CS 372

Assignment 1 - KNN, Naive Bayes, Python, and You!

- Grace Biggs
- Brandon L'Abbe
- Lane Thompson
- Zhiyi Zhan

In [26]:
```
# Question 1. (Grace Biggs)
# Exploratory Data Analysis For Iris Dataset
## 1.1. Check for missing data and duplicates.
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

df = pd.read_csv('iris.csv')
print("Dataframe: ")
print(df)

print("Duplicate rows: ")
print(df.duplicated().sum()) # no duplicates

print("Missing values: ")
print(df.isnull().sum()) # 8 values missing from the sepal width column, most are Versicolor.

mode = "median"

if mode == "median":
    # Fill missing values with median
    df['sepal width (cm)'] = df.groupby('species')['sepal width (cm)'].transform(
        lambda x: x.fillna(x.median())
    )
elif mode == "mean":
```

```
    df['sepal width (cm)'] = df.groupby('species')['sepal width (cm)'].transform(
        lambda x: x.fillna(x.mean())
    )
elif mode == "drop":
    # Drop rows with missing values in sepal_width
    df = df.dropna(subset=['sepal width (cm)'])

print(df.isnull().sum())  # Should show no missing values
print(df.groupby('species')['sepal width (cm)'].describe())  # Check distributions
print(df.duplicated().sum()) # Still no duplicates
```

```
Dataframe:
     sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
0             5.993117          2.500989           4.542148          1.348742
1             5.657568          3.714626           1.800290          0.531637
2             7.751423          2.662903           7.174372          2.335111
3             5.938142          3.034625           4.448674          1.426435
4             7.054747          2.741609           4.268965          1.469104
..                 ...               ...                ...               ...
147           5.857968          3.949036           1.170024          0.305553
148           5.966645          2.334692           3.927622          1.186082
149           7.197063          2.845467           5.882156          2.105997
150          10.000000          3.500000           8.000000          2.000000
151           4.500000          2.000000           1.500000          0.200000

        species
0    versicolor
1        setosa
2     virginica
3    versicolor
4    versicolor
..          ...
147      setosa
148  versicolor
149   virginica
150   virginica
151      setosa

[152 rows x 5 columns]
Duplicate rows:
0
Missing values:
sepal length (cm)    0
sepal width (cm)     8
petal length (cm)    0
petal width (cm)     0
species              0
dtype: int64
sepal length (cm)    0
sepal width (cm)     0
petal length (cm)    0
petal width (cm)     0
species              0
```

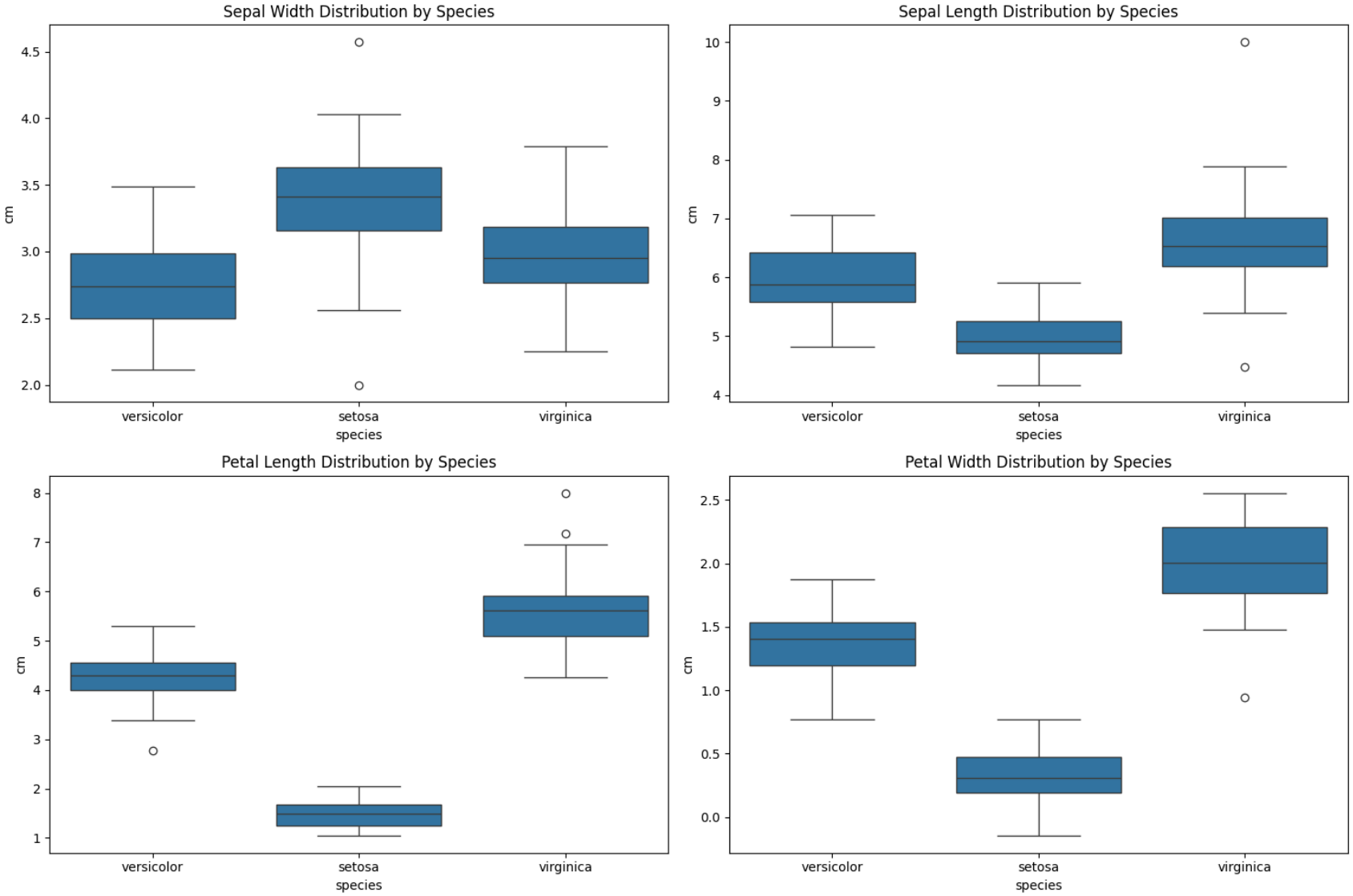
```
dtype: int64
            count      mean       std       min       25%       50%       75%  \
species
setosa       51.0  3.394057  0.426475  2.000000  3.155648  3.412096  3.634380
versicolor   50.0  2.764053  0.370867  2.113478  2.496900  2.736450  2.988227
virginica    51.0  2.973556  0.345741  2.251872  2.768942  2.952301  3.183904

                max
species
setosa      4.571993
versicolor  3.486770
virginica   3.790361
0
```

a.ii. Imputation is best used when you want to retain a dataset's sample size at the risk of some potentially inaccurate data. Conversely, Deletion is best when you can get away with reducing the sample size. Dropping the rows with missing values would mean losing 5% of the total dataset and all the useful information we could glean from the non-empty values in those rows. As such, we'll go ahead and use Median Imputation.

b. There were no duplicate rows in the provided dataset.

In [27]:
```python
import matplotlib.pyplot as plt
import seaborn as sns

# 1. Create feature-specific box plots with species breakdown
plt.figure(figsize=(15, 10))

# Sepal width - known to have outliers
plt.subplot(2, 2, 1)
sns.boxplot(x='species', y='sepal width (cm)', data=df)
plt.title('Sepal Width Distribution by Species')
plt.ylabel('cm')

# Sepal length
plt.subplot(2, 2, 2)
sns.boxplot(x='species', y='sepal length (cm)', data=df)
plt.title('Sepal Length Distribution by Species')
plt.ylabel('cm')

# Petal length
plt.subplot(2, 2, 3)
```

```
sns.boxplot(x='species', y='petal length (cm)', data=df)
plt.title('Petal Length Distribution by Species')
plt.ylabel('cm')

# Petal width
plt.subplot(2, 2, 4)
sns.boxplot(x='species', y='petal width (cm)', data=df)
plt.title('Petal Width Distribution by Species')
plt.ylabel('cm')

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

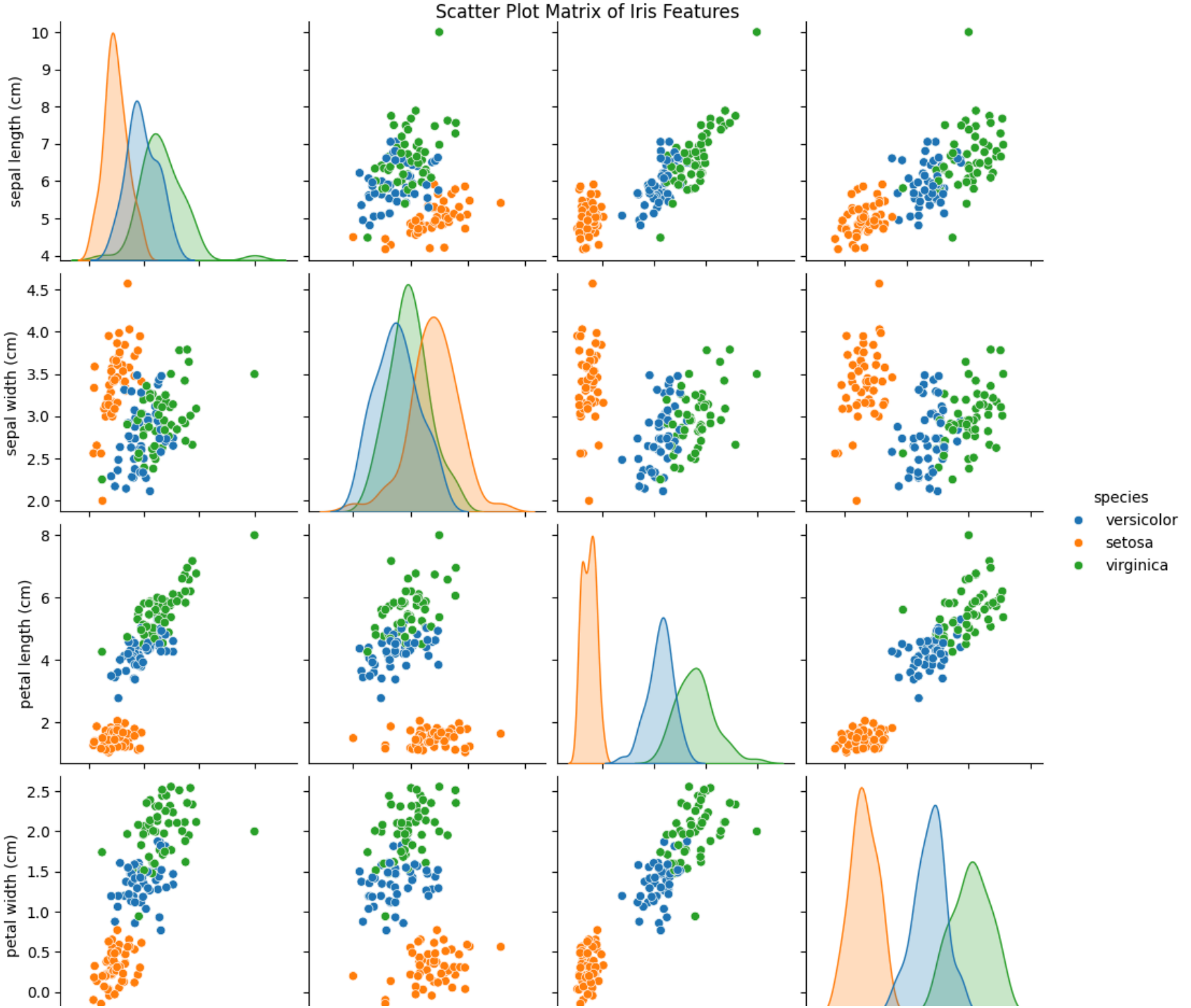


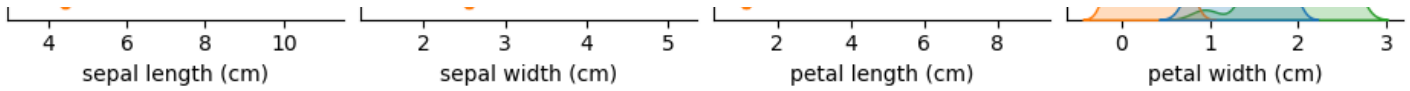
a. Box Plots indicate outliers if they lie outside 1.5x the Interquartile Range (where the middle 50% of your data is).

b. Virginica sports the most variation of the three species, with the most outliers and the widest overall distributions. Setosa is characterized by its small petals and short-but-wide sepals. Versicolor, by comparison, feels like the middle child of the three, with the most middling distributions.

c. We have eight outliers. Of them, five outliers belong to Virginica. This could be due to natural variation of Virginica, skewed feature distributions, or subjectivity in the measurements regarding where petals/sepals' lengths begin and end. Virginica also has a much wider distribution for petal length and width than the other species.

In [28]:
```
## 1.3. Draw a Scatter Plot Matrix for Iris flowers' descriptive features.
sns.pairplot(df, hue='species', diag_kind='kde')
plt.suptitle('Scatter Plot Matrix of Iris Features', y=1.0)
plt.show()
```

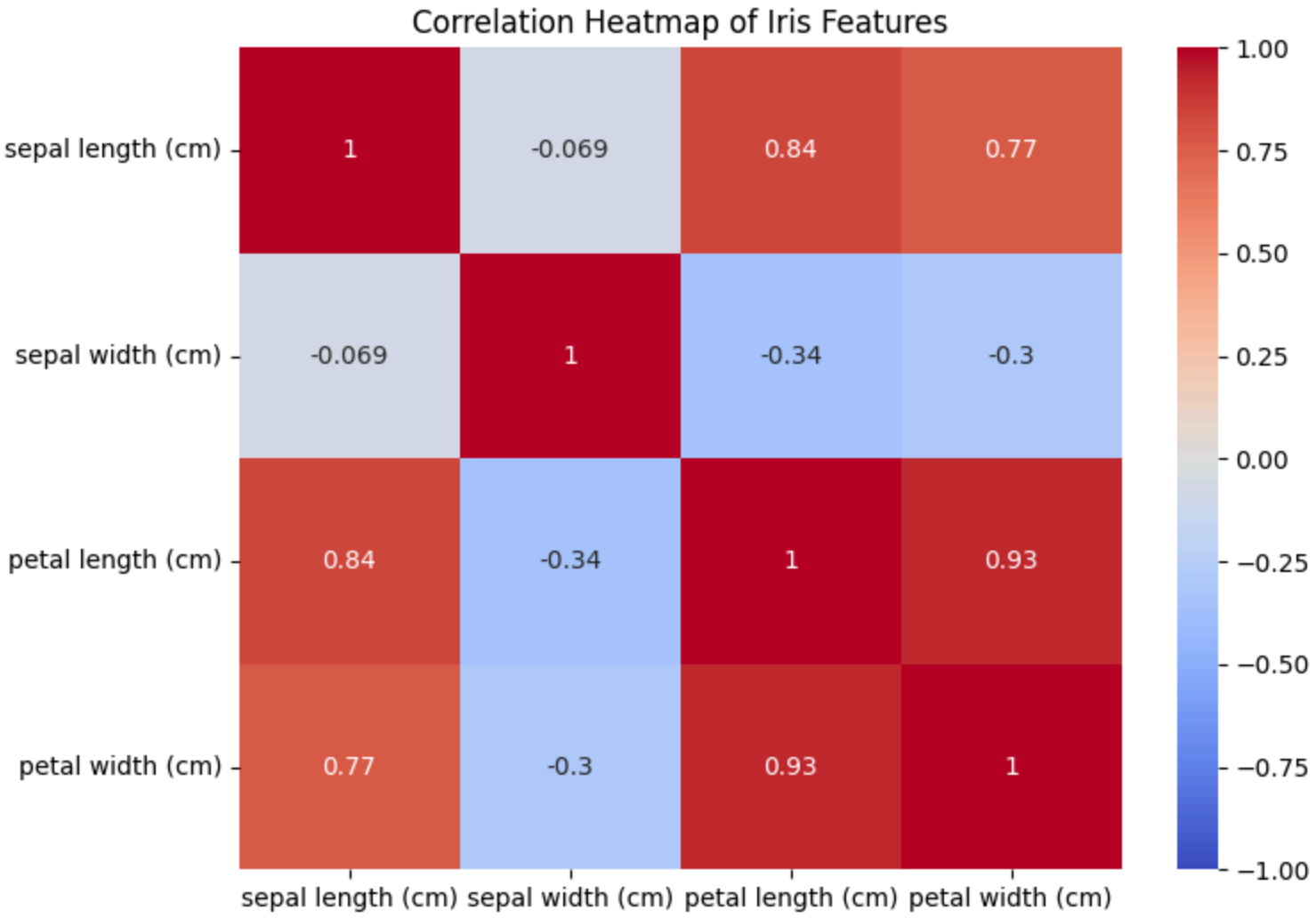

Scatter Plot Matrix of Iris Features
sepal length (cm)
sepal width (cm)
petal length (cm)
petal width (cm)
species
versicolor
setosa
virginica



a. At a glance, the petal lengths and widths are the most separate from each other between species - note the plots on the diagonals. Setosa in particular shares almost none of the same lengths as the other two species.

b. Scatterplot Matrices visualize the relationships between features and let you see if they're correlated. For example, this one lets us see that petal widths and sepal lengths seem to be linearly correlated, while sepal lengths and sepal widths seem to be independent of each other (really, anything to do with sepal widths seems independent of everything else).

In [29]:
```
## 1.4. Plot Heatmap correlation for descriptive features.
plt.figure(figsize=(8,6))
corr = df.drop(columns='species').corr()
sns.heatmap(corr, annot=True, cmap='coolwarm', vmin=-1, vmax=1)
plt.title('Correlation Heatmap of Iris Features')
plt.show()
```



b.ii. The most striking takeaway from this heatmap is that sepal width seems to be aggressively unrelated from the other three features, while everything else is pretty clearly correlated.

In [30]:
```
# Split the data 80/20.
## 1. Preserve the proportion of each class when splitting.
## 2. Shuffle data before split. Also make sure this code block reruns and randomizes the split correctly.
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

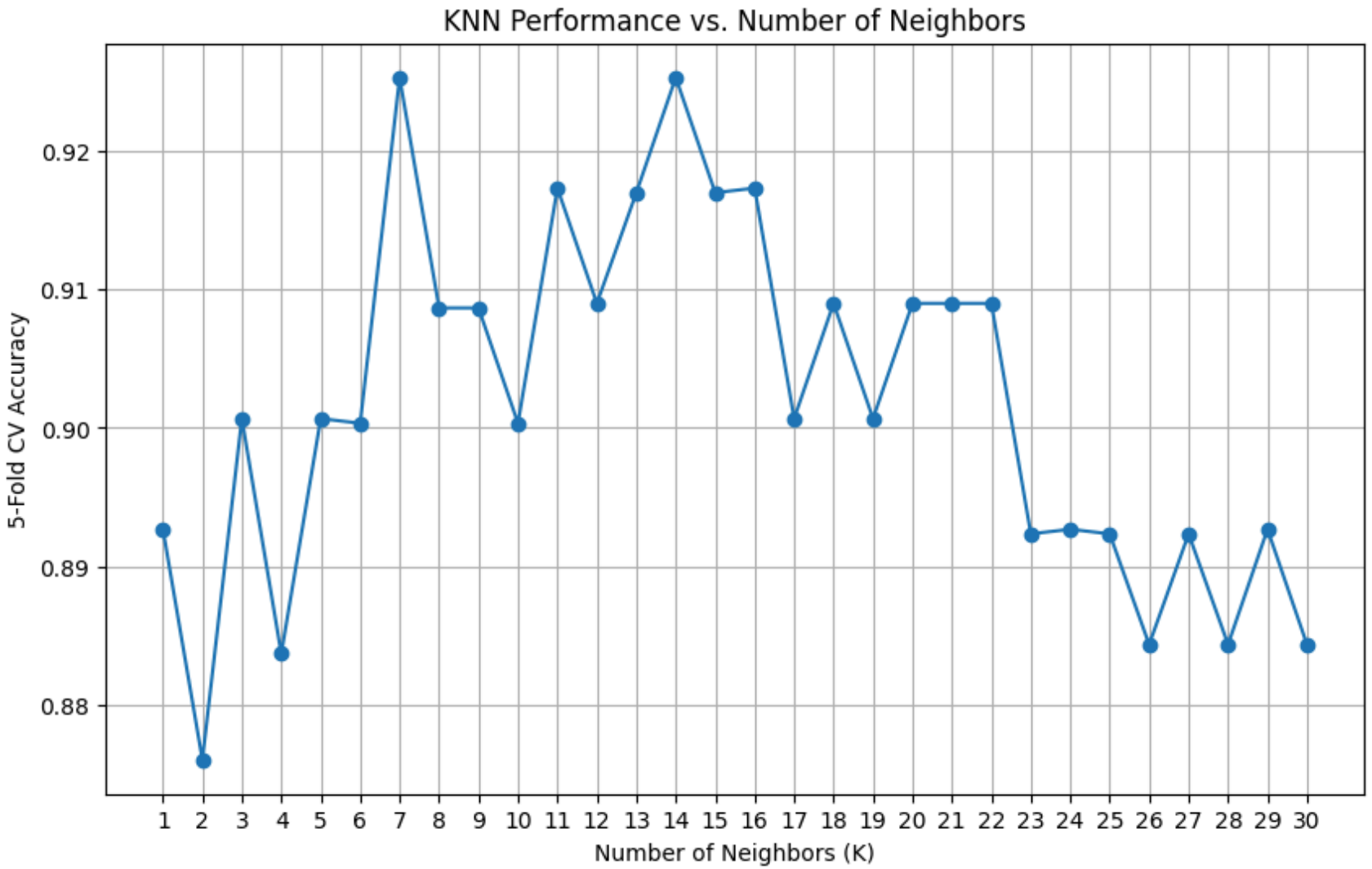
```
from sklearn.model_selection import train_test_split # https://scikit-learn.org/stable/modules/generated/sklearn.mode

X = df.drop(columns='species')
y = df['species']
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    stratify=y,
    shuffle=True,
    random_state=42
)
```

In [31]:

```
# KNN
## 3. Use StandardScaler to put all descriptive features on the same scale.
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.preprocessing import StandardScaler # https://scikit-learn.org/stable/modules/generated/sklearn.preproce
from sklearn.neighbors import KNeighborsClassifier # https://scikit-learn.org/stable/modules/generated/sklearn.neighb
from sklearn.model_selection import cross_val_score # https://scikit-learn.org/stable/modules/cross_validation.html

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

## 4. Implement KNN using SKLearn's KNeighborsClassifier.
## 5. Experiment with different distance metrics (Euclidean and Manhattan). Compare the two and comment on your findi
metrics = ['euclidean', 'manhattan']
results = {}

for metric in metrics:
    knn = KNeighborsClassifier(n_neighbors=5, metric=metric)
    knn.fit(X_train_scaled, y_train)
    train_acc = knn.score(X_train_scaled, y_train)
    test_acc = knn.score(X_test_scaled, y_test)
    results[metric] = {'train': train_acc, 'test': test_acc}

print("Metric Comparison:")
for metric, scores in results.items():
    print(f"{metric.capitalize()} - Train: {scores['train']:.4f}, Test: {scores['test']:.4f}")
# Euclidean performs 3% better than Manhattan on the Test dataset, and only marginally worse than Manhattan on the tr
# Both are comparable to each other but it's safe to say that Euclidean provides better accuracy.
```

```python
## 6. Choose the best number of neighbors using 5-fold cross validation.
# https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html
k_values = np.arange(1, 31) # Realistically overkill but this makes for a nice graph
cv_scores = []

for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k, metric='euclidean')
    scores = cross_val_score(knn, X_train_scaled, y_train, cv=5, scoring='accuracy')
    cv_scores.append(scores.mean())

## 7. Draw a 2D plot to show the average accuracy of KNN classifier vs. different values of K. Analyze the result and
plt.figure(figsize=(10, 6))
plt.plot(k_values, cv_scores, marker='o', linestyle='-')
plt.title('KNN Performance vs. Number of Neighbors')
plt.xlabel('Number of Neighbors (K)')
plt.ylabel('5-Fold CV Accuracy')
plt.xticks(k_values)
plt.grid(True)
plt.show()

best_k = k_values[np.argmax(cv_scores)]
print(f"Best K: {best_k} with accuracy: {max(cv_scores):.4f}")

## 8. Evaluate your model on the test data using Accuracy based on the best K found above.
best_knn = KNeighborsClassifier(n_neighbors=best_k, metric='euclidean')
best_knn.fit(X_train_scaled, y_train)
test_accuracy = best_knn.score(X_test_scaled, y_test)
print(f"\nTest Accuracy (k={best_k}): {test_accuracy:.4f}")

## Show misclassified flowers in a table, with the true label in one column and the predicted table in another columr
y_pred = best_knn.predict(X_test_scaled)
misclassified = X_test.copy()
misclassified['true_species'] = y_test
misclassified['predicted_species'] = y_pred
misclassified = misclassified[y_test != y_pred]

print("\nMisclassified Flowers:")
print(misclassified[['true_species', 'predicted_species']])
# Three flowers misclassified
```

```
Metric Comparison:
Euclidean - Train: 0.9256, Test: 0.9032
Manhattan - Train: 0.9339, Test: 0.8710
```



```
Best K: 7 with accuracy: 0.9253

Test Accuracy (k=7): 0.9032

Misclassified Flowers:
    true_species predicted_species
110   versicolor         virginica
111    virginica        versicolor
84    versicolor         virginica
```

5. Euclidean performs 3% better than Manhattan on the Test dataset, and only marginally worse than Manhattan on the training set. Both are comparable to each other but it's safe to say that Euclidean provides better accuracy.

6. (This comment intentionally left blank.)

7. K=7 is the obvious best choice, but K=14 having the same accuracy is notable. It seems that the range of 7 through 16 contains the most reasonable choices, after which we see a noticeable drop-off in accuracy.

8. Seems our model has some trouble with versicolors and virginicas. This might be related to Virginicas having the most outliers out of the species. Overall though, the model has pretty good accuracy and is definitely a step up from random guessing.

In [32]:
```python
# Naive Bayes
## 1. Use the built-in SKLearn library to predict the species of Iris in the test dataset.
from sklearn.naive_bayes import GaussianNB # https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.Ga
from sklearn.metrics import accuracy_score # https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accura

nb_classifier = GaussianNB()
nb_classifier.fit(X_train_scaled, y_train)
y_pred_nb = nb_classifier.predict(X_test_scaled)

## 2. Use Accuracy to assess the performance of your classifier.
nb_accuracy = accuracy_score(y_test, y_pred_nb)
print(f"\nNaive Bayes Test Accuracy: {nb_accuracy:.4f}")

## 3. Show misclassified flowers in a table, with the true label in one column and the predicted table in another col
misclassified_nb = X_test.copy()
misclassified_nb['true_species'] = y_test
misclassified_nb['predicted_species'] = y_pred_nb
misclassified_nb = misclassified_nb[y_test != y_pred_nb]
```

```
print("\nMisclassified Flowers (Naive Bayes):")
print(misclassified_nb[['true_species', 'predicted_species']])
# Accuracy and misclassifications are exactly the same as KNN...
```

```
Naive Bayes Test Accuracy: 0.9032

Misclassified Flowers (Naive Bayes):
    true_species predicted_species
110   versicolor         virginica
111    virginica        versicolor
84    versicolor         virginica
```

In [33]:

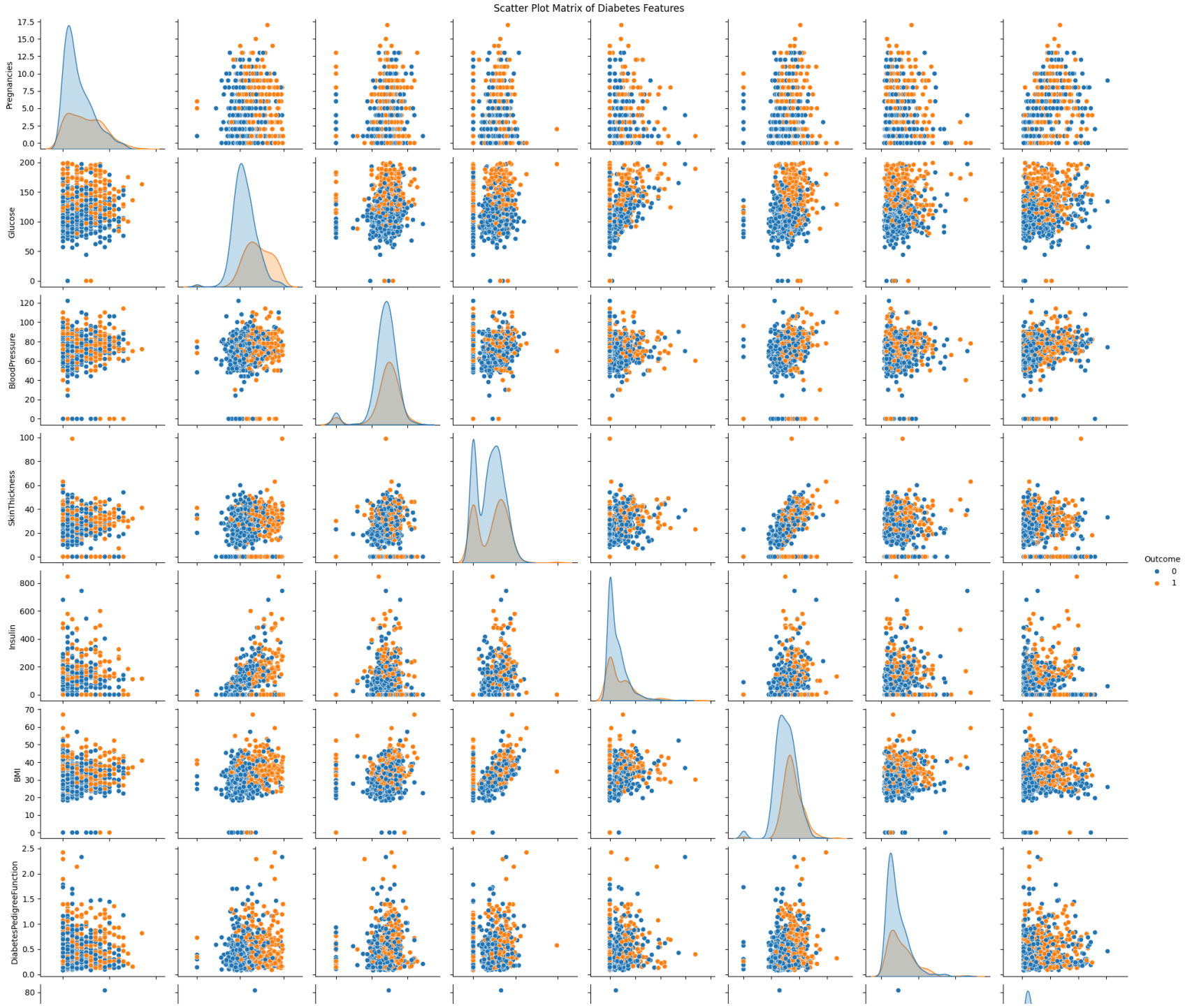
```
# Question 2. Predicting the onset of diabetes based on diagnostic measures.
# Exploratory Data Analysis of Pima Indians Diabetes Database.
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

## 1. Check for missing data and duplicates.
df = pd.read_csv('diabetes.csv')
print(df)

print("Duplicate rows: ")
print(df.duplicated().sum()) # no duplicates

# For this dataset, there are no empty values, but there are some columns with an invalid value of 0
# We will deal with this in Step 5 where we are suppose to identify unrealistic values
print("Missing values: ")
print(df.isnull().sum())

## 2. Draw a Scatter Plot Matrix
sns.pairplot(df, hue='Outcome', diag_kind='kde')
plt.suptitle('Scatter Plot Matrix of Diabetes Features', y=1.0)
plt.show()

## 3. Draw a Box Plot to spot outliers.
plt.figure(figsize=(10, 6))
sns.boxplot(data=df.drop(columns='Outcome'))
plt.title('Box Plot of Diabetes Features')
plt.show()

# TODO: Document decisions on how to handle outliers based on your analysis and reasoning.
```

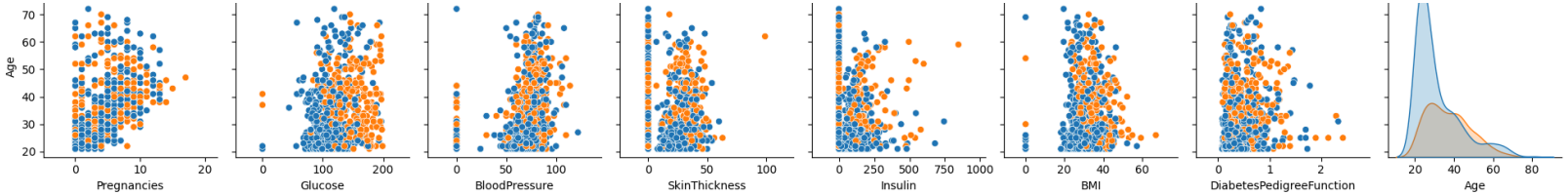
```
## 4. Plot Heatmap correlation for descriptive features.
plt.figure(figsize=(8,6))
corr = df.drop(columns='Outcome').corr()
sns.heatmap(corr, annot=True, cmap='coolwarm', vmin=-1, vmax=1)
plt.title('Correlation Heatmap of Diabetes Features')
plt.show()

## 5. Identify features containing unrealistic zero values
# We want to check Glucose, Blood Pressure, and BMI for zeroes, all other columns can have 0 values
# To deal with this, we should treat them the same as missing values, replacing the 0s with the median value

sns.histplot(df, y='Glucose')
plt.title('Before Imputation Glucose Histogram')
plt.show()

sns.histplot(df, y='BloodPressure')
plt.title('Before Imputation Blood Pressure Histogram')
plt.show()

sns.histplot(df, y='BMI')
plt.title('Before Imputation BMI Histogram')
plt.show()

sns.kdeplot(df, y='Glucose')
plt.title('Before Imputation Glucose Density Plot')
plt.show()

sns.kdeplot(df, y='BloodPressure')
plt.title('Before Imputation Blood Pressure Density Plot')
plt.show()

sns.kdeplot(df, y='BMI')
plt.title('Before Imputation BMI Density Plot')
plt.show()

print("Invalid values: ")
missing_value_rows = df.query('Glucose == 0 or BloodPressure == 0 or BMI == 0')
print(missing_value_rows.count()['BMI']) # Arbitraility polling the count for the BMI column (they are all the same c

#Replace invalid values with median
df['Glucose'] = df.groupby('Outcome')['Glucose'].transform(
```

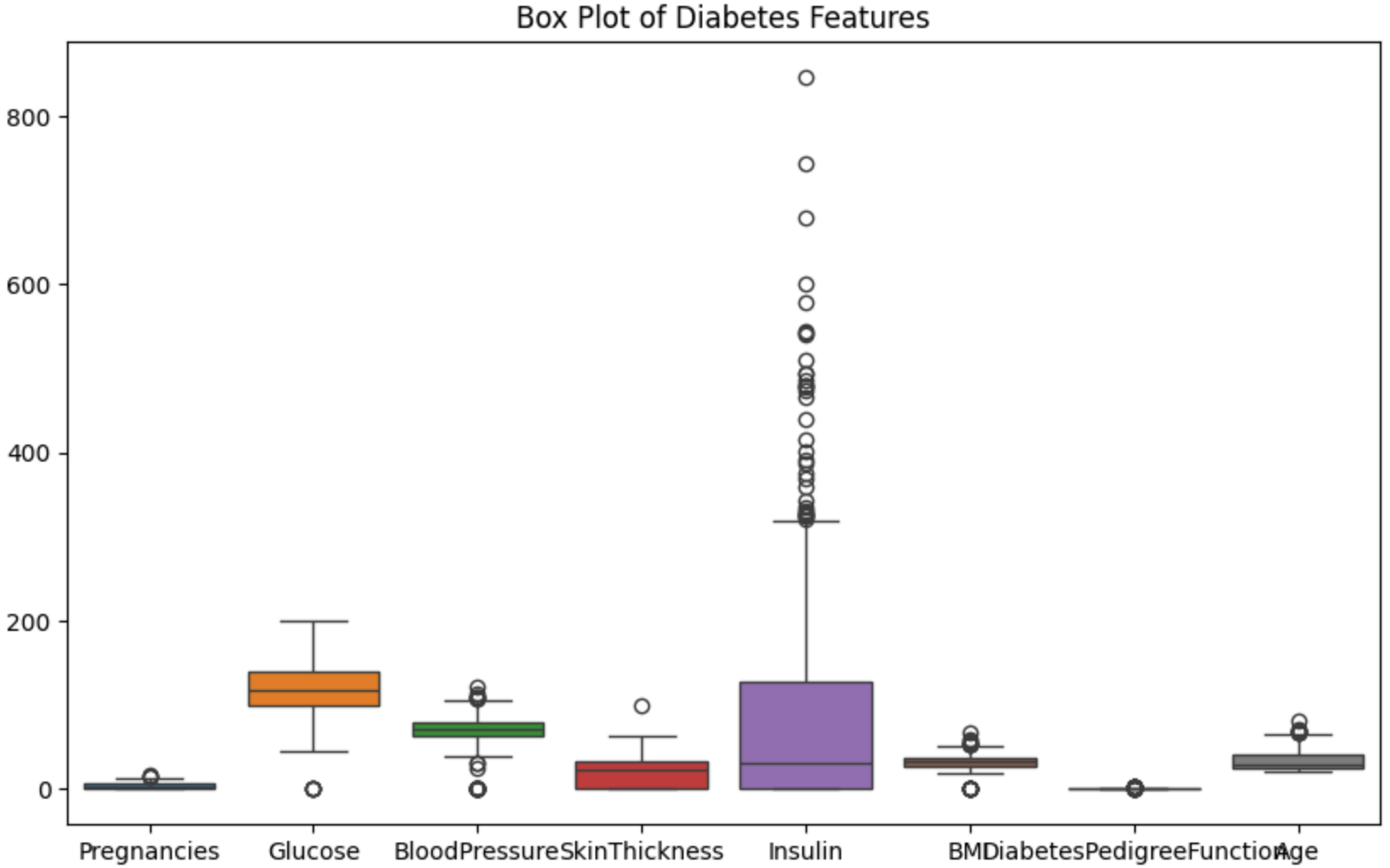
```
    lambda x: x.replace(0, x.median())
)

df['BloodPressure'] = df.groupby('Outcome')['BloodPressure'].transform(
    lambda x: x.replace(0, x.median())
)

df['BMI'] = df.groupby('Outcome')['BMI'].transform(
    lambda x: x.replace(0, x.median())
)

#Post Imputation Plots... These show the 0 groups disappearing
sns.histplot(df, y='Glucose')
plt.title('After Imputation Glucose Histogram')
plt.show()

sns.histplot(df, y='BloodPressure')
plt.title('After Imputation Blood Pressure Histogram')
plt.show()

sns.histplot(df, y='BMI')
plt.title('After Imputation BMI Histogram')
plt.show()

sns.kdeplot(df, y='Glucose')
plt.title('After Imputation Glucose Density Plot')
plt.show()

sns.kdeplot(df, y='BloodPressure')
plt.title('After Imputation Blood Pressure Density Plot')
plt.show()

sns.kdeplot(df, y='BMI')
plt.title('After Imputation BMI Density Plot')
plt.show()


# Split the data 80/20.
## 1. Preserve the proportion of each class when splitting.
## 2. Shuffle data before split. Also make sure this code block reruns and randomizes the split correctly.
import numpy as np
from sklearn.model_selection import train_test_split # https://scikit-learn.org/stable/modules/generated/sklearn.mode
```

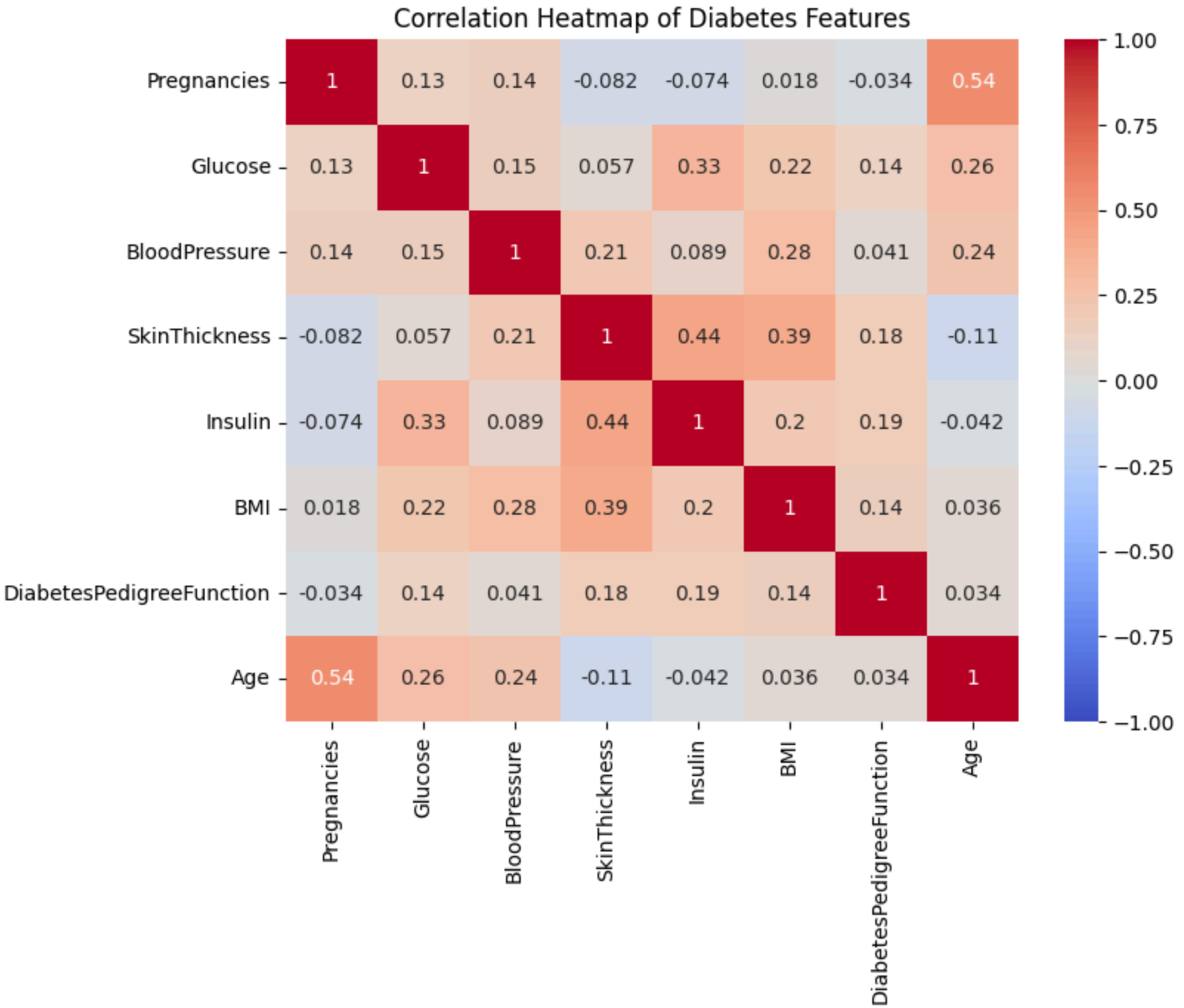
```
X = df.drop(columns='Outcome')
y = df['Outcome']
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    stratify=y,
    shuffle=True,
    random_state=42
)
```

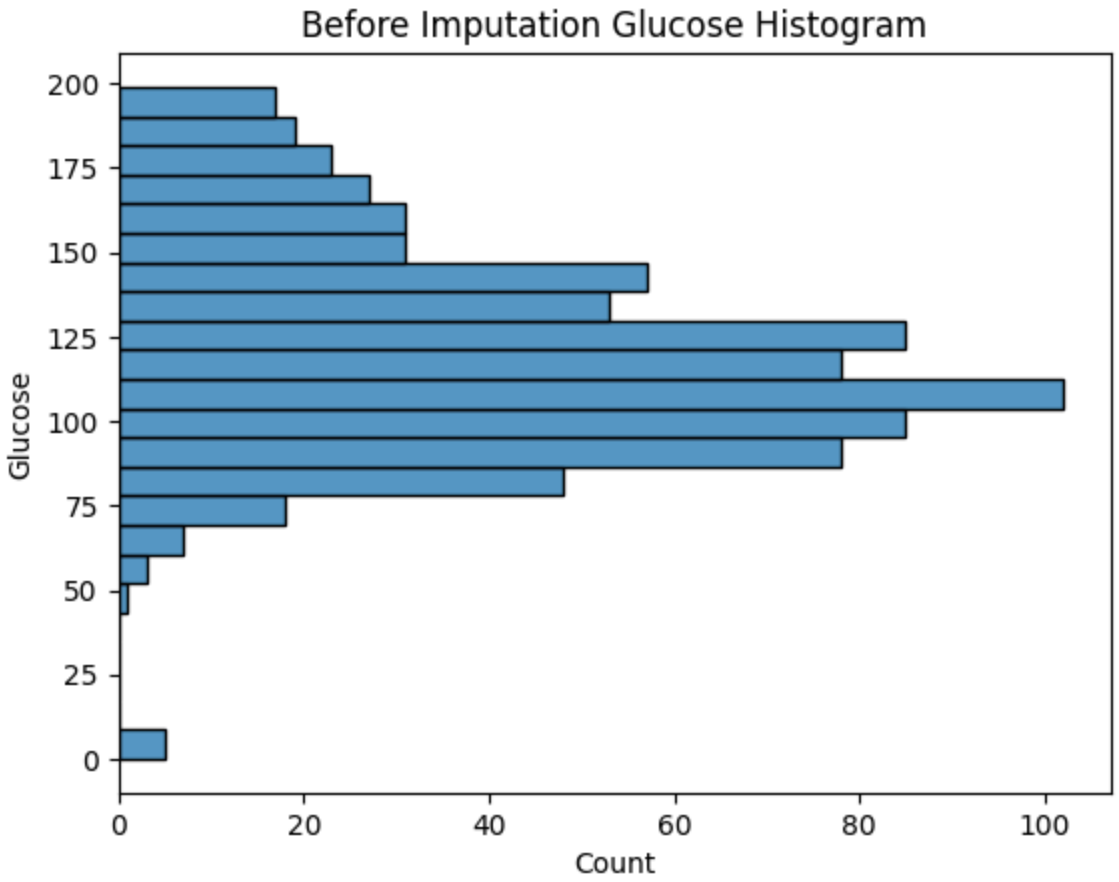
```
     Pregnancies  Glucose  BloodPressure  SkinThickness  Insulin   BMI  \
0              6      148             72             35        0  33.6
1              1       85             66             29        0  26.6
2              8      183             64              0        0  23.3
3              1       89             66             23       94  28.1
4              0      137             40             35      168  43.1
..           ...      ...            ...            ...      ...   ...
763           10      101             76             48      180  32.9
764            2      122             70             27        0  36.8
765            5      121             72             23      112  26.2
766            1      126             60              0        0  30.1
767            1       93             70             31        0  30.4

     DiabetesPedigreeFunction  Age  Outcome
0                       0.627   50        1
1                       0.351   31        0
2                       0.672   32        1
3                       0.167   21        0
4                       2.288   33        1
..                        ...  ...      ...
763                     0.171   63        0
764                     0.340   27        0
765                     0.245   30        0
766                     0.349   47        1
767                     0.315   23        0

[768 rows x 9 columns]
Duplicate rows:
0
Missing values:
Pregnancies                 0
Glucose                     0
BloodPressure               0
SkinThickness               0
Insulin                     0
BMI                         0
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
dtype: int64
```

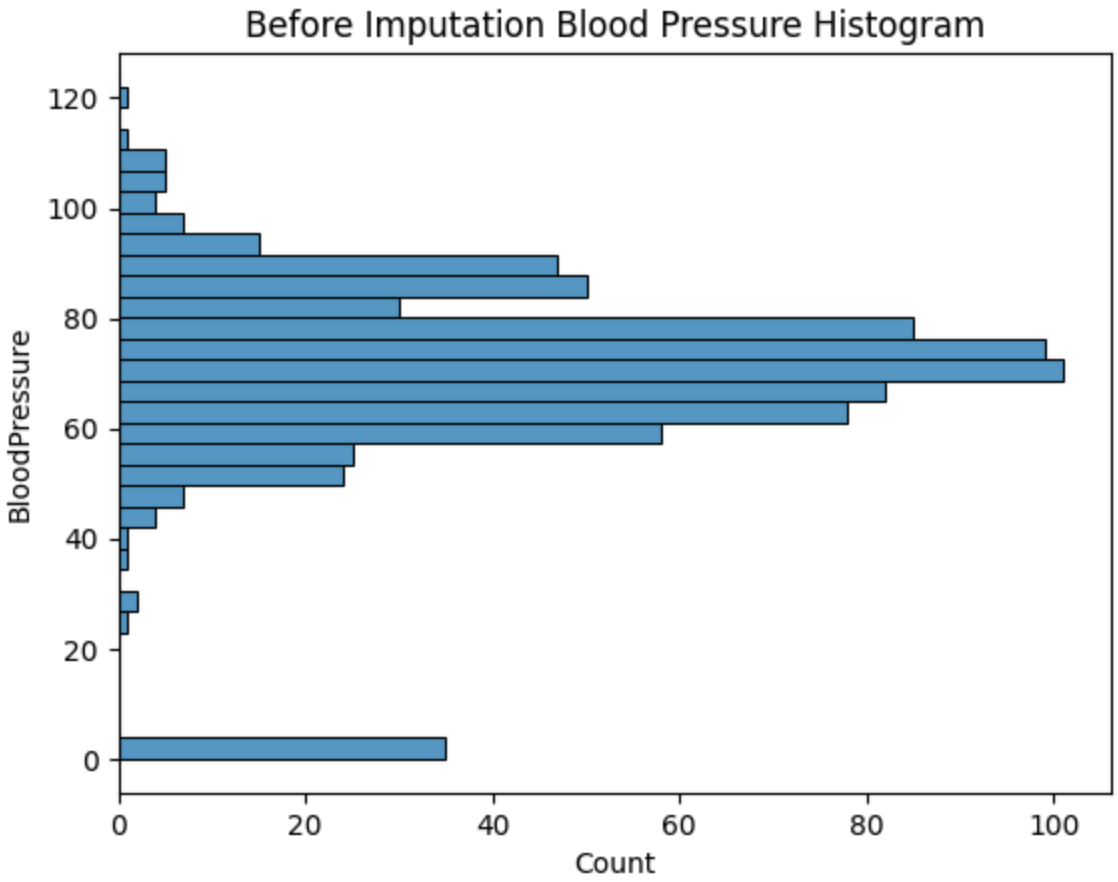

Scatter Plot Matrix of Diabetes Features
Pregnancies
Glucose
BloodPressure
SkinThickness
Insulin
BMI
DiabetesPedigreeFunction
Outcome
0
1

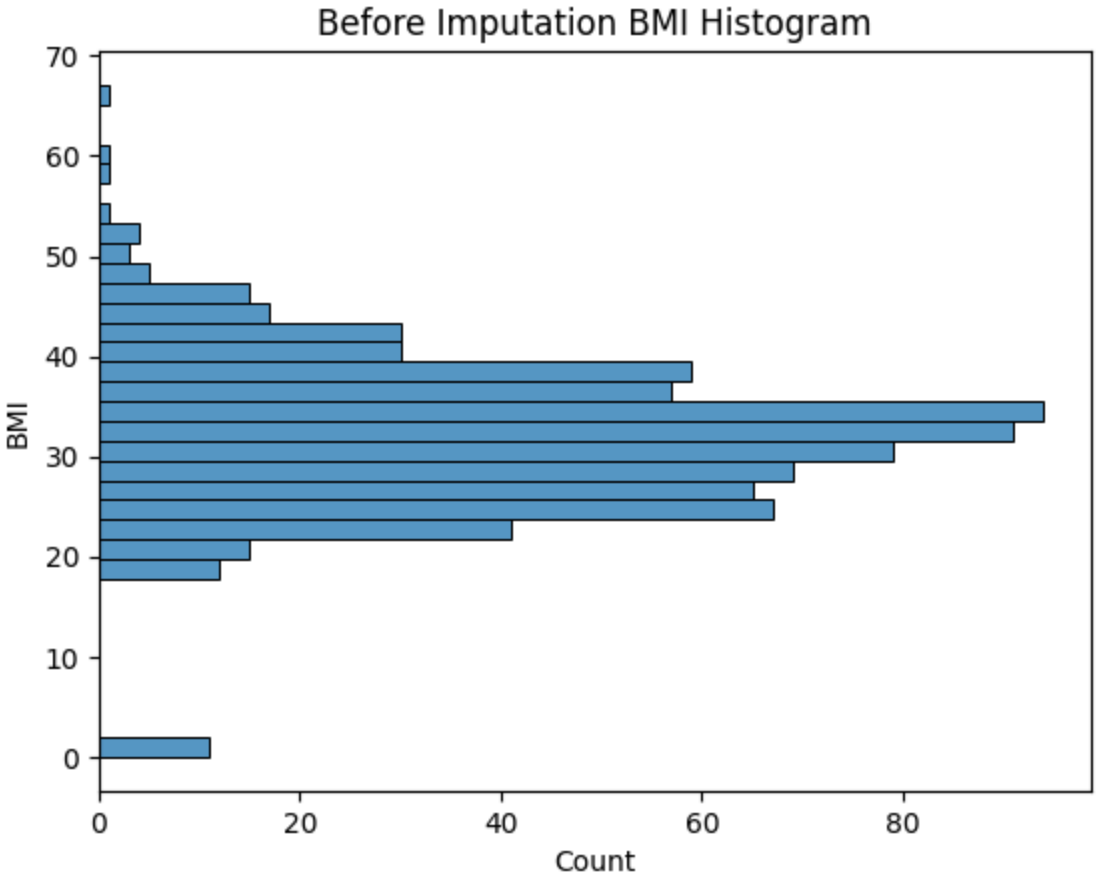

Age
Pregnancies
Glucose
BloodPressure
SkinThickness
Insulin
BMI
DiabetesPedigreeFunction
Age

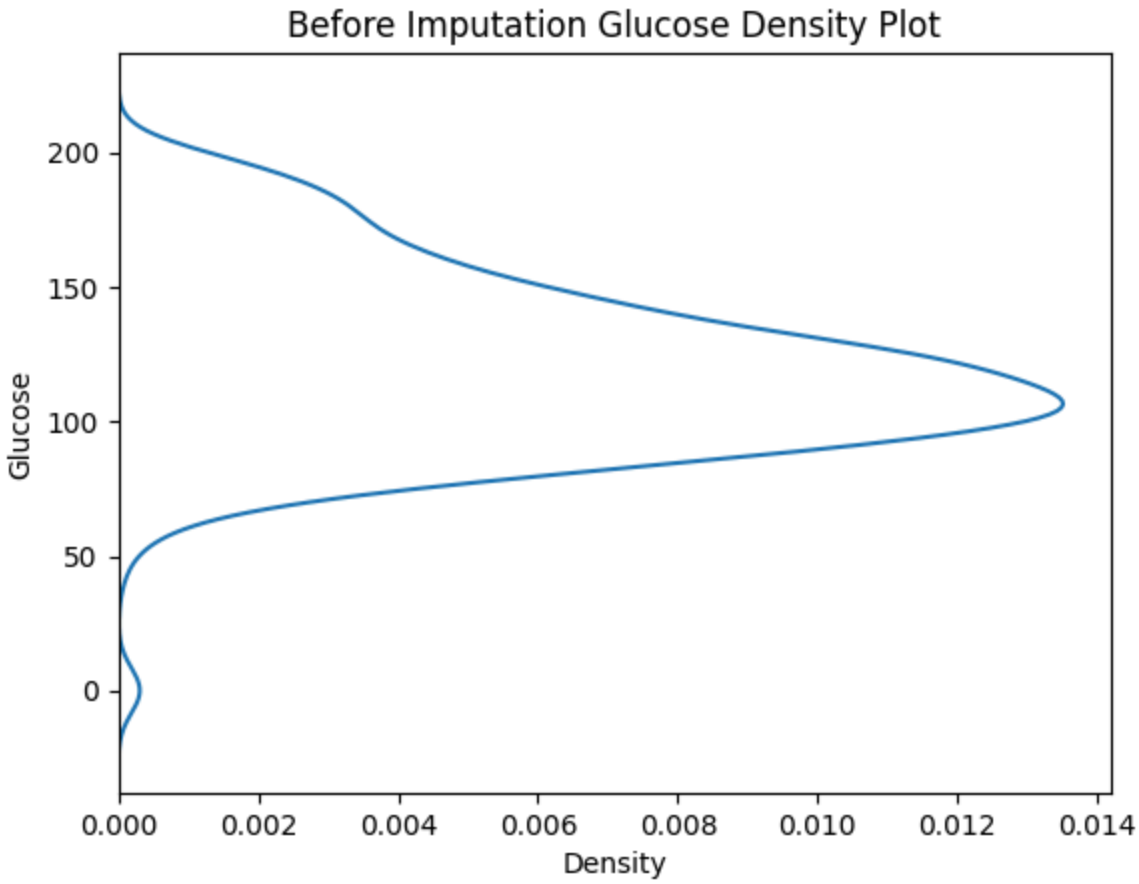

Box Plot of Diabetes Features
800
600
400
200
0
Pregnancies
Glucose
BloodPressure
SkinThickness
Insulin
BMI
DiabetesPedigreeFunction
Age

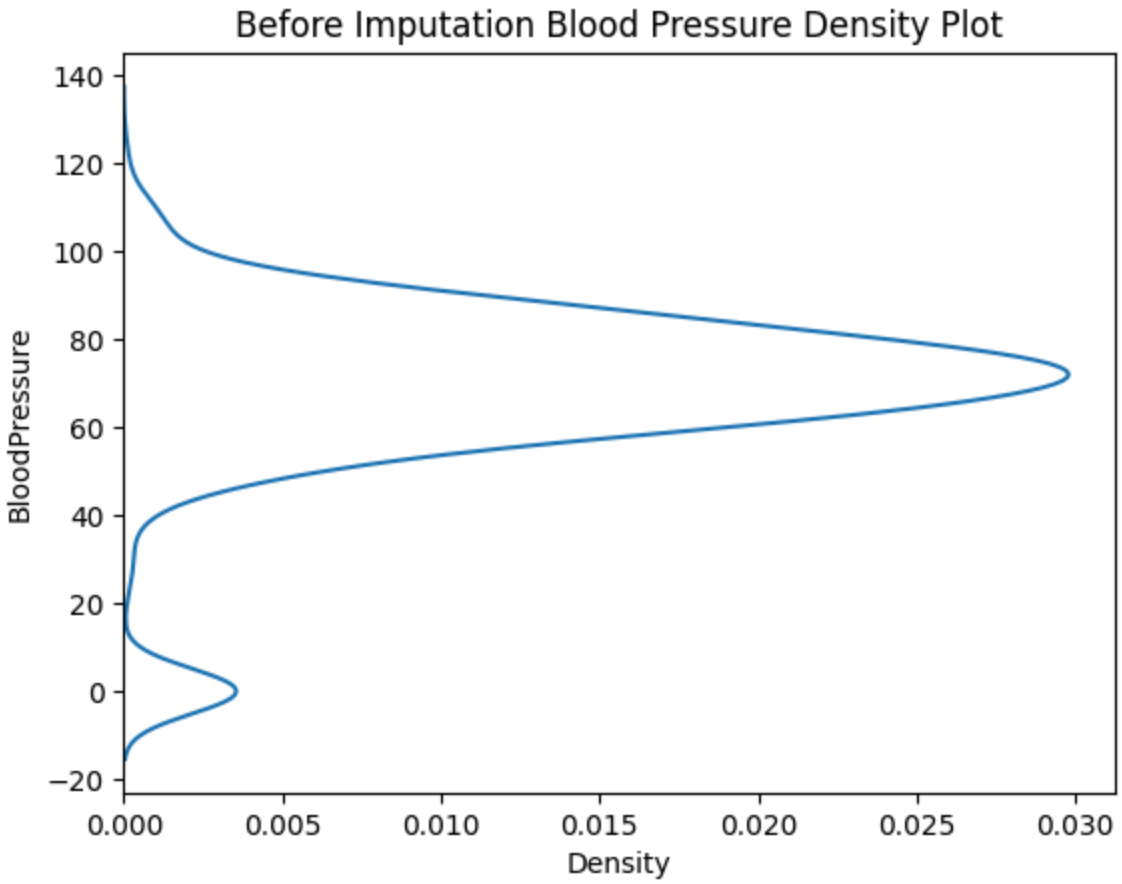

Correlation Heatmap of Diabetes Features
Pregnancies
Glucose
BloodPressure
SkinThickness
Insulin
BMI
DiabetesPedigreeFunction
Age
1 0.13 0.14 -0.082 -0.074 0.018 -0.034 0.54
0.13 1 0.15 0.057 0.33 0.22 0.14 0.26
0.14 0.15 1 0.21 0.089 0.28 0.041 0.24
-0.082 0.057 0.21 1 0.44 0.39 0.18 -0.11
-0.074 0.33 0.089 0.44 1 0.2 0.19 -0.042
0.018 0.22 0.28 0.39 0.2 1 0.14 0.036
-0.034 0.14 0.041 0.18 0.19 0.14 1 0.034
0.54 0.26 0.24 -0.11 -0.042 0.036 0.034 1
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

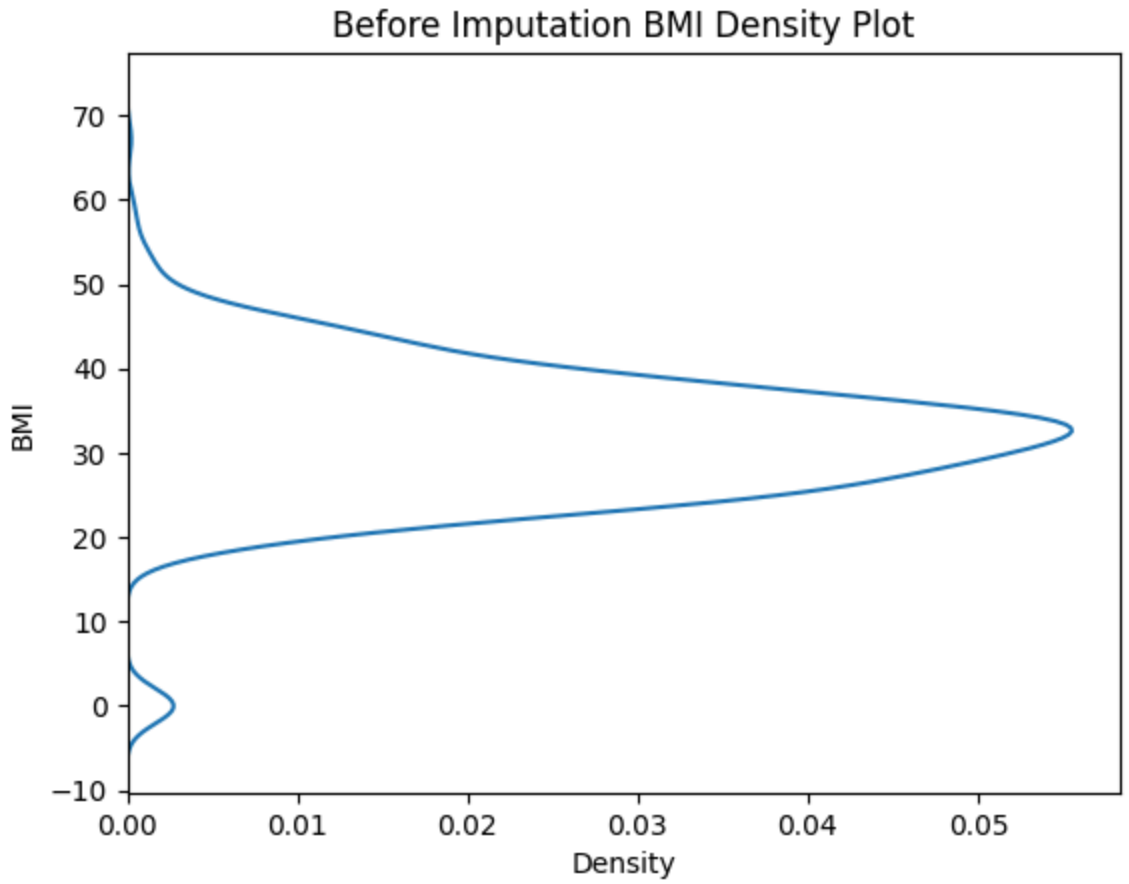

Before Imputation Glucose Histogram
Glucose
Count
200
175
150
125
100
75
50
25
0
0
20
40
60
80
100


Before Imputation Blood Pressure Histogram
BloodPressure
Count
120
100
80
60
40
20
0
0
20
40
60
80
100

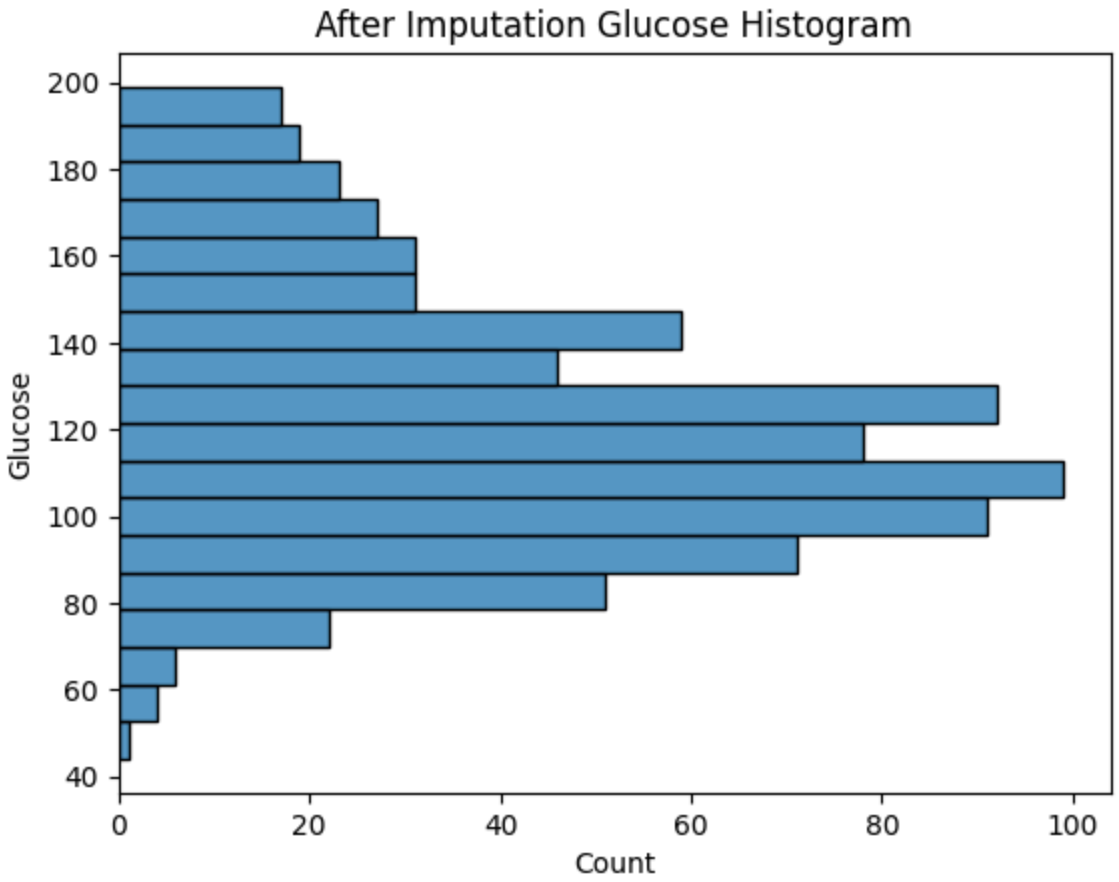

Before Imputation BMI Histogram
BMI
Count
70
60
50
40
30
20
10
0
0
20
40
60
80

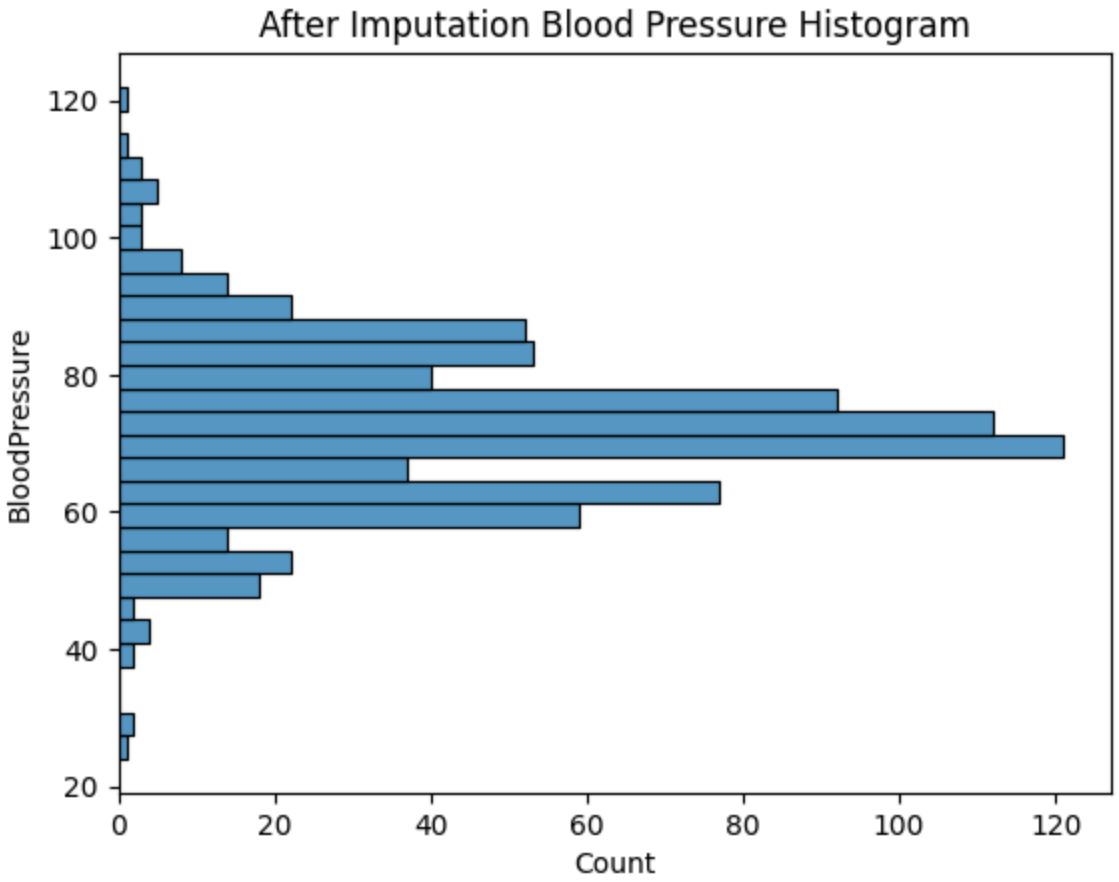

Before Imputation Glucose Density Plot
Glucose
200
150
100
50
0
0.000
0.002
0.004
0.006
0.008
0.010
0.012
0.014
Density

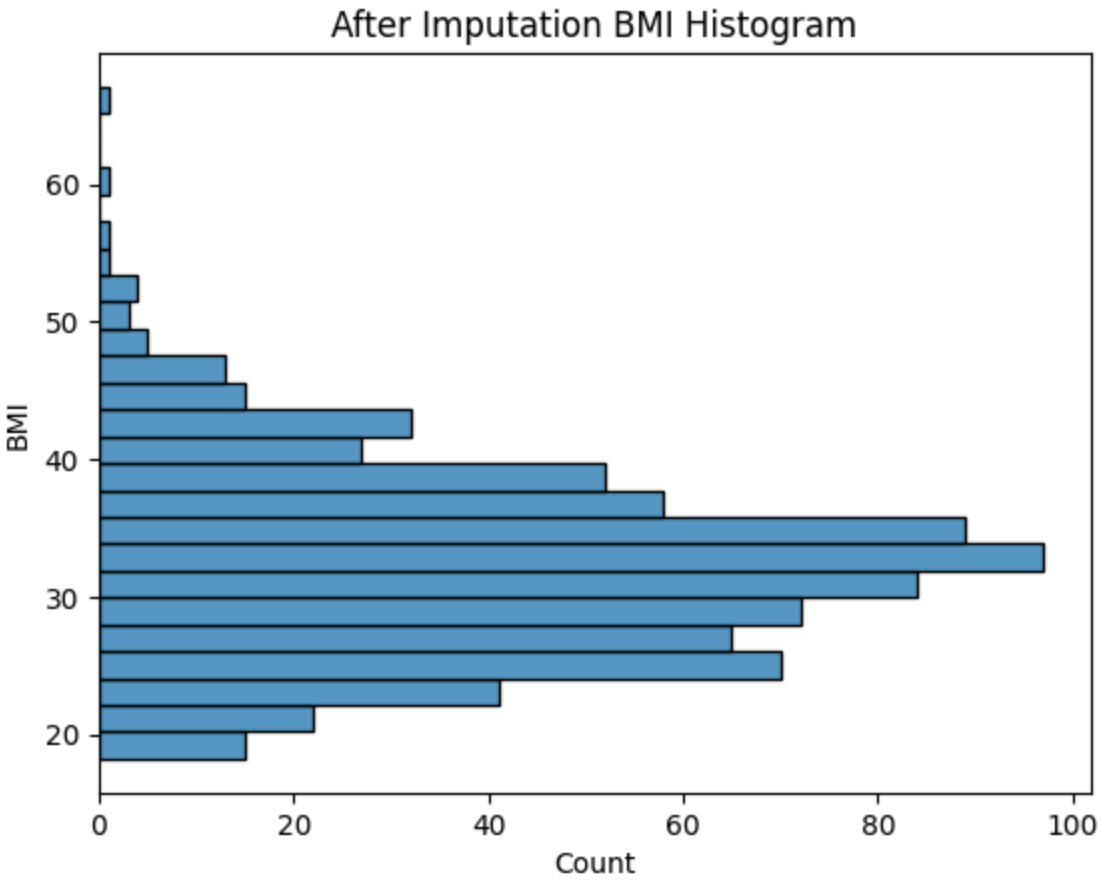

Before Imputation Blood Pressure Density Plot
BloodPressure
140
120
100
80
60
40
20
0
−20
0.000
0.005
0.010
0.015
0.020
0.025
0.030
Density

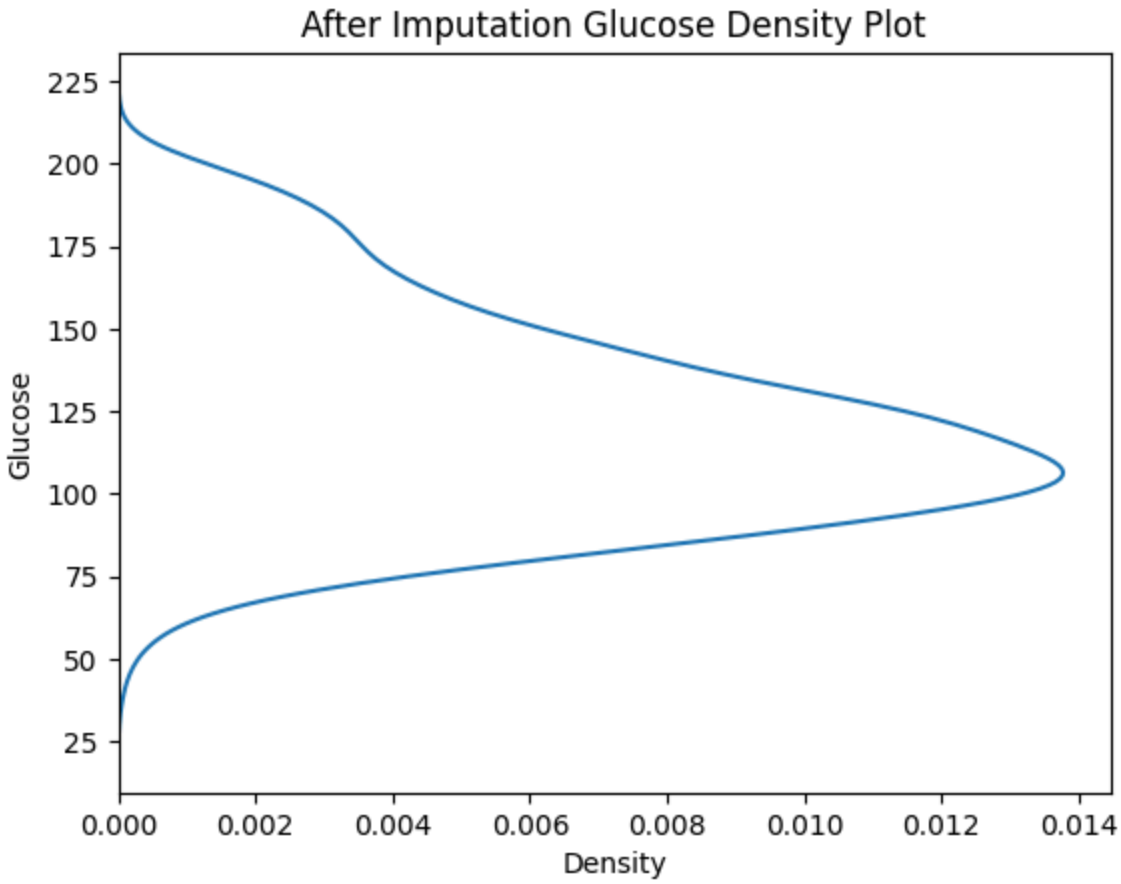


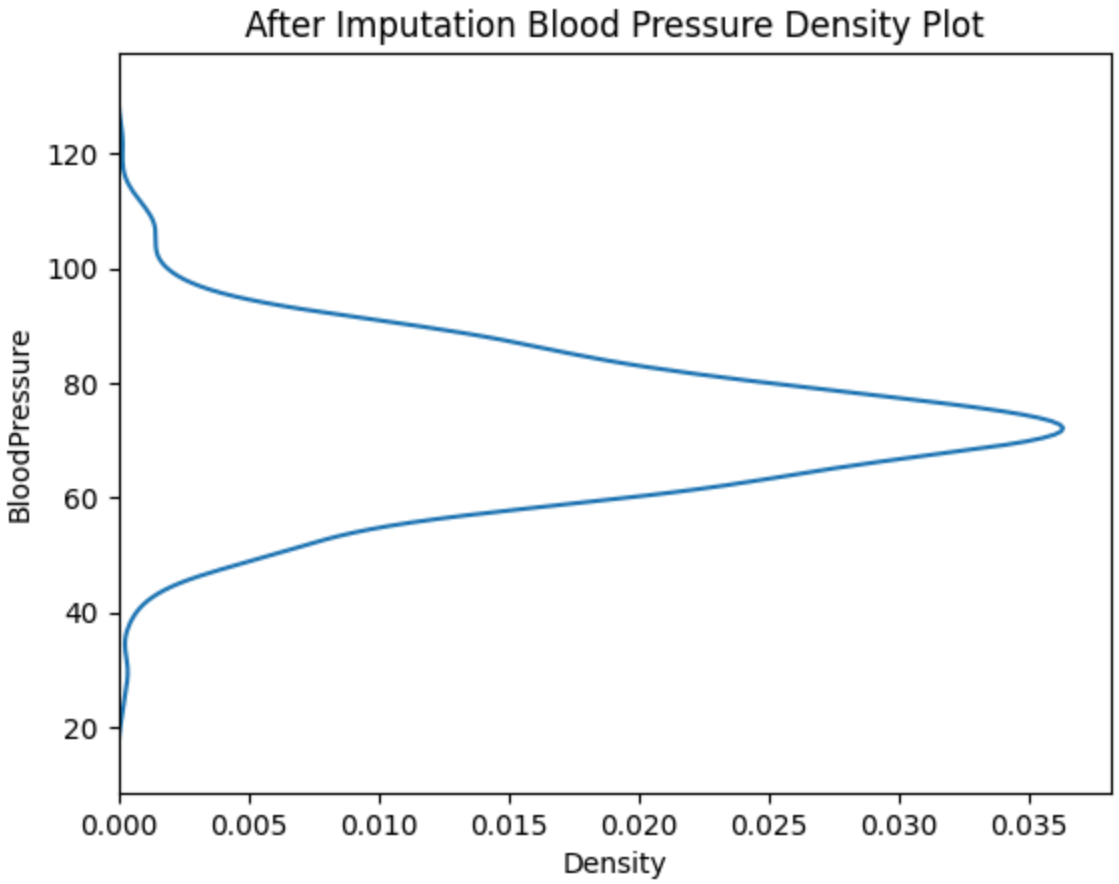

```
Invalid values:
44
```

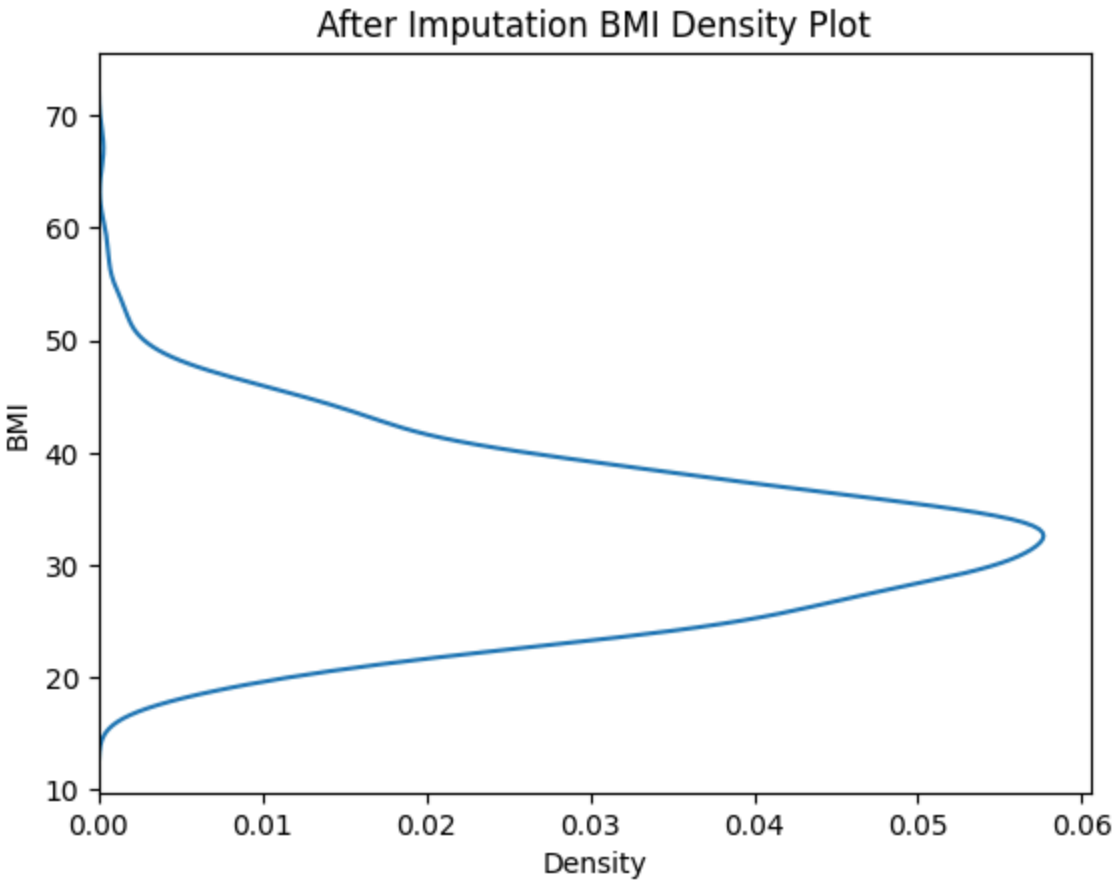

After Imputation Glucose Histogram
Glucose
200
180
160
140
120
100
80
60
40
0
20
40
60
80
100
Count


After Imputation Blood Pressure Histogram
BloodPressure
120
100
80
60
40
20
0
20
40
60
80
100
120
Count


After Imputation BMI Histogram
BMI
60
50
40
30
20
0
20
40
60
80
100
Count


After Imputation Glucose Density Plot
Glucose
225
200
175
150
125
100
75
50
25
0.000
0.002
0.004
0.006
0.008
0.010
0.012
0.014
Density


After Imputation Blood Pressure Density Plot
BloodPressure
120
100
80
60
40
20
0.000
0.005
0.010
0.015
0.020
0.025
0.030
0.035
Density



In [34]:
```python
#Question 2 Part 1 (Zhiyi)
# 1. [7 points] You need to implement KNN from scratch:
#a) [5 points] Implement Euclidean Distance from scratch (please, don't use built-in library).
#b) [2 points] Use Min-Max to normalize the dataset.
def euclidean_distance(row0, row1):
    return np.sqrt(np.sum((row0 - row1) ** 2))


def min_max_normalize(dataset):
    dataset_normalized = dataset.copy()
    for col in dataset.columns:
        min_val = dataset[col].min()
        max_val = dataset[col].max()
        dataset_normalized[col] = (dataset[col] - min_val) / (max_val - min_val)
```

```
    return dataset_normalized

min_max_normalize(df)
```

Out[34]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	0.352941	0.670968	0.489796	0.353535	0.000000	0.314928	0.234415	0.483333	1.0
1	0.058824	0.264516	0.428571	0.292929	0.000000	0.171779	0.116567	0.166667	0.0
2	0.470588	0.896774	0.408163	0.000000	0.000000	0.104294	0.253629	0.183333	1.0
3	0.058824	0.290323	0.428571	0.232323	0.111111	0.202454	0.038002	0.000000	0.0
4	0.000000	0.600000	0.163265	0.353535	0.198582	0.509202	0.943638	0.200000	1.0
...	...	...	...	...	...	...	...	...	...
763	0.588235	0.367742	0.530612	0.484848	0.212766	0.300613	0.039710	0.700000	0.0
764	0.117647	0.503226	0.469388	0.272727	0.000000	0.380368	0.111870	0.100000	0.0
765	0.294118	0.496774	0.489796	0.232323	0.132388	0.163599	0.071307	0.150000	0.0
766	0.058824	0.529032	0.367347	0.000000	0.000000	0.243354	0.115713	0.433333	1.0
767	0.058824	0.316129	0.469388	0.313131	0.000000	0.249489	0.101196	0.033333	0.0

768 rows × 9 columns

In [35]:

```
#2. [20 points] Tune the number of nearest neighbors k
# Reference: Data Science from Scratch --- Joel Grus
from typing import Counter

def majority_vote(labels, distances):
    vote_counts = Counter(labels)
    winner, winner_count = vote_counts.most_common(1)[0]
    num_winners = len([count
                       for count in vote_counts.values()
                       if count == winner_count])
    if num_winners == 1:
        return winner   #unique winner
    else:
```

```
        # In case of tie, KNN algorithm prefers the neighbor with closer distance to the query.
        closest_index = distances.index(min(distances))
        return labels[closest_index]

def knn_classify(k, labeled_points, new_point):
    # order the labeled points from nearest to farthest
    by_distance = sorted(labeled_points, key=lambda pair: euclidean_distance(pair[0], new_point))
    # find the labels for the k closest
    k_nearest_labels = [label for _, label in by_distance[:k]]
    k_nearest_distances = [euclidean_distance(pair[0], new_point) for pair in by_distance[:k]]  # K nearest distances
    # and let them vote
    return majority_vote(k_nearest_labels, k_nearest_distances)
```

In [36]:

```
# 5-fold cross validation
# reference: https://www.geeksforgeeks.org/cross-validation-using-k-fold-with-scikit-learn/
from sklearn.model_selection import KFold


def cross_validate_knn(df, k_values=[1,3,5,7]):
    X = df.iloc[:, :-1].values   #Features
    y = df.iloc[:, -1].values   #Labels
    kf = KFold(n_splits = 5, shuffle = True, random_state = 42)
    accuracies = {k: [] for k in k_values}
    for train_idx, val_idx in kf.split(X):
        X_train, X_val = X[train_idx], X[val_idx]
        y_train, y_val = y[train_idx], y[val_idx]

        labeled_train = list(zip(X_train, y_train))

        for k in k_values:
            predictions = [knn_classify(k, labeled_train, x) for x in X_val]
            accuracy = np.mean(predictions == y_val)
            accuracies[k].append(accuracy)

    #compute mean accuracy for each K value
    avg_accuracies = {k:np.mean(v) for k,v in accuracies.items()}

    #select best K
    best_k = max(avg_accuracies, key = avg_accuracies.get)

    return best_k, avg_accuracies
```

```
# find optimal k using 5-fold Cross Validation

best_k, accuracy_results = cross_validate_knn(df)
print(f"Optimal K: {best_k}")
print(accuracy_results)
```

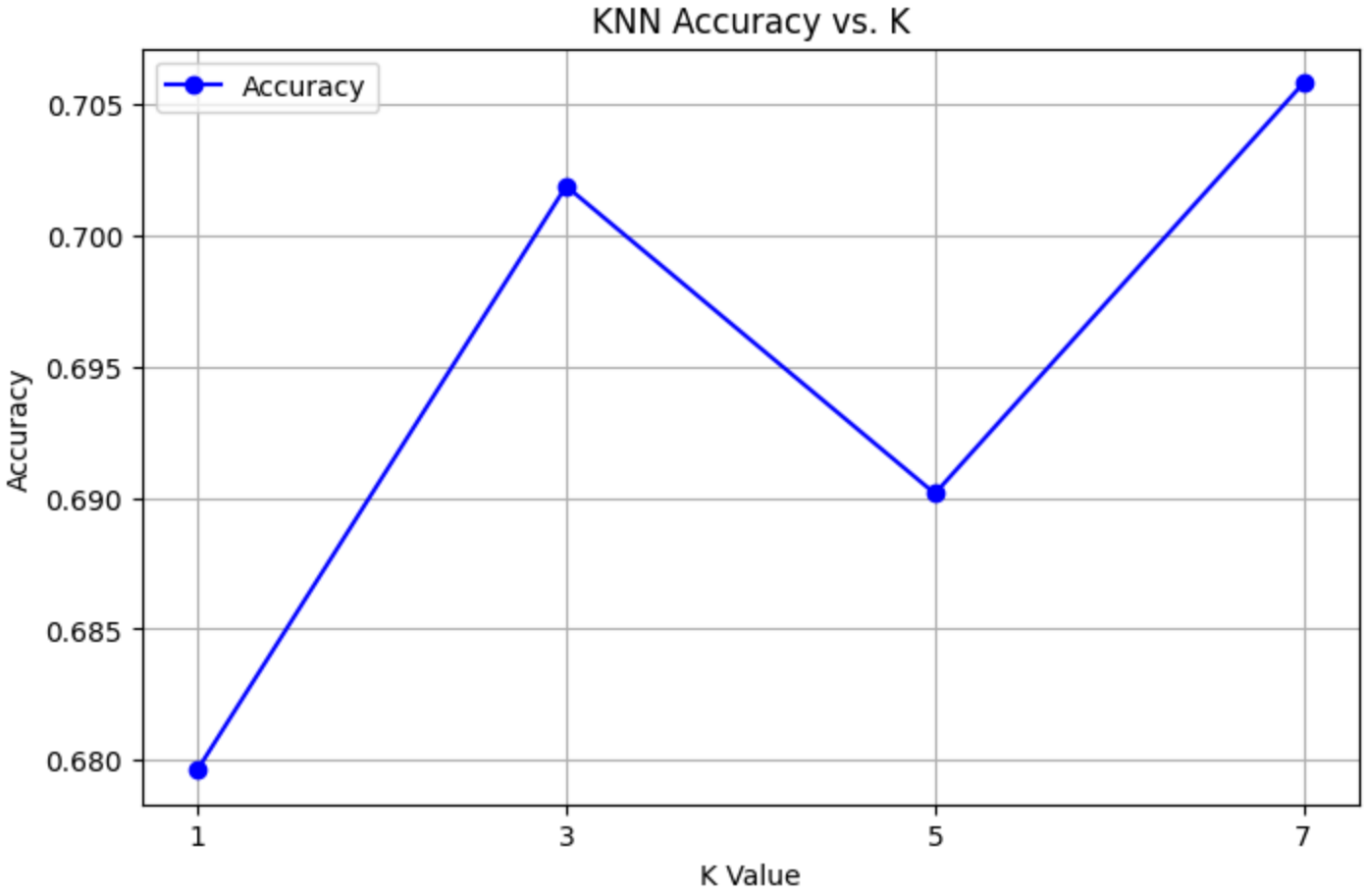
```
Optimal K: 7
{1: 0.6796367031661149, 3: 0.7018589253883372, 5: 0.6901536372124608, 7: 0.7057889822595704}
```

In [ ]:
```
# K value vs. Accuracy: draw a 2D plot to show the accuracy of KNN classifier vs different number of k's.
from sklearn.model_selection import train_test_split

# split dataset (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(df.iloc[:, :-1], df.iloc[:, -1], test_size=0.2)

# prepare labeled training points
labeled_train = list(zip(X_train.values, y_train.values))

# use optimal k = 7 found through cross-validation
predictions = [knn_classify(best_k, labeled_train, x) for x in X_test.values]

# Compute Accuracy
final_accuracy = np.mean(predictions == y_test)
print(f"Final Test Accuracy using K={best_k}: {final_accuracy:.4f}")
```

```
Final Test Accuracy using K=7: 0.7403
```

In [ ]:
```
# K value vs. Accuracy: draw a 2D plot to show the accuracy of KNN classifier vs different number of k's.
# # reference: co-pilot (I used co-pilot to look for the functions to look for the steps to create the plot
# because I'm not familiar with matplot and figure the usage should be straightforward) and
# https://stackoverflow.com/questions/45075638/graph-k-nn-decision-boundaries-in-matplotlib

import matplotlib.pyplot as plt

# Extract K values and corresponding accuracies
k_values = list(accuracy_results.keys())
accuracies = list(accuracy_results.values())

# Create the plot
plt.figure(figsize=(8,5))
plt.plot(k_values, accuracies, marker='o', linestyle='-', color='b', label="Accuracy")

# Labels and title
```

```
plt.xlabel("K Value")
plt.ylabel("Accuracy")
plt.title("KNN Accuracy vs. K")
plt.xticks(k_values)
plt.grid()
plt.legend()
plt.show()
```



In [ ]:
```
# Comment on the result and print misclassified records in table format where you show the true label in one column a
# label in another column.
# reference: co-pilot (same reason as above --- used co-pilot to look for the steps because I was unfamiliar, and the
# reference https://github.com/scikit-learn/scikit-learn/issues/18533)
import pandas as pd
```

```
# Create a DataFrame showing misclassified instances
misclassified = pd.DataFrame({'True Label': y_test, 'Predicted Label': predictions})

# Filter misclassified cases
misclassified_records = misclassified[misclassified['True Label'] != misclassified['Predicted Label']]

# Print the table
print(misclassified_records)
```

	True Label	Predicted Label
660	0	1
336	0	1
276	1	0
419	1	0
667	1	0
433	0	1
254	1	0
387	1	0
657	0	1
541	1	0
645	0	1
184	0	1
444	1	0
72	1	0
153	0	1
755	1	0
95	0	1
246	0	1
728	0	1
659	1	0
166	0	1
622	0	1
272	0	1
638	1	0
405	0	1
107	0	1
506	1	0
335	0	1
270	1	0
218	1	0
473	0	1
673	0	1
569	1	0
542	1	0
286	0	1
560	1	0
304	0	1
54	0	1
212	0	1
44	0	1

In [40]:
```
# Question 2 Part 2 (Lane)
print("\nQuestion 2 Part 2: Predicting the onset of diabetes based on diagnostic measures.")
# fit a Gaussian Naive Bayes model to the training data
from sklearn.naive_bayes import GaussianNB # https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.Ga
model = GaussianNB()
model.fit(X_train, y_train)
```

```
Question 2 Part 2: Predicting the onset of diabetes based on diagnostic measures.
```

Out[40]:

GaussianNB

```
GaussianNB()
```

In [41]:
```
# Q2P2 Accuracy
from sklearn.metrics import accuracy_score
y_pred = model.predict(X_test)
naive_accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of Naive Bayes model: {naive_accuracy:.4f}")
```

```
Accuracy of Naive Bayes model: 0.7662
```

In [42]:
```
# Q2P2 Show misclassified samples
misclassified_diabetes = X_test.copy()
misclassified_diabetes['true_outcome'] = y_test
misclassified_diabetes['predicted_outcome'] = y_pred
misclassified_diabetes = misclassified_diabetes[y_test != y_pred]
print("\nMisclassified Samples (Diabetes Prediction):")
print(misclassified_diabetes[['true_outcome', 'predicted_outcome']])
```

Misclassified Samples (Diabetes Prediction):

	true_outcome	predicted_outcome
660	0	1
276	1	0
356	1	0
419	1	0
667	1	0
364	0	1
337	1	0
657	0	1
541	1	0
645	0	1
451	1	0
710	0	1
379	0	1
444	1	0
264	1	0
757	1	0
153	0	1
95	0	1
594	0	1
371	0	1
659	1	0
577	1	0
622	0	1
139	0	1
638	1	0
750	1	0
280	1	0
335	0	1
267	0	1
218	1	0
673	0	1
569	1	0
286	0	1
560	1	0
54	0	1
212	0	1

[12 points] Model Evaluation

Compare, comment, and analyze the result of all the classifiers you built thus far Naïve Bayes & KNN using built-in library (you implemented in Question 1) with KNN from scratch implementation in Question 2 on ***Pima Indians Diabetes Database*** data. To compare the performance of your classifier based on:

I. **[6 points]** execution time.
II. **[6 points]** Accuracy

In [43]:
```
# (Lane)
training_data = X_train
training_label = y_train
test_data = X_test
test_label = y_test
best_k = best_k  # Use the best k found from cross-validation
# Q1 KNN from library
import time
from sklearn.neighbors import KNeighborsClassifier # https://scikit-learn.org/stable/modules/generated/sklearn.neighb
knn_library = KNeighborsClassifier(n_neighbors=best_k, metric='euclidean')
time_start = time.time()
knn_library.fit(training_data, training_label)
time_end = time.time()
print(f"KNN from Library Training Time: {time_end - time_start:.4f} seconds")
test_accuracy = knn_library.score(test_data, test_label)
print(f"KNN from Library Test Accuracy (k={best_k}): {test_accuracy*100:.2f} %\n")
# Q1 Naive Bayes from library
nb_library = GaussianNB()
time_start = time.time()
nb_library.fit(training_data, training_label)
time_end = time.time()
print(f"Naive Bayes from Library Training Time: {time_end - time_start:.4f} seconds")
test_accuracy_nb = nb_library.score(test_data, test_label)
print(f"Naive Bayes from Library Test Accuracy: {test_accuracy_nb*100:.2f} %\n")
# Q2P1 KNN from scratch accuracy
time_start = time.time()
labeled_train = list(zip(test_data.values, test_label.values))
```

```
knn_accuracy = np.mean([knn_classify(best_k, labeled_train, x) == y for x, y in zip(test_data.values, test_label.valu
time_end = time.time()
print(f"KNN from Scratch Training Time: {time_end - time_start:.4f} seconds")
print(f"KNN from Scratch Test Accuracy (k={best_k}): {knn_accuracy*100:.2f} %")
```

```
KNN from Library Training Time: 0.0015 seconds
KNN from Library Test Accuracy (k=7): 74.03 %

Naive Bayes from Library Training Time: 0.0020 seconds
Naive Bayes from Library Test Accuracy: 76.62 %

KNN from Scratch Training Time: 0.1240 seconds
KNN from Scratch Test Accuracy (k=7): 77.27 %
```

Results from Comparison

After comparing Naive Bayes and KNN from libraries and the KNN built from scratch, the library implementations run significantlly faster. This isn't surprising because they use C on the backend. With accuracy, though, the KNN built from scratch achieves better test accuracy then either of the library. This looks great, but we would need to do more testing to rule out overffiting.