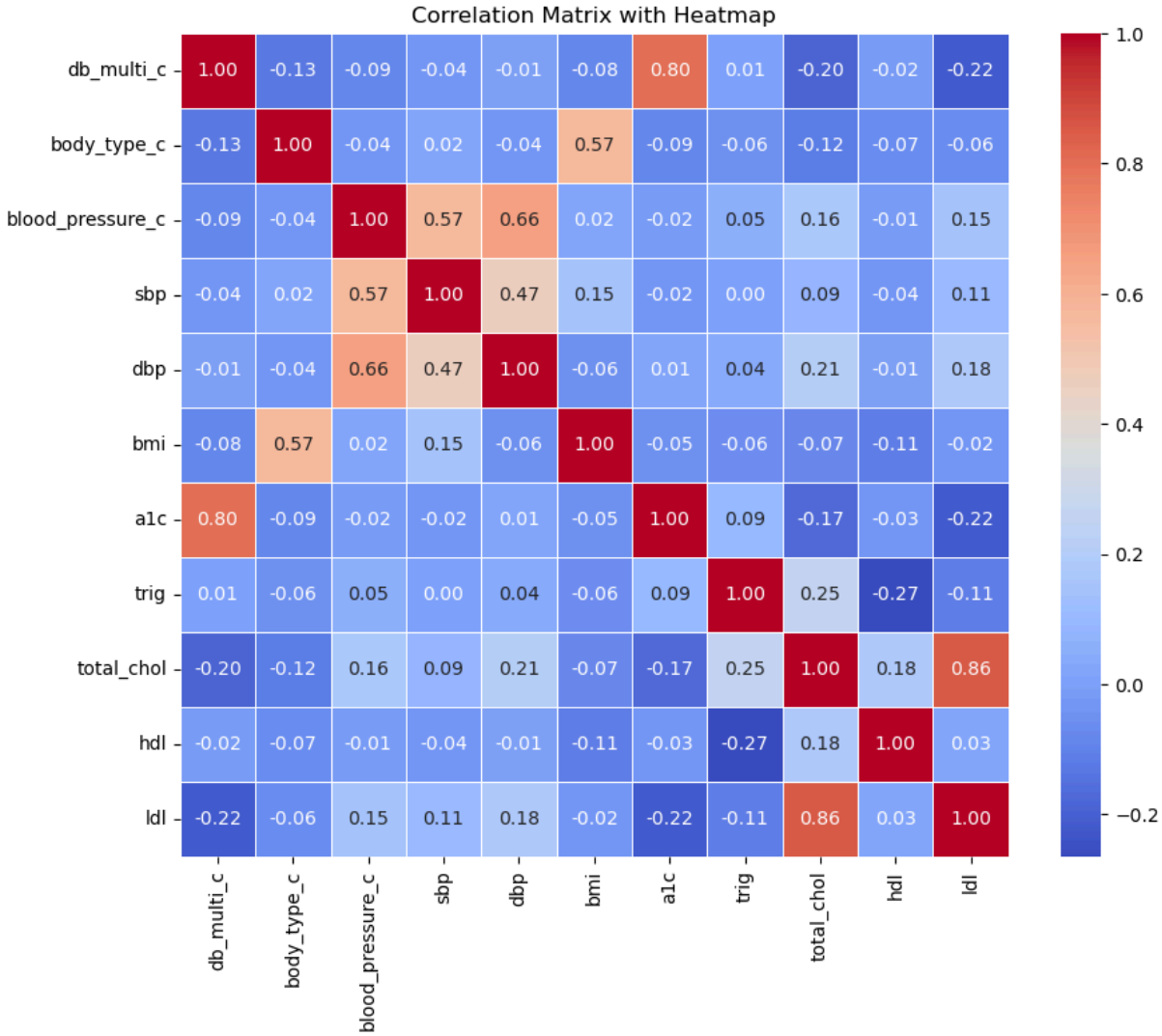
In [ ]:
```
sns.barplot(data=df_init, x='db_multi', y='a1c', hue='blood_pressure',
            estimator='median', errorbar=None, palette='Reds', hue_order=['N
plt.title('Clustered Barplot of Diabetes Status and A1C (all visits)')
plt.legend( loc='upper left')
plt.xlabel('Diabetes Class')
plt.ylabel('A1C')
plt.show()
```



In [ ]:
```
# Define numeric columns and subplot grid dimensions
numeric_cols = ['wt', 'sbp', 'dbp', 'bmi', 'a1c', 'trig', 'hdl', 'total_chol
n_cols = 3
n_rows = (len(numeric_cols) + n_cols - 1) // n_cols

# Create figure and axes for subplots
fig, axes = plt.subplots(n_rows, n_cols, figsize=(5 * n_cols, 5 * n_rows))
fig.suptitle('Distribution of Numerics by Diabetes Status', fontsize=20)

# Flatten axes array for easy iterating
axes = axes.flatten()

# Plot KDE for each numeric column with a hue
for i, col in enumerate(numeric_cols):
    sns.kdeplot(data=df_init, x=col, hue="db_multi", ax=axes[i], fill=True)
    axes[i].set_title(f'{col} Distribution')
    axes[i].set_xlabel(col)
    axes[i].set_ylabel('Density')

# Hide any unused axes if the number of numeric columns isn't a perfect mult
for ax in axes[len(numeric_cols):]:
    ax.set_visible(False)

plt.tight_layout(rect=[0, 0.03, 1, 0.95])  # Adjust layout to make room for
plt.show()
```

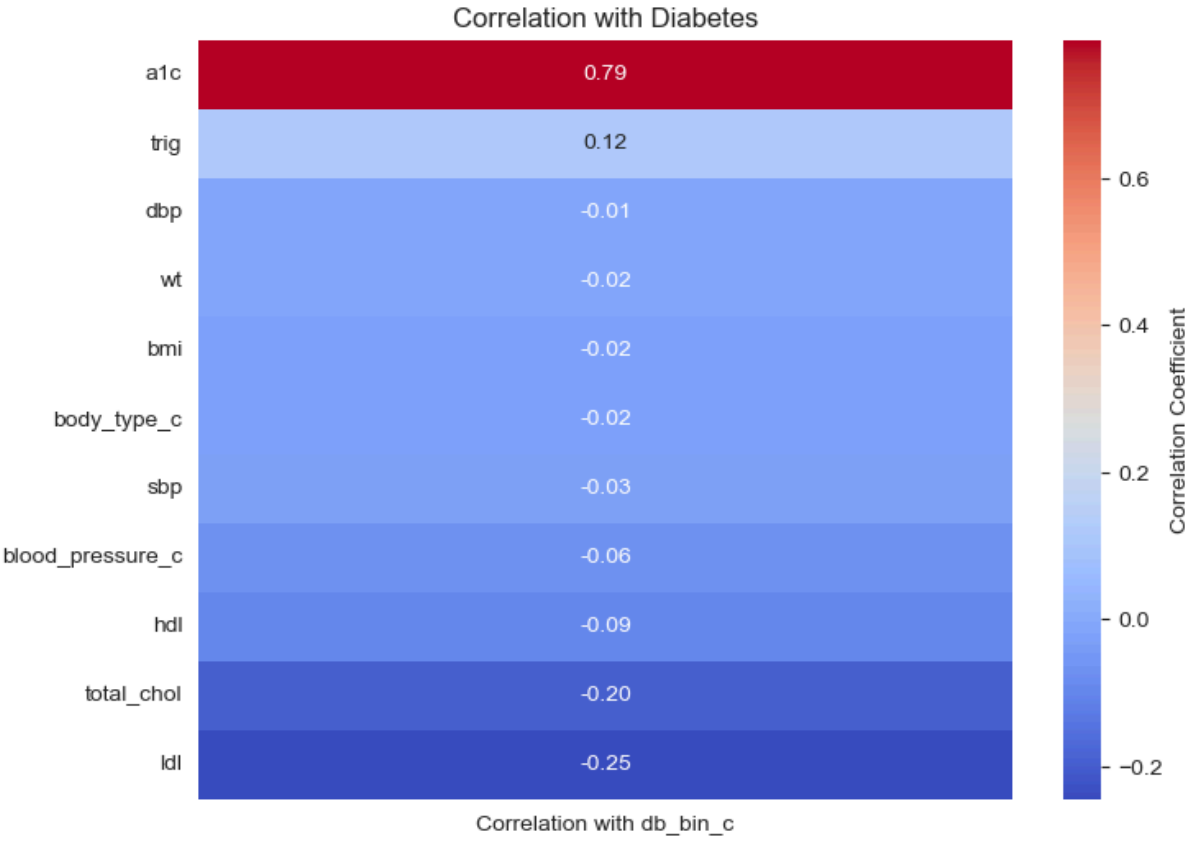


In [ ]:
```
# Select numeric columns for correlation matrix (including the newly coded c
numeric_cols = ['db_multi_c','body_type_c','blood_pressure_c', 'sbp', 'dbp',

# Calculate the correlation matrix
corr = df_init[numeric_cols].corr()


# Plot the heatmap of the correlation matrix using seaborn
plt.figure(figsize=(10, 8))
sns.heatmap(corr, annot=True, cmap='coolwarm', fmt=".2f", linewidths=.5)
plt.title('Correlation Matrix with Heatmap')
plt.show()
```



In [ ]:
```
# Select only numeric columns from df_init
df_numeric = df_init.select_dtypes(include=[np.number])

# Remove 'db_multi_c' column if it exists in df_numeric to ensure it's not i
if 'db_multi_c' in df_numeric.columns:
    df_numeric = df_numeric.drop(columns=['db_multi_c','subject'])

# Compute the correlation matrix
correlation_matrix = df_numeric.corr()

# Extract the correlation of 'db_bin_c' and drop its self-correlation
target_corr = correlation_matrix['db_bin_c'].drop('db_bin_c')

# Convert to DataFrame for heatmap compatibility and rename column for clari
target_corr_df = target_corr.to_frame(name='Correlation with db_bin_c')

# Sort correlation values in descending order for better visualization
target_corr_sorted_df = target_corr_df.sort_values(by='Correlation with db_b

# Plotting setup
sns.set_style("white")
sns.set_palette("PuBuGn_d")
plt.figure(figsize=(8, 6))  # Adjust size to accommodate all variables if ne
```

```
sns.heatmap(target_corr_sorted_df, cmap="coolwarm", annot=True, fmt='.2f', c
plt.title('Correlation with Diabetes')
plt.show()
```



Research Outcomes

Outcomes we hope to achieve after intervention over a 6 - 12 month follow up

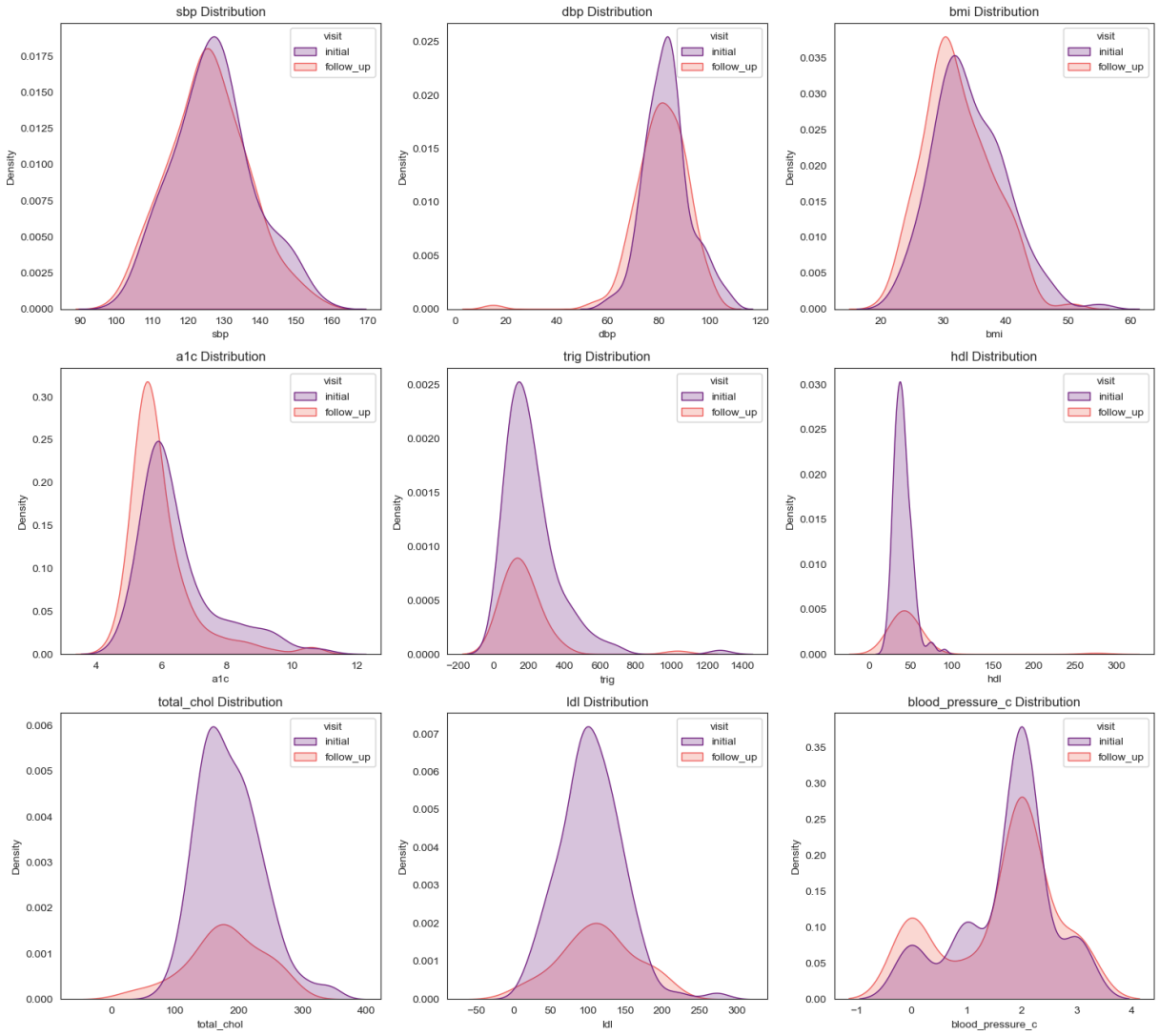
Weight loss: Evaluate percent of patients that achieved 5%; Evaluate percent of patients that achieved 10%

A1c reduction: Evaluate percent of patients that achieved HbA1c less than 6.5%; Evaluate percent of patients that reduced HbA1c by more than 1%

In [ ]:
```
# Define numeric columns and subplot grid dimensions
numeric_cols = ['sbp', 'dbp', 'bmi', 'a1c', 'trig', 'hdl', 'total_chol', 'lc
n_cols = 3
n_rows = (len(numeric_cols) + n_cols - 1) // n_cols

# Create figure and axes for subplots
fig, axes = plt.subplots(n_rows, n_cols, figsize=(5 * n_cols, 5 * n_rows))
fig.suptitle('Distribution of Labs by Between Visits', fontsize=20)

# Flatten axes array for easy iterating
axes = axes.flatten()

# Define a color palette
palette = "magma"
```

```
# Plot KDE for each numeric column with a hue
for i, col in enumerate(numeric_cols):
    sns.kdeplot(data=df_research, x=col, hue="visit", ax=axes[i], fill=True,
    axes[i].set_title(f'{col} Distribution')
    axes[i].set_xlabel(col)
    axes[i].set_ylabel('Density')

# Hide any unused axes if the number of numeric columns isn't a perfect mult
for ax in axes[len(numeric_cols):]:
    ax.set_visible(False)

plt.tight_layout(rect=[0, 0.03, 1, 0.95])  # Adjust layout to make room for
plt.show()
```

Distribution of Labs by Between Visits



In [ ]:
```
from scipy.stats import ttest_rel

# Perform the paired t-test between initial and follow-up bmi

# Filter DataFrame for initial and follow-up visits
df_initial = df_research[df_research['visit'] == 'initial'][['subject', 'bmi
```

```
df_follow_up = df_research[df_research['visit'] == 'follow_up'][['subject',

# Merge the initial and follow-up DataFrames on subject
df_paired = pd.merge(df_initial, df_follow_up, on='subject', suffixes=('_ini

# Drop any rows with NaN values in 'bmi'
df_paired.dropna(subset=['bmi_initial', 'bmi_follow_up'], inplace=True)

# Perform the paired t-test
t_stat, p_value = ttest_rel(df_paired['bmi_initial'], df_paired['bmi_follow_

# Calculate Cohen's d
mean_diff = np.mean(df_paired['bmi_initial'] - df_paired['bmi_follow_up'])
std_diff = np.std(df_paired['bmi_initial'] - df_paired['bmi_follow_up'], ddo
cohen_d = mean_diff / std_diff

t_stat, p_value, cohen_d
```

Out[ ]: (9.102393995410429, 1.4857958470373326e-15, 0.8014211042992031)

In [ ]:
```
# Filter DataFrame for initial and follow-up visits for A1C similarly to BMI
df_initial_a1c = df_research[df_research['visit'] == 'initial'][['subject',
df_follow_up_a1c = df_research[df_research['visit'] == 'follow_up'][['subjec

# Merge the initial and follow-up A1C DataFrames on subject
df_paired_a1c = pd.merge(df_initial_a1c, df_follow_up_a1c, on='subject', suf

# Drop any rows with NaN values in 'a1c'
df_paired_a1c.dropna(subset=['a1c_initial', 'a1c_follow_up'], inplace=True)

# Merge the BMI and A1C change DataFrames to analyze together
df_changes = pd.merge(df_paired[['subject', 'bmi_initial', 'bmi_follow_up']]
                      df_paired_a1c[['subject', 'a1c_initial', 'a1c_follow_u

# Calculate changes in BMI and A1C
df_changes['bmi_change'] = df_changes['bmi_follow_up'] - df_changes['bmi_ini
df_changes['a1c_change'] = df_changes['a1c_follow_up'] - df_changes['a1c_ini

# Calculate the Pearson correlation between changes in BMI and changes in A1
correlation_result = df_changes[['bmi_change', 'a1c_change']].corr().iloc[0,
correlation_result
```

Out[ ]: 0.132372417213833

In [ ]:
```
# Perform the paired t-test for A1C between initial and follow-up visits
t_stat_a1c, p_value_a1c = ttest_rel(df_paired_a1c['a1c_initial'], df_paired_

# Calculate the mean difference and standard deviation for Cohen's d calcula
mean_diff_a1c = np.mean(df_paired_a1c['a1c_initial'] - df_paired_a1c['a1c_fo
std_diff_a1c = np.std(df_paired_a1c['a1c_initial'] - df_paired_a1c['a1c_foll
cohen_d_a1c = mean_diff_a1c / std_diff_a1c

t_stat_a1c, p_value_a1c, cohen_d_a1c
```

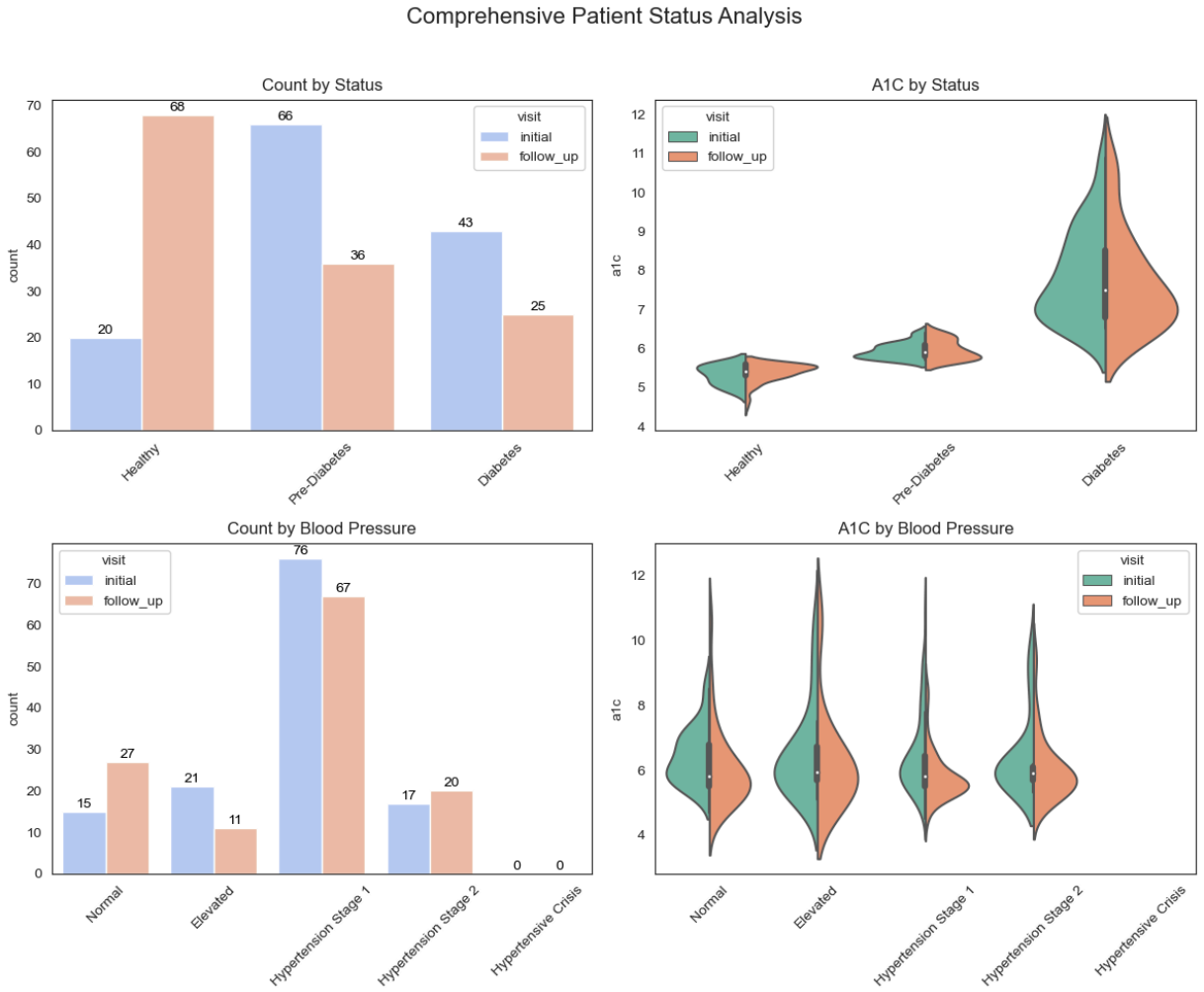
Out[ ]: (7.26401799604107, 3.6521347639572414e-11, 0.649713521157185)

In [ ]:
```
#calculating the percentage of patients achieving significant changes

# Filter and merge data for initial and follow-up visits directly
df_initial = df_research[df_research['visit'] == 'initial'][['subject', 'wt'
df_follow_up = df_research[df_research['visit'] == 'follow_up'][['subject',
df_paired = pd.merge(df_initial, df_follow_up, on='subject')

# Calculate changes in weight and A1C
df_paired['wt_change'] = df_paired['wt_follow_up'] - df_paired['wt_initial']
df_paired['a1c_change'] = pd.to_numeric(df_paired['a1c_follow_up'], errors='

# Determine significant weight loss and A1C reduction
df_paired['significant_wt_loss'] = (df_paired['wt_change'] / df_paired['wt_i
df_paired['significant_10_wt_loss'] = (df_paired['wt_change'] / df_paired['w
df_paired['significant_a1c_reduction'] = df_paired['a1c_change'] <= -0.5
df_paired['significant_1_a1c_reduction'] = df_paired['a1c_change'] <= -1.0

# Calculate the percentage of patients achieving significant changes
percent_significant_wt_loss = 100 * df_paired['significant_wt_loss'].mean()
percent_significant_10_wt_loss = 100 * df_paired['significant_10_wt_loss'].m
percent_significant_a1c_reduction = 100 * df_paired['significant_a1c_reducti
percent_significant_1_a1c_reduction = 100 * df_paired['significant_1_a1c_rec

(percent_significant_wt_loss, percent_significant_10_wt_loss, percent_signif
```

Out[ ]:
```
(51.93798449612403, 17.05426356589147, 38.759689922480625, 15.5038759689922
47)
```

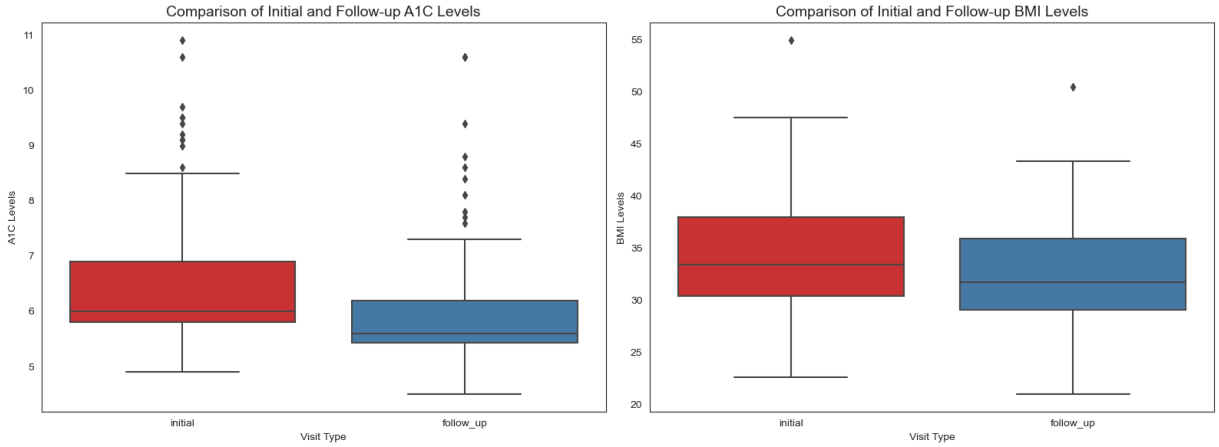
In [ ]:
```
# Setup for the figure with four subplots
fig, axes = plt.subplots(2, 2, figsize=(12, 10))
fig.suptitle('Comprehensive Patient Status Analysis', fontsize=16)

# First subplot - Count plot for 'db_multi'
countplot1 = sns.countplot(ax=axes[0, 0], data=df_research, x='db_multi', hu
axes[0, 0].set_title('Count by Status')
axes[0, 0].set_xlabel('')
axes[0, 0].tick_params(axis='x', rotation=45)
for p in countplot1.patches:
    countplot1.annotate(f'{int(p.get_height())}', (p.get_x() + p.get_width()
                        ha='center', va='center', fontsize=10, color='black',
                        textcoords='offset points')

# Second subplot - Violin plot for 'a1c' by 'db_multi'
sns.violinplot(ax=axes[0, 1], data=df_research, x='db_multi', y='a1c', hue='
axes[0, 1].set_title('A1C by Status')
axes[0, 1].set_xlabel('')
axes[0, 1].tick_params(axis='x', rotation=45)

# Third subplot - Count plot for 'blood_pressure'
countplot2 = sns.countplot(ax=axes[1, 0], data=df_research, x='blood_pressur
axes[1, 0].set_title('Count by Blood Pressure')
axes[1, 0].set_xlabel('')
axes[1, 0].tick_params(axis='x', rotation=45)
for p in countplot2.patches:
    height = int(p.get_height()) if p.get_height() > 0 else 0  # Handle case
```

```
    countplot2.annotate(f'{height}', (p.get_x() + p.get_width() / 2., height
                        ha='center', va='center', fontsize=10, color='black',
                        textcoords='offset points')

# Fourth subplot - Violin plot for 'a1c' by 'blood_pressure'
sns.violinplot(ax=axes[1, 1], data=df_research, x='blood_pressure', y='a1c',
axes[1, 1].set_title('A1C by Blood Pressure')
axes[1, 1].set_xlabel('')
axes[1, 1].tick_params(axis='x', rotation=45)

# Adjust layout
plt.tight_layout(rect=[0, 0, 1, 0.96])  # Adjust the layout to make room for
plt.show()
```

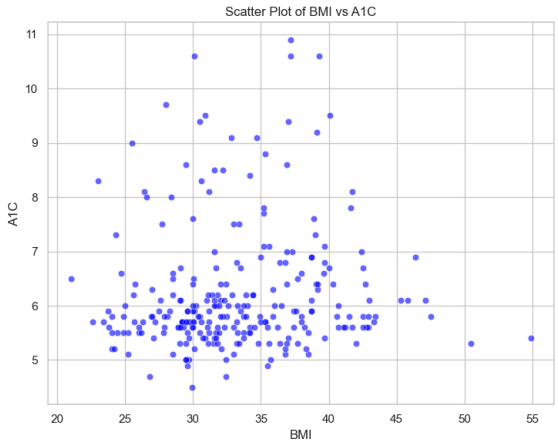


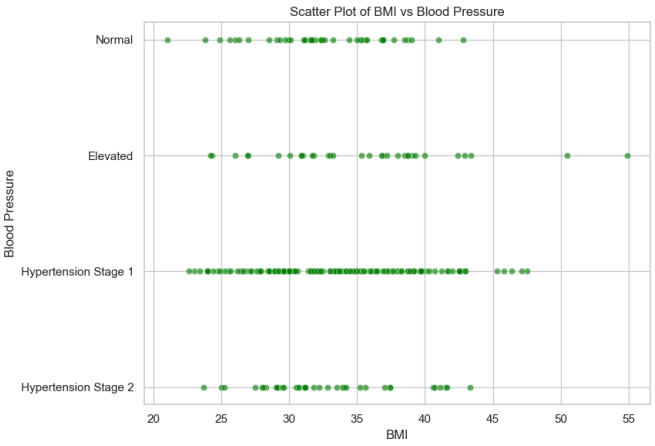
In [ ]:
```
#plotting the boxplot for A1C and BMI
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(16, 6))

# First subplot for A1C
sns.boxplot(x='visit', y='a1c', data=df_research, palette="Set1", ax=axes[0]
axes[0].set_title('Comparison of Initial and Follow-up A1C Levels', fontsize
axes[0].set_xlabel('Visit Type')
axes[0].set_ylabel('A1C Levels')


# Second subplot for BMI
```

```
sns.boxplot(x='visit', y='bmi', data=df_research, palette="Set1", ax=axes[1]
axes[1].set_title('Comparison of Initial and Follow-up BMI Levels', fontsize
axes[1].set_xlabel('Visit Type')
axes[1].set_ylabel('BMI Levels')


# Adjust layout for better spacing
plt.tight_layout()
plt.show()
```



In [ ]:
```
import seaborn as sns
import matplotlib.pyplot as plt

# Set the aesthetic style of the plots
sns.set(style="whitegrid")

# Create a figure with subplots
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(16, 6))

# Plot BMI vs A1C on the first subplot
sns.scatterplot(ax=axes[0], x='bmi', y='a1c', data=df_research, color='blue'
axes[0].set_title('Scatter Plot of BMI vs A1C')
axes[0].set_xlabel('BMI')
axes[0].set_ylabel('A1C')

# Plot BMI vs Blood Pressure on the second subplot
sns.scatterplot(ax=axes[1], x='bmi', y='blood_pressure', data=df_research, c
axes[1].set_title('Scatter Plot of BMI vs Blood Pressure')
axes[1].set_xlabel('BMI')
axes[1].set_ylabel('Blood Pressure')

# Adjust layout
plt.tight_layout()

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





Hypothesis

Empowering the veteran with knowledge and simple assessments allows clinicians to guide veterans in making scientifically based nutritional and behavioral changes, which will greatly improve their overall health and outcomes in the future.

Results BMI Reduction Analysis:

The analysis yielded a t-statistic of 9.10, with a corresponding p-value of less than 0.001. This indicates that there is a statistically significant difference in BMI between the initial and follow-up visits, with the probability of observing such a large effect by chance being less than one in a thousand.

Additionally, Cohen's d, a measure of effect size, was calculated to quantify the magnitude of the difference between the two time points. A Cohen's d value of 0.80 was observed, suggesting a large effect size according to standard interpretations (0.2 = small, 0.5 = medium, 0.8 = large). This large effect size underscores the practical significance of the BMI reduction, indicating that the decrease in BMI observed in this study is not only statistically significant but also of considerable magnitude.

A1C Level Analysis:

Similarly, the analysis for A1C levels yielded a t-statistic of 7.26, with a corresponding p-value of less than 0.001. This p-value demonstrates a statistically significant difference in A1C levels between the initial and follow-up visits.

Cohen's d was calculated to quantify the magnitude of the change in A1C levels between the visits. A Cohen's d value of 0.65 was noted, suggesting a medium to large effect size. This effect size highlights the substantial clinical relevance of the A1C reduction, marking the decrease as not only statistically significant but also clinically meaningful.

Weight Loss Achievements:

- Approximately 51.94% of patients achieved a weight loss of 5% or more from their initial weight.

- Around 17.05% of patients managed a weight loss of 10% or more.

A1C Reduction Achievements:

- Nearly 40.00% of patients achieved a significant reduction in their A1C levels, defined as a decrease of 0.5% or more.
- About 16.00% of patients reduced their A1C by more than 1%.

Classifcation Model and Evaluation

In [ ]:
```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV, cross_val_score, KFold
from sklearn.metrics import accuracy_score, classification_report, roc_curve

# List of columns to keep
columns = ['sbp', 'dbp', 'bmi', 'trig', 'hdl', 'total_chol', 'ldl','db_bin_c

# Selecting the columns from the DataFrame
df_mod = df[columns]
```

In [ ]:
```
# Define features and target
X = df_mod.drop('db_bin_c', axis=1)
y = df_mod['db_bin_c']

# Splitting the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rar

# Defining three pipelines for different classifiers
pipelines = {
    'logistic_regression': Pipeline([
        ('scaler', StandardScaler()),
        ('classifier', LogisticRegression(random_state=42))
    ]),
    'random_forest': Pipeline([
        ('scaler', StandardScaler()),
        ('classifier', RandomForestClassifier(random_state=42))
    ]),
    'svc': Pipeline([
        ('scaler', StandardScaler()),
        ('classifier', SVC(probability=True, random_state=42))
    ])
}

# Train each pipeline and evaluate
model_accuracies = {}
for name, pipeline in pipelines.items():
    pipeline.fit(X_train, y_train)
    predictions = pipeline.predict(X_test)
```

```
    model_accuracies[name] = accuracy_score(y_test, predictions)

model_accuracies
```

Out[ ]:
```
{'logistic_regression': 0.8085106382978723,
 'random_forest': 0.8085106382978723,
 'svc': 0.8191489361702128}
```

In [ ]:
```
# Parameter grid for Random Forest
param_grid = {
    'classifier__n_estimators': [100, 200, 300],
    'classifier__max_depth': [None, 10, 20, 30],
    'classifier__min_samples_split': [2, 5, 10],
    'classifier__min_samples_leaf': [1, 2, 4],
    'classifier__max_features': ['auto', 'sqrt', 'log2']
}

# Grid search with cross-validation
rf_pipeline = pipelines['random_forest']
grid_search = GridSearchCV(rf_pipeline, param_grid, cv=5, verbose=2, scoring
grid_search.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 324 candidates, totalling 1620 fits
[CV] END classifier__max_depth=None, classifier__max_features=auto, classifi
er__min_samples_leaf=1, classifier__min_samples_split=2, classifier__n_estim
ators=100; total time=   0.0s
[CV] END classifier__max_depth=None, classifier__max_features=auto, classifi
er__min_samples_leaf=1, classifier__min_samples_split=2, classifier__n_estim
ators=100; total time=   0.0s
[CV] END classifier__max_depth=None, classifier__max_features=auto, classifi
er__min_samples_leaf=1, classifier__min_samples_split=2, classifier__n_estim
ators=100; total time=   0.0s
[CV] END classifier__max_depth=None, classifier__max_features=auto, classifi
er__min_samples_leaf=1, classifier__min_samples_split=2, classifier__n_estim
ators=100; total time=   0.0s
[CV] END classifier__max_depth=None, classifier__max_features=auto, classifi
er__min_samples_leaf=1, classifier__min_samples_split=2, classifier__n_estim
ators=100; total time=   0.0s
[CV] END classifier__max_depth=None, classifier__max_features=auto, classifi
er__min_samples_leaf=1, classifier__min_samples_split=2, classifier__n_estim
ators=200; total time=   0.0s
[CV] END classifier__max_depth=None, classifier__max_features=auto, classifi
er__min_samples_leaf=1, classifier__min_samples_split=2, classifier__n_estim
ators=200; total time=   0.0s
[CV] END classifier__max_depth=None, classifier__max_features=auto, classifi
er__min_samples_leaf=1, classifier__min_samples_split=2, classifier__n_estim
ators=200; total time=   0.0s
[CV] END classifier__max_depth=None, classifier__max_features=auto, classifi
er__min_samples_leaf=1, classifier__min_samples_split=2, classifier__n_estim
ators=200; total time=   0.0s
[CV] END classifier__max_depth=None, classifier__max_features=auto, classifi
er__min_samples_leaf=1, classifier__min_samples_split=2, classifier__n_estim
ators=200; total time=   0.0s
[CV] END classifier__max_depth=None, classifier__max_features=auto, classifi
er__min_samples_leaf=1, classifier__min_samples_split=2, classifier__n_estim
ators=300; total time=   0.0s
[CV] END classifier__max_depth=None, classifier__max_features=auto, classifi
er__min_samples_leaf=1, classifier__min_samples_split=2, classifier__n_estim
ators=300; total time=   0.0s
[CV] END classifier__max_depth=None, classifier__max_features=auto, classifi
er__min_samples_leaf=1, classifier__min_samples_split=2, classifier__n_estim
ators=300; total time=   0.0s
[CV] END classifier__max_depth=None, classifier__max_features=auto, classifi
er__min_samples_leaf=1, classifier__min_samples_split=2, classifier__n_estim
ators=300; total time=   0.0s
[CV] END classifier__max_depth=None, classifier__max_features=auto, classifi
er__min_samples_leaf=1, classifier__min_samples_split=2, classifier__n_estim
ators=300; total time=   0.0s
[CV] END classifier__max_depth=None, classifier__max_features=auto, classifi
er__min_samples_leaf=1, classifier__min_samples_split=5, classifier__n_estim
ators=100; total time=   0.0s
[CV] END classifier__max_depth=None, classifier__max_features=auto, classifi
er__min_samples_leaf=1, classifier__min_samples_split=5, classifier__n_estim
ators=100; total time=   0.0s
[CV] END classifier__max_depth=None, classifier__max_features=auto, classifi
er__min_samples_leaf=1, classifier__min_samples_split=5, classifier__n_estim
ators=100; total time=   0.0s
[CV] END classifier__max_depth=None, classifier__max_features=auto, classifi
```

```
[CV] END classifier__max_depth=30, classifier__max_features=log2, classifier__min_samples_leaf=4, classifier__min_samples_split=10, classifier__n_estimators=100; total time=   0.0s
[CV] END classifier__max_depth=30, classifier__max_features=log2, classifier__min_samples_leaf=4, classifier__min_samples_split=10, classifier__n_estimators=100; total time=   0.0s
[CV] END classifier__max_depth=30, classifier__max_features=log2, classifier__min_samples_leaf=4, classifier__min_samples_split=10, classifier__n_estimators=100; total time=   0.0s
[CV] END classifier__max_depth=30, classifier__max_features=log2, classifier__min_samples_leaf=4, classifier__min_samples_split=10, classifier__n_estimators=100; total time=   0.0s
[CV] END classifier__max_depth=30, classifier__max_features=log2, classifier__min_samples_leaf=4, classifier__min_samples_split=10, classifier__n_estimators=100; total time=   0.0s
[CV] END classifier__max_depth=30, classifier__max_features=log2, classifier__min_samples_leaf=4, classifier__min_samples_split=10, classifier__n_estimators=200; total time=   0.1s
[CV] END classifier__max_depth=30, classifier__max_features=log2, classifier__min_samples_leaf=4, classifier__min_samples_split=10, classifier__n_estimators=200; total time=   0.1s
[CV] END classifier__max_depth=30, classifier__max_features=log2, classifier__min_samples_leaf=4, classifier__min_samples_split=10, classifier__n_estimators=200; total time=   0.1s
[CV] END classifier__max_depth=30, classifier__max_features=log2, classifier__min_samples_leaf=4, classifier__min_samples_split=10, classifier__n_estimators=200; total time=   0.1s
[CV] END classifier__max_depth=30, classifier__max_features=log2, classifier__min_samples_leaf=4, classifier__min_samples_split=10, classifier__n_estimators=200; total time=   0.1s
[CV] END classifier__max_depth=30, classifier__max_features=log2, classifier__min_samples_leaf=4, classifier__min_samples_split=10, classifier__n_estimators=300; total time=   0.1s
[CV] END classifier__max_depth=30, classifier__max_features=log2, classifier__min_samples_leaf=4, classifier__min_samples_split=10, classifier__n_estimators=300; total time=   0.1s
[CV] END classifier__max_depth=30, classifier__max_features=log2, classifier__min_samples_leaf=4, classifier__min_samples_split=10, classifier__n_estimators=300; total time=   0.1s
[CV] END classifier__max_depth=30, classifier__max_features=log2, classifier__min_samples_leaf=4, classifier__min_samples_split=10, classifier__n_estimators=300; total time=   0.1s
[CV] END classifier__max_depth=30, classifier__max_features=log2, classifier__min_samples_leaf=4, classifier__min_samples_split=10, classifier__n_estimators=300; total time=   0.1s
```

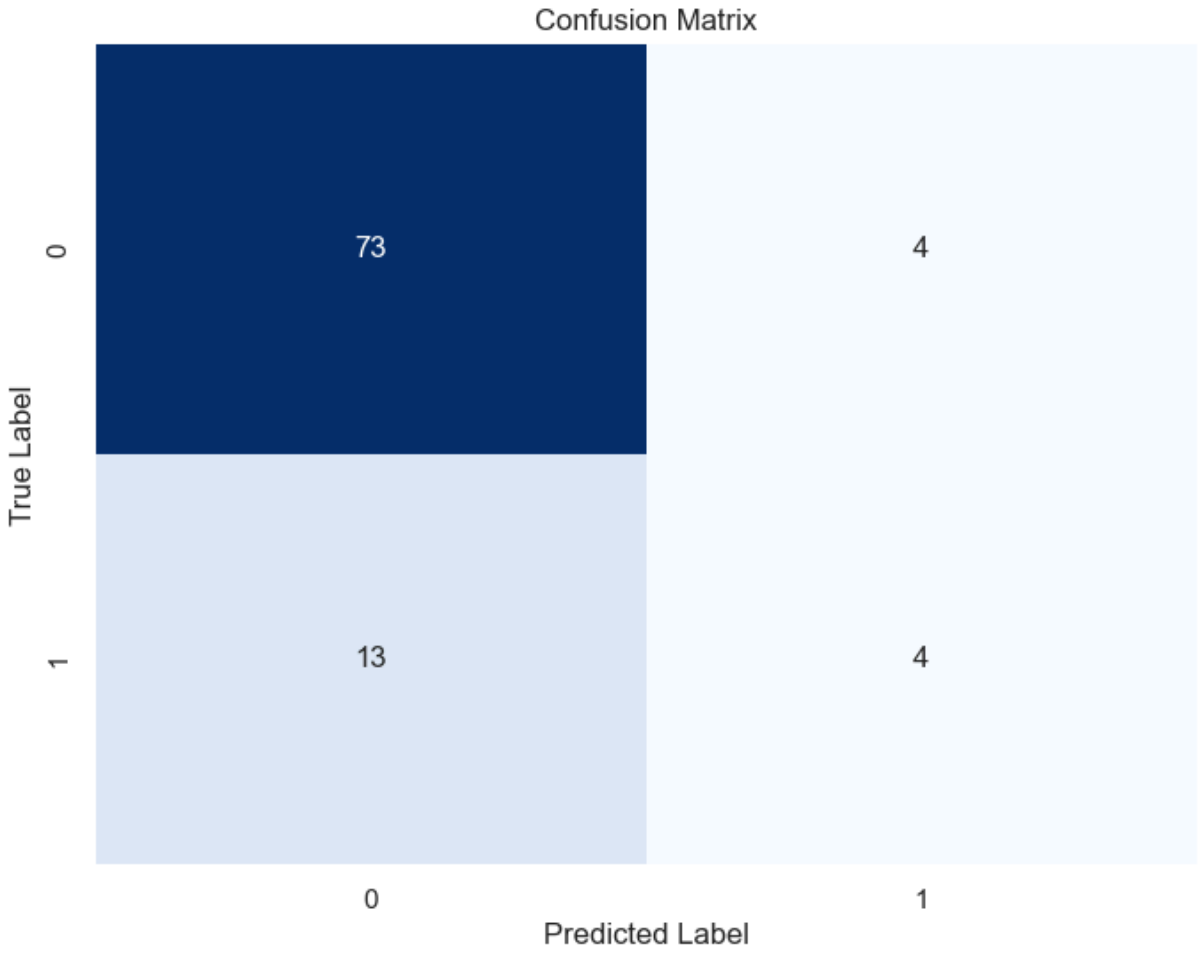
Out[ ]:

```
▸ GridSearchCV
▸ estimator: Pipeline
  ▸ StandardScaler
  ▸ RandomForestClassifier
```

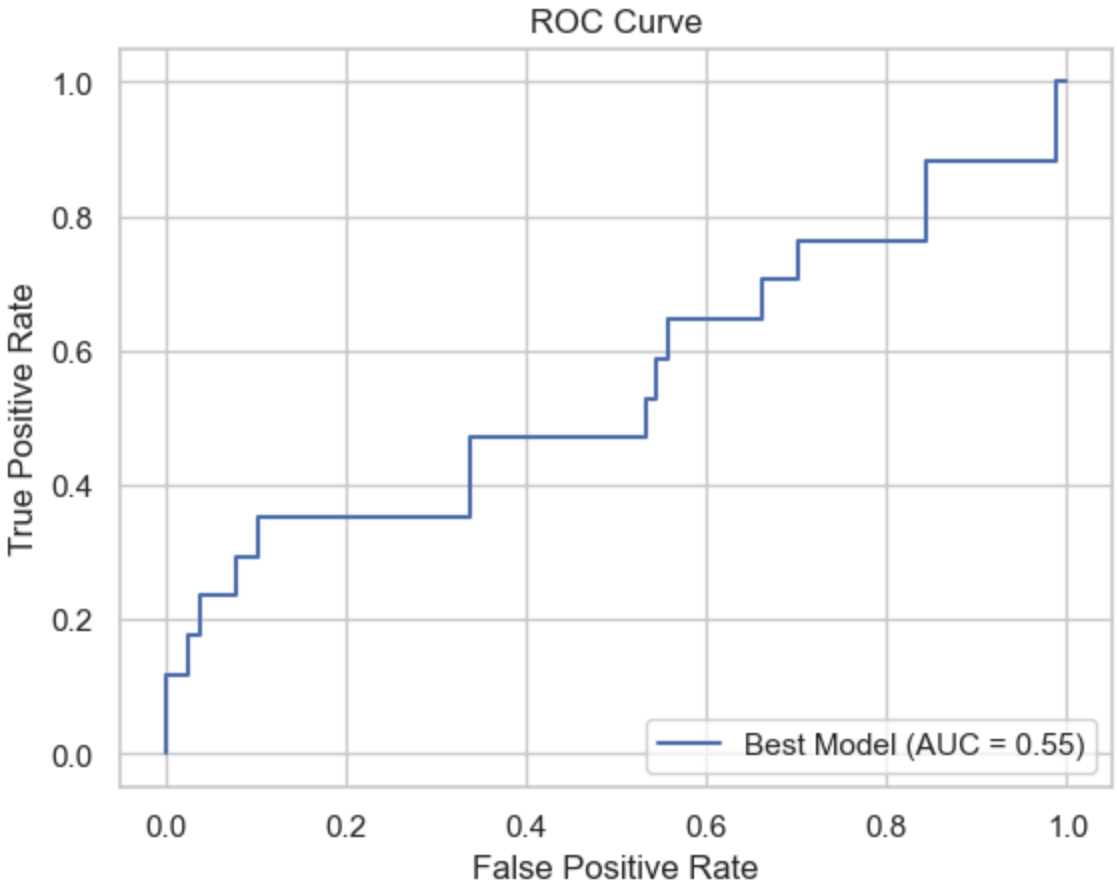
In [ ]:
```
# Evaluate it on the test set
best_model = grid_search.best_estimator_
test_predictions = best_model.predict(X_test)
test_accuracy = accuracy_score(y_test, test_predictions)
print("Test set accuracy: {:.2f}".format(test_accuracy))
```

```
Test set accuracy: 0.82
```

In [ ]:
```
# Generate the confusion matrix
cm = confusion_matrix(y_test, test_predictions)

# Plot using seaborn
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt="d", cmap='Blues', cbar=False)
plt.title('Confusion Matrix')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()

# Print classification report for precision, recall, f1-score
print("Classification Report:")
print(classification_report(y_test, test_predictions))

# ROC Curve and AUC
fpr, tpr, thresholds = roc_curve(y_test, best_model.predict_proba(X_test)[:,
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(8, 6))
roc_display = RocCurveDisplay(fpr=fpr, tpr=tpr, roc_auc=roc_auc, estimator_r
roc_display.plot()
plt.title('ROC Curve')
plt.show()

print(f"Area Under Curve (AUC): {roc_auc:.2f}")
```

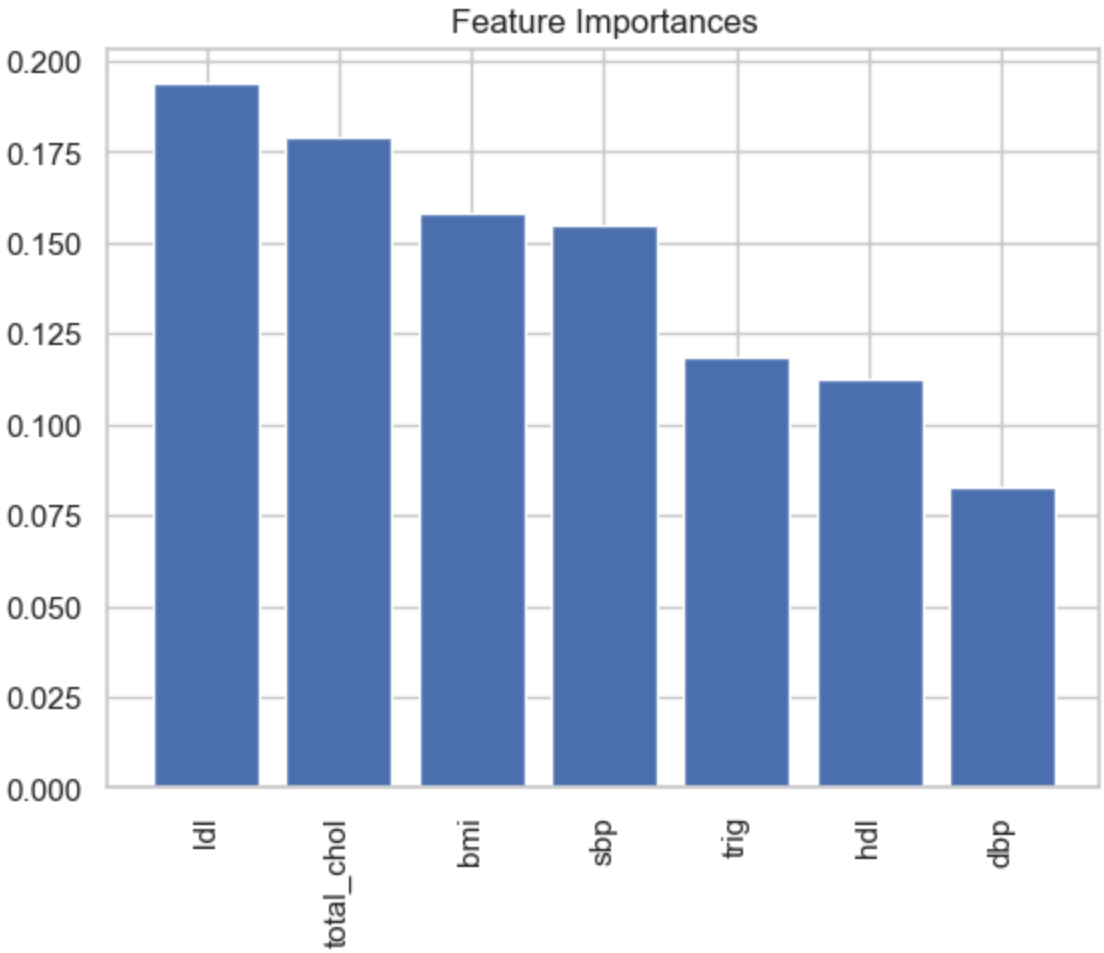



```
Classification Report:
              precision    recall  f1-score   support

           0       0.85      0.95      0.90        77
           1       0.50      0.24      0.32        17

    accuracy                           0.82        94
   macro avg       0.67      0.59      0.61        94
weighted avg       0.79      0.82      0.79        94

<Figure size 800x600 with 0 Axes>
```




```
Area Under Curve (AUC): 0.55
```

In [ ]:
```
# Get feature importances
importances = best_model.named_steps['classifier'].feature_importances_
features = X_train.columns

# Sort feature importances in descending order
indices = np.argsort(importances)[::-1]

# Rearrange feature names so they match the sorted feature importances
sorted_features = [features[i] for i in indices]

# Create a bar chart
plt.figure()
plt.title('Feature Importances')
plt.bar(range(X_train.shape[1]), importances[indices])
plt.xticks(range(X_train.shape[1]), sorted_features, rotation=90)
plt.show()
```



Overall Model Performance

Accuracy: 0.83, indicating that 83% of all predictions are correct across both classes.

Macro Average:

- Precision: 0.67, average precision across classes, which can be misleading in imbalanced datasets.
- Recall: 0.59, average recall across classes.
- F1-Score: 0.61, average F1-score across classes.

Weighted Average:

- Precision: 0.79, weighted by the support of each class, giving more importance to the majority class.
- Recall: 0.82, corresponding to overall accuracy.
- F1-Score: 0.79, weighted by support, more reflective of the true predictive performance on the dataset.

Implications:

- The model performs very well for class 0 but struggles with class 1, as evidenced by the low recall and F1-score for class 1. This suggests that the model is conservative in predicting class 1, likely due to class imbalance and weak feature association.

Regression Model and Evaluation

In [ ]:
```
# Create a DataFrame of feature importances
col = ['a1c', 'bmi', 'sbp', 'dbp', 'trig', 'hdl', 'total_chol', 'ldl']
df_linear = df[col]
```

In [ ]:
```
from sklearn.model_selection import train_test_split, cross_val_score, KFold
from sklearn.linear_model import LinearRegression, Ridge
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score


# Define cross-validation strategy
cv = KFold(n_splits=5, shuffle=True, random_state=42)

# Define features and target
X = df_linear.drop('a1c', axis=1)
y = df_linear['a1c']

# Define the models
regression_models = {
    'linear_regression': Pipeline([
        ('scaler', StandardScaler()),
        ('regressor', LinearRegression())
    ]),
    'ridge_regression': Pipeline([
        ('scaler', StandardScaler()),
        ('regressor', Ridge(random_state=42))
    ]),
    'random_forest': Pipeline([
        ('scaler', StandardScaler()),
        ('regressor', RandomForestRegressor(random_state=42))
    ]),
    'gradient_boosting': Pipeline([
        ('scaler', StandardScaler()),
        ('regressor', GradientBoostingRegressor(random_state=42))
    ])
}

# Initialize DataFrame to store results
results = pd.DataFrame(columns=['Model', 'Fold', 'MSE', 'R2'])

# Evaluate models using cross-validation
for name, pipeline in regression_models.items():
    mse_scores = cross_val_score(pipeline, X, y, cv=cv, scoring='neg_mean_sq
    r2_scores = cross_val_score(pipeline, X, y, cv=cv, scoring='r2')
    folds = list(range(1, cv.get_n_splits() + 1))
    model_results = pd.DataFrame({
        'Model': [name] * len(mse_scores),
```

```
        'Fold': folds,
        'MSE': -mse_scores,
        'R2': r2_scores
    })
    results = pd.concat([results, model_results], ignore_index=True)

print("Cross-Validation Results:")
print(results.groupby('Model').agg({'MSE': 'mean', 'R2': 'mean'}).reset_inde
```

```
Cross-Validation Results:
               Model       MSE        R2
0  gradient_boosting  1.198859 -0.378671
1  linear_regression  0.941868 -0.023537
2      random_forest  1.049244 -0.192486
3   ridge_regression  0.941736 -0.023264
```

In [ ]:
```
ridge_pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('regressor', Ridge(random_state=42))
])

param_grid = {
    'regressor__alpha': [0.1, 1, 10, 100]
}

# Assume 'cv' is defined elsewhere, e.g., cv=5
grid_search = GridSearchCV(ridge_pipeline, param_grid, cv=5, scoring='neg_me

# Fit GridSearchCV
grid_search.fit(X, y)

# Best parameters and best score from GridSearchCV
best_params = grid_search.best_params_
best_score = -grid_search.best_score_
print("Best parameters for Ridge Regression:", best_params)
print("Best MSE from GridSearchCV:", best_score)
```

```
Fitting 5 folds for each of 4 candidates, totalling 20 fits
Best parameters for Ridge Regression: {'regressor__alpha': 100}
Best MSE from GridSearchCV: 0.9398893572442463
```

In [ ]:
```
# Splitting the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rar

# Use the best estimator from the grid search to predict on the test set
tuned_ridge = grid_search.best_estimator_
y_pred_test = tuned_ridge.predict(X_test)

# Calculate MSE and R² for the test set
test_mse = mean_squared_error(y_test, y_pred_test)
test_r2 = r2_score(y_test, y_pred_test)

print("Test MSE for Tuned Ridge Regression:", test_mse)
print("Test R² for Tuned Ridge Regression:", test_r2)
```

```
Test MSE for Tuned Ridge Regression: 1.5588957234659837
Test R² for Tuned Ridge Regression: 0.049422340299792755
```

In [ ]:
```
from joblib import dump, load

# Save the model to a file
dump(tuned_ridge, 'tuned_ridge_model.joblib')
```

Out[ ]:
```
['tuned_ridge_model.joblib']
```